Inna Pereverzeva

Formal Development of Resilient Distributed Systems

Turku Centre *for* Computer Science

TUCS Dissertations
No 203, October 2015

Formal Development of Resilient Distributed Systems

Inna Pereverzeva

To be presented, with the permission of the Faculty of Science and Engineering of the Åbo Akademi University, for public criticism in Auditorium Gamma on October 2, 2015, at 12 noon.

Turku Centre for Computer Science
Åbo Akademi University
Faculty of Science and Engineering
Joukahaisenkatu 3-5, 20520 Turku
Finland

2015

Supervisors

Associated Professor Elena Troubitsyna
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

Adjunct Professor Linas Laibinis
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

Reviewers

Professor Giovanna Di Marzo Serugendo
Institute of Information Services Science
University of Geneva
Battelle - Bâtiment A
Rte de Drize 7 CH-1227 Carouge
Switzerland

Dr. Michael R. Poppleton
ECS, Faculty of Physical Sciences and Engineering
University of Southampton
Southampton, United Kingdom
SO17 1BJ

Opponent

Professor Giovanna Di Marzo Serugendo
Institute of Information Services Science
University of Geneva
Battelle - Bâtiment A
Rte de Drize 7 CH-1227 Carouge
Switzerland

ISBN 978-952-12-3253-4
ISSN 1239-1883
Painosalama Oy, Turku

To my dearest Mum and Dad

Моим дорогим маме и папе

A journey of a thousand miles begins with a single step.

– *Laozi*

Abstract

Resilience is the property of a system to remain trustworthy despite changes. Changes of a different nature, whether due to failures of system components or varying operational conditions, significantly increase the complexity of system development. Therefore, advanced development technologies are required to build robust and flexible system architectures capable of adapting to such changes. Moreover, powerful quantitative techniques are needed to assess the impact of these changes on various system characteristics.

Architectural flexibility is achieved by embedding into the system design the mechanisms for identifying changes and reacting on them. Hence a resilient system should have both advanced monitoring and error detection capabilities to recognise changes as well as sophisticated reconfiguration mechanisms to adapt to them. The aim of such reconfiguration is to ensure that the system stays operational, i.e., remains capable of achieving its goals.

Design, verification and assessment of the system reconfiguration mechanisms is a challenging and error prone engineering task. In this thesis, we propose and validate a formal framework for development and assessment of resilient systems. Such a framework provides us with the means to specify and verify complex component interactions, model their cooperative behaviour in achieving system goals, and analyse the chosen reconfiguration strategies. Due to the variety of properties to be analysed, such a framework should have an integrated nature. To ensure the system functional correctness, it should rely on formal modelling and verification, while, to assess the impact of changes on such properties as performance and reliability, it should be combined with quantitative analysis.

To ensure scalability of the proposed framework, we choose Event-B as the basis for reasoning about functional correctness. Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has a mature industrial-strength tool support – the Rodin platform. Proof-based verification as well as the reliance on abstraction and decomposition adopted in Event-B provides the designers with a powerful support for the development of complex systems. Moreover, the top-down system development

by refinement allows the developers to explicitly express and verify critical system-level properties.

Besides ensuring functional correctness, to achieve resilience we also need to analyse a number of non-functional characteristics, such as reliability and performance. Therefore, in this thesis we also demonstrate how formal development in Event-B can be combined with quantitative analysis. Namely, we experiment with integration of such techniques as probabilistic model checking in PRISM and discrete-event simulation in SimPy with formal development in Event-B. Such an integration allows us to assess how changes and different reconfiguration strategies affect the overall system resilience. The approach proposed in this thesis is validated by a number of case studies from such areas as robotics, space, healthcare and cloud domain.

Sammanfattning

Resiliens är ett systems egenskap för att förbli pålitligt trots förändringar. Förändringar av olika karaktär, antingen på grund av fel i systemkomponenter eller varierande driftsförhållanden, ökar avsevärt på komplexiteten i systemutveckling. Därför behövs avancerade utvecklingsteknologier för att bygga robusta och flexibla systemarkitekturer som kan anpassa sig till sådana förändringar. Dessutom behövs kraftfulla kvantitativa metoder för att bedöma effekterna av dessa förändringar på olika systemegenskaper.

Arkitektonisk flexibilitet uppnås genom att bygga in mekanismer i systemdesignen för att identifiera förändringar och reagera på dem. Därför bör ett resilient system ha både avancerad övervaknings- och fel upptäckningsförmåga för att känna igen förändringar samt sofistikerade omkonfigureringsmekanismer för att anpassa sig till dem. Syftet med en sådan omkonfigurering är att se till att systemet förblir i drift, det vill säga, fortfarande kan uppnå sina mål.

Design, kontroll och utvärdering av systemets omkonfigureringsmekanismer är en utmanande och felbenägen ingenjörsuppgift. I denna avhandling föreslår vi och validerar ett formellt ramverk för utveckling och utvärdering av resilienta system. Ett sådant ramverk ger oss möjlighet att specificera och verifiera komplexa komponentinteraktioner, modellera deras kooperativa beteende för att uppnå systemets mål och analysera de valda omkonfigureringstrategierna. På grund av de olika egenskaper som skall analyseras, bör ett sådant ramverk har en integrerad karaktär. För att säkerställa systemets funktionella korrekthet bör det bygga på formell modellering och verifiering, men för att också kunna bedöma effekterna av förändringar på sådana egenskaper som prestanda och tillförlitlighet bör det kombineras med kvantitativ analys.

För att säkerställa skalbarhet av det föreslagna ramverket, väljer vi Event-B som grund för resonemang om funktionell korrekthet. Event-B är ett tillståndsbaserat formellt metod som främjar correct-by-construction utvecklingsparadigmen för att uppnå korrekthet och formell verifiering med hjälp av teorembevisning. Event-B har ett välutvecklat verktygsstöd med industriell styrka – Rodin-plattformen. Bevis-baserad verifiering samt beroendet av abstraktion och problem uppdelning som används i Event-B ger de-

signers ett kraftfullt stöd för utveckling av komplexa system. Dessutom tillåter top-down systemutvecklingen med precisering utvecklarna att uttryckligen beskriva och verifiera kritiska på systemnivå egenskaper.

Förutom att säkerställa funktionell korrekthet, måste vi för att uppnå resiliens också analysera ett antal icke-funktionella egenskaper såsom tillförlitlighet och prestanda. Därför visar vi i denna avhandling också hur formell utveckling i Event-B kan kombineras med kvantitativ analys. Nämligen, vi experimenterar med integrering av sådana tekniker som probabilistiskt model-checking modellprövning i PRISM och diskret simulering i SimPy med formell utveckling i Event-B. En sådan integration ger oss möjlighet att bedöma hur förändringar och olika omkonfigureringsstrategier påverkar det övergripande systemets resiliens. Den metod som föreslås i avhandlingen valideras med ett antal fallstudier från sådana områden som robotik, flygindustri, sjukvård och molntjänsk.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to both my supervisors, Associated Prof. Elena Troubitsyna and Adjunct Prof. Linas Laibinis. Without you, I could never have done this thesis. Your ideas, enthusiasm, and constant support were always encouraging factors during my work. I am thankful for believing in my potential as well as for our fruitful discussions and the nice moments we had working together. I would also like to wholeheartedly thank you for all your help in writing this thesis.

I am most grateful to Prof. Giovanna Di Marzo Serugendo and Dr. Michael Poppleton who kindly accepted to review this thesis and whose valuable comments improved both quality and readability of its introductory part. I am very thankful to Prof. Giovanna Di Marzo Serugendo for also agreeing to act as an opponent at the public defence of the thesis.

I extend my sincere thanks to all members of the Department of Information Technologies at Åbo Akademi University and Turku Centre for Computer Science for providing excellent working environment and friendly atmosphere. I would like to express my gratitude to my colleagues from the Embedded system laboratory. In particular, I am thankful to Prof. Johan Lilius, Dr. Sébastien Lafond, Dr. Leonidas Tsiopoulos, Dr. Johan Ersfolk, Dr. Yuliya Prokhorova, John-Eric Saxén, Wictor Lund, Stefan Grönroos, Simon Holmbacka, Sudeep Kanur, and Georgios Georgakarakos. Additionally, I would like to thank all the members of the Distributed Systems Laboratory. In particular, I am thankful to Associated Prof. Marina Waldén, Adjunct Prof. Luigia Petre, Dr. Pontus Boström, Dr. Marta Olszewska, Dr. Mats Neovius, Dr. Maryam Kamali, Dr. Sergey Ostroumov, Petter Sandvik, Jonatan Wiik, and Mojgan Kamali.

I am especially grateful to Dr. Anton Tarasyuk for his great support, advices and encouragement throughout my PhD studies. I wish to thank Johan Ersfolk and Pontus Boström, who kindly agreed to help me with the Swedish version of the abstract. I am very grateful to all my co-authors for the knowledge they shared with me. In particular, I wish to thank Prof. Michael Butler, Dr. Asieh Salehi Fathabadi, Benjamin Byholm, Prof. Ivan Porres, Dr. Timo Latvala, Laura Nummila, Marcus Holmberg, Mikko Pöri and Kuan Eeik.

I would like to acknowledge the generous financial support provided to me by the Department of Information Technologies at Åbo Akademi University and Turku Centre for Computer Science. I am also grateful and honoured for the research scholarships and grants I have received from the Nokia Foundation, Svenska tekniska vetenskapsakademien i Finland, the Ulla Tuominen Foundation, and the Academy of Finland.

Last but not the least, I would like to thank my family and all my friends (from Åbo Akademi and not only) for their continued support and cheeriness throughout these years. Without you, all my life would be incomplete. Above all, my deepest thanks go to my dearest parents, Galina and Yuri. Thank you for your love, tender care and endless trust that you gave me. This dissertation is dedicated to you.

Inna Pereverzeva
Turku, August 2015

List of original publications

I Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, Timo Latvala and Laura Nummila, Formal Development and Assessment of a Reconfigurable On-board Satellite System. In: Frank Ortmeier, Peter Daniel (Eds.), *Proceedings of 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2012)*, LNCS 7612, 210–222, Springer, 2012.

II Inna Pereverzeva, Michael J. Butler, Asieh Salehi Fathabadi, Linas Laibinis and Elena Troubitsyna, Formal Derivation of Distributed MapReduce. In: Yamine Aït Ameur, Klaus-Dieter Schewe (Eds.), *Proceedings of 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, LNCS 8477, 238–254, Springer-Verlag Berlin Heidelberg, 2014.

III Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis, Formal Development of Critical Multi-Agent Systems: A Refinement Approach. In: Cristian Constantinescu, Miguel P. Correia (Eds.), *Proceedings of 9th European Dependable Computing Conference (EDCC 2012)*, 156–161, IEEE Computer Society, 2012.

IV Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis, A Refinement-Based Approach to Developing Critical Multi-Agent Systems. In: *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 1, 69–91, 2013.

V Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis, Formal Goal-Oriented Development of Resilient MAS in Event-B. In: Mats Brorsson, Luis Miguel Pinho (Eds.), *Proceedings of 17th International Conference on Reliable Software Technologies (Ada-Europe 2012)*, LNCS 7308, 147–161, Springer-Verlag Berlin Heidelberg, 2012.

VI Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis, A Case Study in Formal Development of a Fault Tolerant Multi-Robotic System. In: Paris Avgeriou (Ed.), *Proceedings of 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, LNCS 7527, 16–31, Springer-Verlag Berlin Heidelberg, 2012.

VII Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis, Formal Development and Quantitative Assessment of a Resilient Multi-robotic System. In: Anatoliy Gorbenko, Alexander Romanovsky, Vyacheslav S. Kharchenko (Eds.), *Proceedings of 5th International Workshop on Software Engineering for Resilient Systems (SERENE 2013)*, LNCS 8166, 109–124, Springer-Verlag Berlin Heidelberg, 2013.

VIII Linas Laibinis, Inna Pereverzeva and Elena Troubitsyna, Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems. *TUCS Technical Report 1133*, TUCS, April 2015. (Submitted to *Formal Aspects of Computing Journal*)

IX Inna Pereverzeva, Elena Troubitsyna, Linas Laibinis, Marcus Holmberg and Mikko Pöri, Formal Modelling of Resilient Data Storage in Cloud. In: Lindsay Groves and Jing Sun (Eds.), *Proceedings of 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, LNCS 8144, 363–379, Springer-Verlag Berlin Heidelberg, 2013.

X Linas Laibinis, Benjamin Byholm, Inna Pereverzeva, Elena Troubitsyna, Kuan Eeik Tan and Ivan Porres, Integrating Event-B Modelling and Discrete-Event Simulation to Analyse Resilience of Data Stores in the Cloud. In: Elvira Albert, Emil Sekerinski (Eds.), *Proceedings of 11th International Conference on Integrated Formal Methods (iFM 2014)*, LNCS 8739, 103–119, Springer International Publishing Switzerland, 2014.

Contents

Part I

Research Summary

Chapter 1

Motivation and Research Objectives

Modern software systems are becoming more complex, diverse, distributed and heterogeneous. Moreover, often they operate in dynamically changing environments in a highly autonomous manner. Since computer-based software-intensive systems are increasingly used in various safety- and money-critical domains, we need to create powerful development techniques ensuring their trustworthiness, i.e., guaranteeing that an incorrect system behaviour would not result in loss of human life, damage to the environment, or devastating economic consequences.

The concept of resilience [106] has been introduced to represent an ability of a system to deliver its services in a trustworthy way despite changes. It is a generic concept that essentially expresses the demand for the systems that can reliably adapt to changes caused by a broad spectrum of reasons – from component failures to the need to utilise the available resources more efficiently.

The design for resilience poses a significant engineering challenge and requires novel development methods and tools that are versatile, powerful and scalable enough to embrace the complexity of building adaptable yet safe and reliable systems. They should facilitate building robust and reconfigurable systems as well as assess the impact of changes on various system characteristics. Moreover, they should allow us to specify and verify complex component interactions, model their cooperative behaviour in achieving system goals, and analyse the chosen system reconfiguration strategies.

Resilience is a multi-facet system property. Therefore, to develop resilient systems, we need to combine different techniques for modelling and analysis into an integrated approach that can address various design needs. To ensure functional correctness, such an approach should rely on formal modelling and verification, while, to assess the impact of changes, it should

be combined with quantitative analysis.

The main objective of this thesis is to propose a scalable integrated approach to development of complex resilient distributed systems. It should satisfy the following criteria:

- to support rigorous development of such systems in a systematic manner, while enabling reasoning about the system behaviour at different levels of abstraction – from the top-level goals to component interactions;
- to guarantee functional correctness of complex distributed systems, with the incorporated mechanisms to ensure system resilience;
- to facilitate systematic construction of complex reconfigurable architectures as the means to enhance system resilience;
- to provide the developers with a powerful platform for quantitative assessment of resilience and finding optimal trade-offs between system reliability and performance.

We also intend to validate the proposed approach in a number of critical domains to ensure its benefits for development of resilient distributed systems.

Organisation of the thesis. The thesis consists of two parts. Part I contains an overview of the research presented in the thesis, while Part II consists of reprints of the original research publications. Part I is structured as follows. In Chapter 2 we introduce the main concepts that we rely on in this thesis: resilience, goal-oriented development and reconfiguration. In Chapter 3 we give an overview of our modelling and assessment techniques – the Event-B method, the probabilistic PRISM model checker and discrete event simulation framework in SimPy. In Chapter 4 we present our main contributions – a formal approach to development of resilient reconfigurable systems and integrated quantitative resilience assessment. In Chapter 5 we give a detailed description of the research results published in each original paper. In Chapter 6 we overview the related work. Finally, in Chapter 7 we discuss the achieved results and outline the possible future research directions.

Chapter 2

Background: Concepts

In this chapter, we give an overview of the main phenomena concepts and properties appearing in the development of resilient distributed systems. We consider the notion of "resilience" as an evolution of the dependability concept and discuss how goal-oriented development facilitates engineering of resilient systems. In particular, we focus on the dynamic system reconfiguration as the main mechanism for achieving system resilience.

2.1 Resilience Concept

Resilience is a fairly new concept that has been intensively discussed over the last years. Though various interpretations exist, in this thesis we rely on the *dependability-based* definition that was proposed by Laprie [105, 106]:

System resilience is used to designate an ability of the system to persistently deliver its services in a dependable way even when facing changes.

Resilience is an evolution of the dependability concept that focuses on studying the impact of *changes* on system trustworthiness. A *change* is a broad term that may be viewed differently in various domains. The changes can be systematised according to their nature, prospect and timing issues [106]:

- *nature:* functional, environmental or technological;
- *prospect:* foreseen, foreseeable, unforeseen (or drastic) changes;
- *timing:* short term (e.g., seconds to hours), medium term (e.g., hours to months) and long term changes (e.g., months to years).

Resilience extends the dependability concept by emphasising the need to build systems that are flexible and adaptive. It requires implementation of the advanced reconfiguration mechanisms and flexible strategies for efficient

utilisation of the system components to cope with changes and tolerate faults [160].

Since resilience is an evolution of the notion of dependability, majority of its concepts are grounded in the classical definitions proposed for dependability that are discussed next.

2.2 Dependability: Basic Definitions

Dependability is one of the main requirements that we impose on a broad range of computer-based systems. It can be defined as the ability of a system to deliver services that can be justifiably trusted [9, 8]. Dependability is an integrated concept that includes such key attributes as:

- *availability:* the ability of the system to provide a service at any given instant of time;
- *reliability:* the ability of the system to provide a service over a specified interval of time;
- *safety:* the ability of a system to deliver a service under given conditions without catastrophic consequences to the user(s) and environment;
- *integrity:* the absence of improper system alterations;
- *maintainability:* the ability of a system to be restored to a state in which it can deliver correct service;
- *confidentiality:* the absence of unauthorised disclosure of information.

Different *threats* may introduce undesirable deviations in service provisioning and thus jeopardise dependability. Traditionally, threats can be classified into the following categories: *failures*, *errors* and *faults* [9, 8]. Essentially, these terms designate the chain of propagation of a fault to the system boundary as defined below:

- *a failure:* an event that occurs then the delivered service deviates from the desirable (correct) service;
- *a error:* an internal system state that may lead to the subsequent system failure;
- *a fault:* a defect within the system. By their nature, faults can be internal (e.g., a software bug, a memory bit "stuck") or external (e.g., a production defect, a human mistake, an electromagnetic perturbation). In general, a fault might be an origin of an error. However, not all faults produce errors.

Traditionally, engineering dependable systems relies on four main techniques: *fault prevention*, *fault removal*, *fault forecasting* and *fault tolerance* [9, 8].

Fault prevention is a set of techniques aimed at preventing an introduction of faults during the system development process. It comprises, among others, a choice of rigorous development methodologies as well as adopting a suitable standard of quality. *Fault removal* techniques are used to identify and remove errors in the system. The activities of fault removal process include system verification as well as corrective and preventive maintenance of the system. *Fault forecasting* methods are based on prediction and evaluation of the impact of faults on the system behaviour. The evaluation might have both qualitative and quantitative aspects. The qualitative assessment helps to identify and classify failures as well as define combinations of component faults that may lead to a system failure. The quantitative analysis is performed to assess the degree of satisfaction for the required attributes of dependability. Finally, *fault tolerance* techniques are used to develop the system in a such way that it is able to continue its functioning despite the faults.

All these techniques provide the designers with different means to cope with faults. The techniques complement each other and allow the designers to ensure a high degree of system dependability. In this thesis, fault prevention is implemented via formal modelling, fault removal employs theorem proving to verify various system properties, while probabilistic model checking and discrete event simulation are used for fault forecasting. Moreover, we extensively rely on a variety of fault tolerance mechanisms in our development of resilient systems. Since faults constitute the most common class of changes with which a resilient system should cope, next we give an overview of the fault tolerance concepts in more detail.

Fault Tolerance. Fault tolerance techniques aim at ensuring that the system continues to deliver the required service even in the presence of faults. Fault tolerance is usually implemented in two main steps – *error detection* and *system recovery. Error detection* is used to identify the presence of errors. In its turn, *system recovery* aims at eliminating the detected errors (via *error recovery*) and preventing faults from re-activation (via *fault handling*) [71].

Error recovery takes one of the following three forms:

- *backward recovery:* bringing the system back to a previous (correct) state;
- *forward recovery:* moving the system into a new state, from which it can operate (sometimes, in a degraded operational mode);

- *compensation:* putting the system into an error-free state (which relies on the condition that the system has enough redundancy to mask the detected error without service degrading).

In its turn, *fault handling* is a process aimed at preventing faults from being activated again. It can be conducted in three steps. The first step – *fault diagnosis* – determines the causes of errors. The next step is *isolation*. It comprises the actions required to prevent the faulty component(s) from being invoked in the further executions. The last step is *system reconfiguration*. It consists of modifying the (part of the) system structure in such a way that the system continues to provide an acceptable, but possibly degraded, service.

Fault tolerance is achieved by the reliance on redundancy. Different forms of redundancy allow the system either to mask a failure, i.e., nullify its effect at the system level, or to detect a fault and provide (usually temporary) degraded services in the presence of failures. While redundancy enables fault tolerance, it also increases complexity of the system.

Traditionally, the fault tolerance techniques are applied to cope with a number of anticipated situations including failures of software components as well as other abnormal system states. It is desirable to ensure that a system under construction reacts predictably in the presence of such abnormal situations. In this thesis we demonstrate that this task can be greatly facilitated by formal modelling.

Faults can be considered as a simple form of changes, and hence, fault tolerance constitutes an essential mechanism of achieving resilience. Fault tolerance ensures that faults do not prevent the system from delivering its services, i.e., allows it to achieve its *goals*. Since goals provide us with a suitable mechanism for representing the behaviour of a complex resilient system, next we give a detailed overview of this concept.

2.3 Goal-Based Development

Goals are the functional and non-functional objectives of a system [171, 172]. In software engineering, goals have been recognised as useful primitives for capturing system requirements. The reasoning in terms of goals promotes structuring the top-down system design. Goals allow the designers to explore different architectural alternatives. They constitute suitable basics for reasoning about the system behaviour. In particular, resilience can be seen as a property that allows the system to progress towards achieving its goals.

Usually, the system has different types of goals. They are often interdependent. Goals can be structured, e.g., to form a hierarchy. Generally, they can be formulated at different levels of abstraction: high-level goals represent the overall system objectives, while lower-level goals might define

the objectives to be achieved by subsystems or components [171, 172]. Links between goals represent various interdependencies, i.e., the situations where goals affect each other. Traditionally, AND/OR decomposition-abstraction links are introduced to represent the intended goal structure. The process of goal detailisation (i.e., decomposition into subgoals) is performed until a certain level of granularity is reached, i.e., when a subgoal can be assigned to and consecutively realised by the system components – agents [171, 172]. Agent is an active component that performs a task and contributes to goal achievement [172, 171, 170].

The agent concept provides us with a powerful and expressive abstraction for handling complexity of distributed system development. Definition of the term agent varies across the software engineering field [66]. In this thesis, an agent designates a software component that is associated with a certain functionality and is capable to act autonomously in order to meet the design objectives [58, 94]. Correspondingly, *multi-agent systems* are typically decentralised distributed systems composed of agents asynchronously communicating with each other [127]. In this thesis, we consider a decentralised agent system to be a system that operates without a control of central authority.

Typically, agents *interact* with each other in order to achieve their individual or common goals. Interactions might be simple, e.g., information exchange, or complex, e.g., involving requests for service provisioning from one agent to another [94].

Interactions enable agent cooperation. The level and type of cooperation between particular agents is defined by a system organisation. Traditionally, we distinguish between three types of organisations: hierarchical organisation (i.e., one agent may be the manager of the other agents), flat organisation (i.e., agent may work together in a team and communicate with each other directly) and hybrid organisation [94].

Agent *interactions* are achieved by communication. Communication allows the individual agents to share their local information with others agents to facilitate goal achievement. Traditionally, the employed communication mechanism is defined by a certain protocol describing the rules of agent interactions.

The aim of this thesis is to study resilience of multi-agent systems. Therefore, we should explicitly represent off-nominal situations such as agent failures or agent disconnections and assess their impact on the system behaviour. As a result of these off-nominal conditions, agents usually lose an ability to perform their predefined tasks. These might prevent the system from achieving its goals and jeopardise such essential property as safety. The system should recognise such situations and autonomously reconfigure itself to prevent possible harm. Next we give an overview of the aspect of autonomous reconfiguration in detail.

2.4 System Autonomy and Reconfiguration

The concept of system autonomy has been introduced to designate systems that are able to manage themselves [98] without human intervention. Removing humans from the control has been motivated by such reasons as unfeasibility or danger of direct human involvement (due to remote or dangerous environment), a possibility to increase system performance (software usually reacts much quicker than humans) or decrease of system costs [62, 49]. The original concept of system autonomy included such "self" mechanisms as self-configuration, self-repairing, self-healing, self-protection [98, 84]. However, nowadays the autonomic computing paradigm has been broadened and generalised.

System autonomy can be considered from different perspectives, including autonomy of the individual elements forming the system and autonomy of the whole system in general. The autonomic computing paradigm has been widely adopted in various applications and with different degree of autonomy ranging from semi-controlled by humans systems to fully autonomous [62]. Autonomous systems are currently being deployed in many critical applications such as robotics, intelligent monitoring (e.g., healthcare monitoring, traffic jam monitoring), autonomous road vehicles ("driverless cars"), etc.

Typically, the autonomic aspect assumes that a system is capable to monitor its behaviour and dynamically adjust it, if needed. From the resilience perspective, system autonomy can be achieved via dynamic adaptation to various changes and volatile operating conditions. Often adaptation is performed by taking actions that transfer a system from one configuration to another. In general, the adaptation can take a form of parameter adaptation or structural adaptation. The parameter adaptation means changing the measurable system characteristics. The structural adaptation is typically performed via *dynamic reconfiguration*. Essentially, a system *configuration* can be viewed as a specific arrangement of the elements (components) that constitute the system. A configuration is defined by relationships and dependencies between system elements that are established to support achieving system goals. *Dynamic reconfiguration* in its turn assumes that the system is capable to evolve from it current configuration to another one. Dynamic system reconfiguration may imply removal or replacement of configurable elements, which consequently leads to changing of interdependencies between the components. Moreover, reconfiguration may also affect component interactions. The aim of reconfiguration is to ensure that the system remains operational, i.e., capable of achieving its goals and maintaining safe and correct delivery of its services.

In this thesis, we study the reconfiguration aspects in the goal-oriented and service-oriented development paradigms. In particular, the purpose of

reconfiguration is to ensure that the system goals remain achievable. In this thesis, the reconfiguration is based on reallocation of responsibilities between agents either to ensure that the healthy agents can substitute the failed ones (thereby, we ensure handling of negative changes) or to enable more efficient utilisation of agents (hence, we address positive changes). Within service-oriented framework the reconfiguration aims at ensuring that a service can be delivered despite failures of some service-providing components. Reconfiguration here aims at utilising the available service components to re-execute a failed service.

To effectively adapt to changes in the system and its environment, we need also to assess various reconfigurable strategies and architectural alternatives. Indeed, they can guarantee different resilience characteristics, e.g., expressed in the form of performance/reliability ratio. Hence, while developing a resilient distributed system, it is important to consider not only qualitative aspects of system resilience but also its quantitative characteristics. In general, the desirable properties and characteristics to be assessed are identified according to the system goals. In this thesis, we focus on the *design-time* assessment of resilience properties.

Obviously, the design and verification of system resilience is a complex multifacet problem. It requires integrated approaches combining different methods and tools for modelling, verification and quantitative analysis. In the next chapter, we give an overview of the techniques that are used in this thesis to address problems in designing resilient systems.

Chapter 3

Methods and Techniques for Formal Development and Quantitative Assessment

In this chapter, we describe the approaches and tools that we relied on in our modelling, verification and assessment of resilient distributed systems. We also motivate our use of formal techniques as well as the need for integrated approaches.

3.1 Development Methodologies

Development of a resilient system is a challenging engineering task that can be significantly facilitated by the use of formal model-based techniques. It allows the developers to build a system in a *rigorous* way and verify that the system specification meets the requirements. Moreover, formal modelling facilitates systematic derivation of fault tolerance mechanisms and complex reconfiguration solutions.

Traditionally, the methods that have rigorous mathematical basis are called *formal methods*. Formal techniques provide the developers with a strong mathematically-grounded argument about correctness of the system design. The main idea behind the formal modelling and verification is to rely on mathematics and formal logic to avoid imprecision, ambiguity, incompleteness or misunderstanding of system requirements described in natural language [148], and enable formal verification guaranteeing the system under consideration system model adheres to the given specification. Unlike testing, formal techniques allow us to ensure full coverage of possible system behaviours for achieving system resilience.

Traditionally, we distinguish between *proof-based* and *model checking* approaches. The general idea behind the automated proof-based verifica-

tion is following: for given a mathematical or logical statement a computer program (prover) attempts to construct a proof that the statement is true. Typically, *theorem proving* approaches are used to ensure that a model satisfies the desired system properties. Verification is performed without actual model execution or simulation, therefore it allows us to explore the full model state space with respect to the specified properties. Some well-known examples of theorem proving software systems are Isabelle [167, 126], Coq [39], PVS [129, 128], Z3 [45], CVC3 [22, 162, 24], Vampire [140, 141], etc.

In contrast to the proof-based approach, *model checking* is a verification technique that explores all possible system states in a brute-force manner [17, 38]. Specifically, a model checker examines system scenarios in a systematic manner and, thereby, shows whether a given system model satisfies a certain property. Model checking helps us to find violation of the property in specifications by providing counterexamples. There is a big variety of model checkers tools that can be used in verification, e.g., SPIN [80, 81], UPPAAL [26], ProB [168], PRISM [134].

Formal methods are successfully applied in development and verification of complex dependable systems [185]. They are used in such domains as transportation systems [16, 90, 23], space and avionic system [120, 57, 149], traffic management and signaling systems [82, 12], medical devices [30, 95], etc.

Significant advances in integrating formal methods to industrial practice have been achieved in the Deploy project [89]. The project has advanced development of the industrial-strength platform Rodin for state-based modelling and verification of complex resilient systems in the Event-B formalism [3]. This has motivated our choice of Event-B as the formal development framework to be employed in this thesis.

3.2 Event-B Method

In this section, we present our formal development framework – Event-B. The Event-B formalism – a variation of the B Method [2] – is a state-based formal approach that promotes the correct-by-construction development approach and verification by theorem proving [3]. The Event-B framework was influenced by the Action Systems [13, 14, 15] – a formal approach to model distributed, parallel, and reactive systems.

Modelling in Event-B. In Event-B, a system model is specified using the notion of an *abstract state machine* [3]. An abstract state machine encapsulates the model state represented as a collection of model variables, and defines operations on the state. Therefore, *machine* describes the *behaviour* of the modelled system. A machine may also have the accompanying

component, called *context*. A context may include user-defined carrier sets, constants and their properties formulated as model *axioms*. A general form of the Event-B models is given in Figure 4.9.

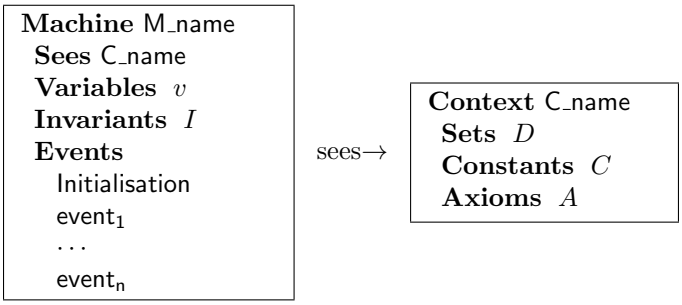


Figure 3.1: Event-B machine and context

An Event-B machine has a name M_name. The model state variables, v, are declared in the **Variables** clause and initialised in the Initialisation event. In Event-B, the model variables are strongly typed by the constraining predicates I called *invariants* given in the **Invariants** clause. The invariants also specify the properties that should be preserved during the system execution. The dynamic system behaviour is defined by the set of atomic *events* specified in the **Events** clause. An event is essentially a *guarded command* that, in the most general form, can be defined as follows:

$$\mathsf{event} \mathrel{\widehat{=}} \textbf{any } vl \textbf{ where } G \textbf{ then } R \textbf{ end}$$

where vl is a list of new local variables, G is the event *guard*, and R is the event *action*.

The guard is a state predicate that defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. The occurrence of events represents the observable behaviour of the system.

In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A non-deterministic assignment is denoted either as $x :\in S$, where S is a set of values, or $x :| P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to S or according to P.

The semantics of Event-B actions is defined using so called *before-after* (BA) predicates [3]. A before-after predicate describes a relationship between the system states before and after execution of an event, as shown in

Table 3.1. Here x and y are disjoint lists (partitions) of the state variables, and x', y' are their values in the after-state.

Table 3.1: Before-after predicates

Action (R)	$BA(R)$
$x := E(x, y)$	$x' = E(x, y) \ \wedge \ y' = y$
$x :\in S$	$\exists z \cdot (z \in S \wedge x' = z) \ \wedge \ y' = y$
$x :\mid P(x, y, x')$	$\exists z \cdot (P(x, z, y) \wedge x' = z) \ \wedge \ y' = y$

Event-B Refinement. Event-B employs a top-down *refinement-based* approach to the system development. Development in Event-B starts from an abstract system specification that models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce non-determinism and introduce detailed design decisions. Refinement usually affects both the context and the machine. Context refinement is a simple extension of the current context achieved by adding new constants, sets and axioms. A machine can be refined in two possible ways either using *data refinement* or *superposition refinement*. In particular, we can replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. In this case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables. Via such a *gluing* invariant – "refinement relation" – we mathematically establish a correspondence between the state spaces of the refined and the abstract machines.

During *superposition refinement*, new implementation details are introduced into the system specification by means of *new* events and *new* variables. These new events can not affect the variables of the abstract specification and only define computations on newly introduced variables.

The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit *skip*. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events *converge*. To prove it, we have to define a *variant* – an expression over a finite subset of natural numbers – and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. In that case, the event obtains the status *anticipated*. This obliges the designer to prove, at some later refinement step, that the event indeed converges.

The correctness and consistency of Event-B models, i.e., verification of the model well-formedness, invariant preservation, deadlock-freeness, cor-

rectness of the refinement steps, is demonstrated by proving the relevant verification theorems – *proof obligations*. Proof obligations are expressed as logical sequences, ensuring that the transformation is performed in a correctness-preserving way [3].

Modelling, refinement and verification of Event-B models is supported by an automated tool – Rodin platform [142]. The platform provides the designers with an integrated modelling environment, supports automatic generation and proving of the necessary proof obligations by means of wide range of automated provers. Moreover, various plug-ins created for Rodin platform allow a modeller to transform models from one representation to another, e.g. from UML to Event-B language [157, 150], or from Event-B specification to programming languages C/C++ [119, 118], ADA [55, 54], etc.

The Event-B refinement process allows us to gradually introduce implementation details, while proving preservation of functional correctness. Such an approach seamlessly weaves verification into the model development and allows us to construct detailed models of complex systems is highly automated incremental manner. By providing an immediate feedback on the correctness of model transformations, it helps us to cope with complexity of the system development. Another important mechanism for handling complexity of formal development is *decomposition*. Model decomposition helps the designers to separate component development from the overall system model but ensure that the components can be recomposed into overall system in a correctness–preserving way [79]. Event-B is equipped with three forms of decomposition: shared-variable [4, 77, 33], shared-event [33] and modularisation [88], all of which are supported by the corresponding Rodin plug-ins [155, 143]. In this thesis we rely on a modularisation extension of Event B [88].

Modularisation. Modularisation extension allows the designers to decompose a system into *modules*. Modules are components containing groups of callable atomic operations [88, 143]. Modules may have their own (external and internal) state and invariant properties. In general, they can be developed separately and then composed with the main system, when needed. Since decomposition is a special kind of refinement, such a model transformation is also a correctness-preserving step that has to be proven by discharging the relevant proof obligations.

A module description consists of two parts – *module interface* and *module body*. Let M be a module. A module interface is a separate Event-B component that has the unique name MI_name. A module interface consists of the external module variables w, the module invariants $MI\_Inv(c, s, w)$, and a collection of module operations, characterised by their pre- and post-conditions defined in the **Operations** clause. In addition, a module inter-

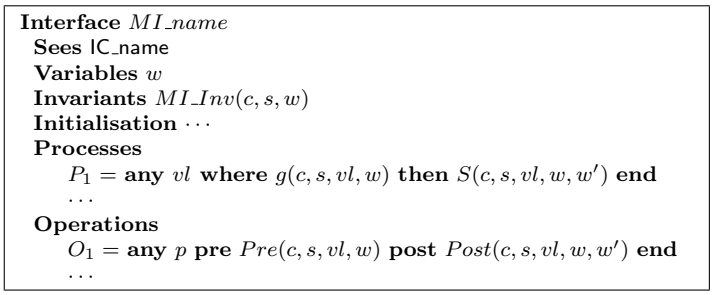


Figure 3.2: Module interface

face may contain a group of standard Event-B events under the **Processes** clause. These events model autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, they describe how the module external variables may change between operation calls. The overall structure of a module interface is shown on Fig.3.2.

A formal development of a module starts with the deciding on its interface. Once an interface is defined, it cannot be changed in any manner during the development. This ensures that a module body may be constructed independently from a system model that relies on the module interface. A *module body* is an Event-B machine. It implements the interface by providing a concrete behaviour for each of the interface operations. To guarantee that each interface operation has a suitable implementation, a set of additional proof obligations are generated. When the module M is imported into another Event-B machine (specified by a special clause **Uses**), the importing machine may invoke the operations of M and read the external variables of M.

We can create several instances of the given module and import them into the same machine. Different instances of a module operate on disjoint state spaces. Identifier prefixes can be supplied in the **Uses** clause of the importing machine to distinguish the variables and the operations of different module instances or those of the importing machine and the imported module. Alternatively, the pre-defined constant set can be supplied as a additional parameter. In the latter case, module instances are created for each element of the given set, thus producing an indexed collection (array) of module instances. A detailed description of indexed modules is given in [86].

The modularisation extension of Event-B facilitates formal development of complex systems by allowing the designers to decompose large specifications into separate components and verify system-level properties at the architectural level. As a result, proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of complex distributed systems.

3.3 Quantitative Assessment

Formal modelling in Event-B allows the designers to derive a complex system architecture, formulate and prove logical system properties and formally verify correctness of the system behaviour. While functional correctness constitutes an important aspect of resilience, we also need to provide the developers with techniques for quantitative resilience assessment. Quantitative assessment plays an important role in the process of resilient system development because it allows the developers to predict the impact of changes on such vital aspects as, e.g., reliability and performance. Moreover, quantitative analysis helps to find suitable trade-offs between these properties as well as evaluate the impact of different architectural alternatives on system resilience. Therefore, in this thesis, we investigate possibility of integration of formal development in Event-B with quantitative resilience assessment. In particular, we study integration with the probabilistic symbolic model checker PRISM [99], and discrete event simulation in SimPy[156].

3.3.1 PRISM model checker

PRISM model checker [99] is one of the leading software tool for formal modelling and verification of systems that exhibit probabilistic behaviour. It provides support for analysis of three types of Markov process – discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC) and Markov decision processes (MDP). Additionally, it supports modelling of (priced) probabilistic timed automata and stochastic games (as a generalisation of MDP) [100]. The state-based modelling language of PRISM relies on the *reactive modules* formalism [7].

A PRISM model consists of a number of *modules* which can interact with each other. The behaviour of each module is described by a set of guarded commands that are quite similar to Event-B events. The latter fact significantly simplifies transformation of Event-B machines to the corresponding PRISM specifications.

While analysing a PRISM model, one can define a number of temporal logic properties to be evaluated by the tool. To assess resilience, we can rely on verifying the time-bounded reachability and reward properties. In the property specification language of PRISM, they can be formulated using the supported temporal logics – PCTL (Probabilistic Computation Tree Logic) [73] for discrete-time models and CSL (Continuous Stochastic Logic) [10, 18] for continuous-time models.

Similarly to Event-B, the PRISM language is a high-level state-based modelling language. Essentially, PRISM supports the use of constants and variables. The variables in PRISM are finite-ranged and strongly typed. They also can be global or local, i.e., associated with a particular *module*.

A PRISM specification is constructed as a parallel composition of modules that can be synchronised using the standard CSP parallel composition. In addition to local variables, each module has a number of guarded commands that determine its dynamic behaviour. Each command consists of a guard and one or more updates over local and global system variables. Each update is annotated with a probabilistic weight (in discrete-time models) or rate (in continuous-time models). Similarly to events in Event-B, a guarded command can be executed (i.e., is enabled) only if its guards evaluate to TRUE. If several guarded commands are enabled at the same time, then the choice between them is defined by the model type – it is non-deterministic for MDP models, probabilistic for DTMC models or modelled as an (exponential) race condition for CTMC models.

PRISM tool has been successfully employed in many domains including distributed coordination algorithms, wireless communication protocols, security as well as dependability and biological models.

To enable probabilistic analysis of Event-B models in PRISM, we rely on the continuous-time probabilistic extension of the Event-B framework [164, 165]. This extension allows us to annotate actions of all model events with real-valued rates and then transform a probabilistically augmented Event-B specification into a continuous-time Markov chain. It also implicitly introduces the notion of time into Event-B models: for any state, the sum of action rates of all enabled in this state events defines a parameter of the exponentially distributed time delay that takes place before some enabled action is triggered.

3.3.2 Discrete-event simulation

Due to similarity between PRISM and Event-B languages, the translation from a Event-B model to a PRISM specification is rather straightforward. It makes the use of PRISM model checker attractive for the performing quantitative assessment. However, the model checking technique does not always scale to large applications. In such case, *simulation* offers a viable alternative for quantitative analysis of resilience.

Traditionally, *simulation* is called the process of imitating how an actual system behaves over time [20]. A simulation generates an artificial system history, thereby enabling analysis of system general behaviour. Simulation is build around the notion of *event* – an occurrence that changes the state of the system. The system state variables are viewed as a collection of all information that is required to define what is happening within the system at a given moment of time.

A widely-used type of simulation is known as discrete-event simulation (DES). In a DES, system state remains constant over an interval of time between two consecutive *events*. Thus events signify occurrences that change

the system state. Events can be classified as either internal or external. *Internal events* occur within the modelled system, while *external events* occur outside the system, but still might affect it. A simulation is run by a mechanism that repeatedly moves simulated time forward to the starting time of the next scheduled event, until there are no more events [152].

From the architectural perspective, a DES system consists of a number of *entities* (e.g., components, processes, agents, etc.), which are either producers or recipients of discrete events. Static entities (e.g., queues, buffers, etc.) can often be represented as *resources*. Resources have limited availability, leading to competition among entities. Waiting for a particular event to occur can lead to a *delay*, lasting for an indefinite amount of time. In other cases, the time estimate may be known in advance. Events can be also *interrupted* and pre-empted, e.g., in reaction to component failures or pre-defined high-priority events.

There are four primary simulation paradigms [20]: process-interaction, event-scheduling, activity scanning, and the three-phase method. In this thesis, we use SimPy [156] – a simulation framework based on process-interaction in Python. Essentially, SimPy is a discrete-event simulation library written in Python. The behaviour of active entities (e.g., customers, requests) is modelled by means of *processes*. All processes settle in an environment and interact with the environment and with each other via events.

Processes are described by simple Python generators. During their lifetime, they create events and yield them in order to wait for them to be triggered. When a process yields an event, the process gets suspended. SimPy resumes the process, when the event occurs. *Timeout* is an important event type. Events of this type are triggered after a certain amount of (simulated) time has passed. SimPy also provides various types of shared resources to model limited capacity congestion points (like servers, queues, buffers, etc.).

Discrete-event simulation represents an attractive technique for quantitative evaluation of different system characteristics. It allows the designers to perform various"what-if" type of analysis that demonstrates sensitivity of the service architecture to changes of its parameters. For instance, it gives an understanding on how the system reacts on peak-loads, how adding new resources affects its performance, what is the relationships between the degree of redundancy and fault tolerance, etc. Moreover, while simulating the behaviour, the designers can also obtain the information on which parameters should be monitored at run-time to optimise a resource allocation strategy. However, to obtain all the above-mentioned benefits, the designers have to ensure that the simulation models are correct and indeed representative of the actual system. In particular, this can be achievable via integration of simulation technique with formal modelling.

Chapter 4

Formal Development and Quantitative Assessment of Resilient Distributed Systems

4.1 Overview of the Proposed Approach

In this chapter, we present the main technical contribution of the thesis – an integrated approach to development and assessment of resilient distributed systems. Our approach relies on formal development by refinement in Event-B, which is augmented with quantitative resilience assessment in the probabilistic model checker PRISM and discrete event simulation in SimPy.

Unreliability of system components and communication channels, complex component interactions as well as highly dynamic operating conditions make the problem of developing resilient distributed systems challenging. To address this problem, we need advanced methods that are able to cope with the complexity inherent to such systems.

The Event-B framework relies on three main mechanisms for coping with complexity: abstraction, decomposition and proofs. Development of a distributed system in Event-B starts from creating an abstract system specification (model). Often such a specification gives a "black-box" model of the system behaviour, i.e., it focuses on defining the externally observable behaviour while abstracting away from the system component architecture and the internal functional behaviour. The initial Event-B model represents a centralised system that exhibits the desired externally observable behaviour and properties. The following refinement steps aim at transforming the abstract model into a detailed system specification by gradually unfolding the

system architecture, precisely defining the functional behaviour as well as deriving a detailed representation of component interactions.

In this chapter, we show how the described above generic approach to development of distributed systems by refinement can be tailored to support resilience-explicit development of different types of systems. The resulting approach shares the common idea of using refinement as the main vehicle for unfolding the system architecture and dynamics. Refinement facilitates systematic introduction of the mechanisms for ensuring system resilience while defining various inter-relationships between the system elements. Moreover, since quantitative assessment of different resilience characteristics is an essential part of the system design for resilience, we show how Event-B models can be augmented with quantitative data and, as a result, serve as a basis for quantitative resilience assessment.

Resilience-explicit development based on functional decomposition. To present our approach, we start first by considering the systems that perform a predefined scenario that can be implemented by a deterministic sequential execution flow. Such kind of the system behaviour is typical for a certain class of systems, which includes, among others, service-oriented and control systems. In service-oriented systems, a service can be often modelled as a sequential composition of subservices. Such a composite service can be provided only if each subservice is successfully executed. A similar type of reasoning can be used for modelling control systems. Since control systems are cyclic, each execution cycle can be represented as a sequential execution of certain functional blocks.

In our approach, we explicitly define the resilience-explicit refinement process for such systems. Specifically, we demonstrate that modelling of not only the nominal system behaviour but also a possibility of system failures already at the abstract level can facilitate a rigorous systematic derivation of the required fault tolerance mechanisms. Then we discuss generic functional decomposition as a refinement step that results in defining a high-level execution flow. We explain how to establish a connection between a global system failure and the corresponding failures in the execution flow. Further, we demonstrate how refinement can be used for deriving the component-based system architecture and linking component failures with those in the system execution flow. Moreover, establishing the connection between the functionality of the system and that of its components allows us to systematically derive the system reconfiguration mechanisms that are based on reallocation of execution of certain functional tasks from the failed components to the healthy ones. Finally, to evaluate the impact of reconfiguration on the system performance and reliability, we augment the resulting Event-B models defining various reconfiguration scenarios with the necessary probabilistic information and demonstrate how to quantitatively assess different reconfiguration strategies.

Modelling component interactions. After presenting the generic resilience-explicit development process for systems with a deterministic sequential execution flow, we focus on detailed analysis of component interactions while providing a certain function (service) or participating in a specific collaboration. To perform the required functions while ensuring fault tolerance, the system components should interact and cooperate with each other. To facilitate reasoning about such cooperative behaviour, we treat components as agents and a resilient distributed system as a multi-agent system. The multi-agent modelling perspective helps us to define the essential properties of cooperative agent activities. As a result, we derive the constraints that should be imposed on agent interactions to ensure correct and safe functioning despite component and communication failures.

Resilient-explicit goal-oriented refinement process. Another large class of distributed systems includes the systems whose execution flow is highly non-deterministic, with a loose connection between functional blocks. Typical examples of such systems are standard multi-agent systems whose components (agents) have some degree of autonomy. For such kind of systems, it is convenient to adopt the goal-oriented reasoning style. In the Section 4.4 of this chapter, we propose a development method for such systems that formalises the resilient-explicit goal-oriented refinement process.

Resilience can be defined as the ability of a system to achieve its objectives – goals – despite failures and other changes. We define a set of specification and refinement patterns that reflect the main concepts of the goal-oriented development. The refinement approach is employed to support the goal decomposition process, thus allowing us to define the system goals at different levels of abstraction. We follow the same generic strategy for development of distributed goal-oriented systems by refinement. Namely, we start by abstractly defining system goals, then perform goal decomposition by refinement, and finally introduce a representation of system agents, whose collaborative activities ensure goal reachability. Therefore, our resilience-explicit goal-oriented refinement approach aims at ensuring goal reachability "by construction". It allows the developers to systematically introduce the required reconfiguration mechanisms to ensure that the system progresses towards achieving its goals despite agent failures (thereby, address "negative" changes) or becomes more performant by using its agents more efficiently (thereby, address "positive" changes).

We consider a dynamic reconfiguration as a powerful technique for achieving system resilience because it allows the system to adapt to changes by modifying its structure, inter-agent relationships and dependencies. However, ensuring correctness of the incorporated reconfiguration mechanisms is a complex task. To address this issue, we formalise the possible interdependencies between goals and agents as well as formulate the conditions

for ensuring goal reachability in a reconfigurable multi-agent system. The proposed formalisation gives a formal systematisation of the introduced concepts and can be seen as generic guidelines for designing reconfigurable systems.

In our resilience-explicit goal-oriented development approach we assume that the agents are sufficiently reliable, i.e., some agents will stay operational during the whole process of goal achieving. To validate such an assumption and derive the constraints on agents reliability, we need to employ quantitative analysis. Quantitative assessment is also required to evaluate the impact of various architectural solutions on the system performance and reliability. Integration with probabilistic model checking in PRISM allows us to achieve these objectives. We augment our Event-B models with quantitative data and transform them into input models for the PRISM model checker. As a result, quantitative assessment allows the designers to make informed design choices and develop systems with predictable resilience characteristics.

Modelling and assessment of resilient architectures Finally, in the last part of this chapter, we investigate how a resilient-explicit refinement approach can be adopted to derive distributed architectures with the incorporated fault tolerance mechanisms. We consider a particular approach to ensure fault tolerance – write-ahead logging (WAL) – and experiment with deriving several alternative architectures implementing it. Each architectural solution exhibit different reliability and performance characteristics. We demonstrate how to derive different architectures by refinement and formally define data integrity and consistency properties that logically formulate reliability characteristics.

Moreover, we propose a graphical notation which facilitates resilience assessment of architectural alternatives by discrete event simulation in SimPy – a library and development framework in Python. The quantitative analysis in SimPy allows us to evaluate the impact of a particular architectural solution on the system reliability/performance ratio.

4.2 Resilience-Explicit Development Based on Functional Decomposition.

In this section, we present our resilience-explicit refinement process for the systems that perform a certain predefined scenario. Our aim is to facilitate rigorous modelling of both nominal and off-nominal system behaviour and to support structured derivation of a functional system specification that integrates the required fault tolerance mechanisms. This is achieved by an explicit representation of the failure behaviour at all levels of abstraction.

Let us assume that the system under construction should provide a service that can be represented as a composition of certain functional blocks as shown in Fig.4.1.

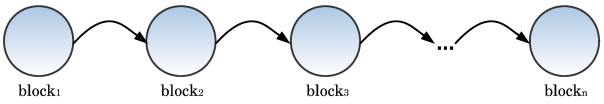


Figure 4.1: Generic execution control flow

In the context of service-oriented systems, the functional blocks correspond to subservices, while in the context of control systems they represent the steps of a single iteration of a control loop. The resulting sequence of functional blocks defines the system execution flow.

In our initial specification, presented in Fig.4.2, we abstractly model the changing status of service execution. Initially the system is idle and can be activated to provide a service, as modelled by the event Activation. This results in changing the value of the boolean variable *idle* from *TRUE* to *FALSE*. Upon service activation, the event Execution becomes enabled. It models the progress of service execution by non-deterministically changing the value of the variable *process*. The set PSTATES = {*FINISHED, UNFINISHED, ABORT*} represents the possible status of the service execution process. The value of *process* remains *UNFINISHED* until all the functional blocks of the service are successfully executed. Upon completion of the service execution, the variable *process* obtains value *FINISHED*, which in turn enables the event Finish. This event changes the status of the system to inactive, i.e., the variable *idle* obtains the value *TRUE*.

The value *ABORT* of the variable *process* designates the occurrence of an unrecoverable failure. The corresponding event Abort deadlocks the specification, i.e., models the fact that the software halts its execution.

Machine abs_behaviour
Variables $idle, process$
Invariants
$idle \in$ BOOL
$process \in$ PSTATES
$idle = TRUE \Rightarrow process = UNFINISHED$
Events
Activation $\widehat{=}$
where $idle = TRUE$
then $idle := FALSE$
end

Execution $\widehat{=}$
where $idle = FALSE \wedge process = UNFINISHED$
then $process :\in$ PSTATES
end
Finish $\widehat{=}$
where $idle = FALSE \wedge process = FINISHED$
then $idle, process = TRUE, UNFINISHED$
end
Abort $\widehat{=}$
when $process = ABORT$
then $skip$
end

Figure 4.2: Abstract System Behaviour Model

Functional Decomposition by Refinement. Next we refine the abstract specification by introducing an explicit representation of the execution flow, i.e., by modelling an execution sequence of the predefined functional blocks. For illustrative purposes, we consider only a simple case of sequential execution. However, in general, we can define any complex scenario (including branching, rollbacking, etc.) by formulating the corresponding axioms in the model context.

For simplicity, we assume that the "id" of each block is defined by its execution order, i.e. $block_1$ is executed first and so on. To explicitly model the impact of failures of individual block executions on the overall service provisioning, we define the following function *block_state*:

$$block\_state \in 1..n \rightarrow \text{BSTATES},$$

where BSTATES = {*NI, OK, POK, NOK*} is an enumerated set of the possible status values for block execution. Initially, none of the block is executed, i.e., the status of each block is *NI*. The block execution can lead to successful completion (*block_state* gets the value *OK*), an unrecoverable failure (*NOK*), or a failure that can be recovered by resigning the block to a different available component (*POK*).

To model functional decomposition, we replace the abstract variable *process* by the variable *block_state*, i.e., perform data refinement. The gluing invariants for such data refinement are given below. An excerpt from the refined specification is shown in Fig. 4.3.

The introduced events Start and Progress model execution of the corresponding functional blocks. They specify the process of sequential selection of one block after another until either all blocks are executed, i.e., the service is completed, or execution of some blocks fails, i.e., service provisioning fails. The sequential order between the events is enforced by the corresponding guards. In particular, the guards ensure that the execution of all previous blocks has been successful completed.

Machine ref1
Variables *idle*, *block_state*
Invariants ...
Events
// first block execution
Start $\widehat{=}$ refines Execution
any *res*
when $res \in \{OK, POK, NOK\} \ \wedge$
$block\_state(1) \neq OK \ \wedge$
$block\_state(1) \neq NOK$
then $block\_state(1) := res$
end

// block execution
Progress $\widehat{=}$ refines Execution
any j, res
where $j > 0 \wedge j < n \wedge$
$res \in \{OK, POK, NOK\} \ \wedge$
$block\_state(j) = OK \ \wedge$
$block\_state(j+1) \neq OK \ \wedge$
$block\_state(j+1) \neq NOK$
then $block\_state(j+1) := res$
end
...

Figure 4.3: Flow Modelling

We formulate and prove the following invariants defining some essential properties of the defined execution flow. The properties postulate that a next block can be chosen for execution only if execution of all the previously chosen blocks was successfully completed and, moreover, the subsequent block was not executed yet:

$$\forall l \cdot l \in 2 \mathinner{..} n \wedge block\_state(l) \neq NI \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} l-1 \Rightarrow block\_state(i) = OK),$$
$$\forall l \cdot l \in 1 \mathinner{..} n-1 \wedge block\_state(l) \neq OK \Rightarrow (\forall i \cdot i \in l+1 \mathinner{..} n \Rightarrow block\_state(i) = NI).$$

The refined model should guarantee that the execution process progresses towards completion of service provisioning. This is ensured by the gluing invariants that establish the relationship between the abstract specification and the functional decomposition introduced by refinement:

$$block\_state[1 \mathinner{..} n] = \{OK\} \Rightarrow process = FINISHED,$$
$$(block\_state[1 \mathinner{..} n] = \{OK, POK, NI\} \vee block\_state[1 \mathinner{..} n] = \{NI\}) \Rightarrow process = UNFINISHED,$$
$$(\exists i \cdot i \in 1 \mathinner{..} n \wedge block\_state(i) = NOK) \Rightarrow process = ABORT.$$

Component Modelling. The purpose of the *Component Modelling* refinement step is to link the functional blocks with the corresponding system components that are responsible for executing them.

We define a variable representing the current state for each system component:

$$comp\_state \in \text{COMPONENTS} \rightarrow \text{CSTATES},$$

where COMPONENTS represents the set of all system components, while CSTATES stands for an enumerated set $\{NA,\ OPERATIONAL,\ FAILED\}$. A component has a status *NA* if it is not currently involved into the execution process. A healthy active component has the status *OPERATIONAL*, while a failed component obtains the status *FAILED*.

To define the relationship between the functional blocks and the components that are responsible for executing them, we introduce the variable *exec*:

$$exec \in \text{COMPONENTS} \leftrightarrow 1..n.$$

Here we do not impose any additional restrictions on the "component-block" interdependency. However, one can specify a certain condition that should hold during the execution, e.g., postulate that a component should be responsible for executing at least one block. We refine the events mod-

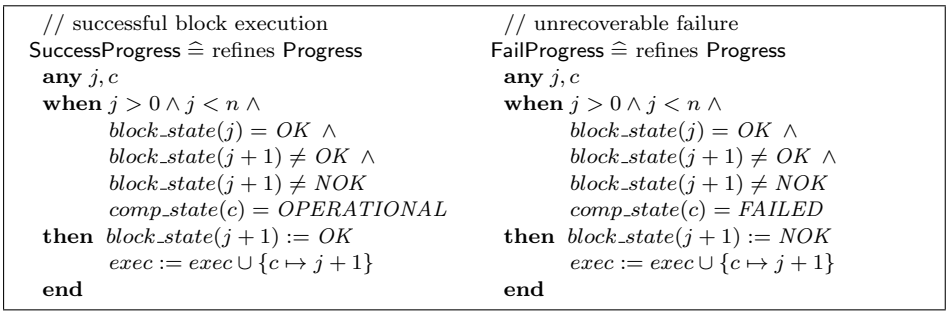


Figure 4.4: Component Modelling

elling three cases of the block execution. Below, Fig. 4.4 presents the events modelling the successful and unrecoverable failed execution of blocks.

To link the status of block execution with the status of the component responsible for executing it, we formulate and prove the invariant, establishing relationship between them:

$$\forall i \cdot i \in 1\,..\,n \wedge block\_state(i) = OK \Rightarrow (\exists c \cdot c \in \text{COMPONENTS} \wedge (c \mapsto i) \in exec \wedge comp\_state(c) = OPERATIONAL).$$

Abstract Reconfiguration Modelling. At this refinement step we introduce an abstract model of reconfiguration that is performed by reassigning responsibility to execute a certain functional block from a failed component to a healthy one. In particular, we define a new function variable *assign* to represent a block assigned to be executed to a component:

$$assign \in \text{COMPONENTS} \leftrightarrow 1..n.$$

We add new events AssignFirstBlock and AssignBlock modelling block assignment (see Fig. 4.5). In the guard of the events SuccessStart, FailStart,

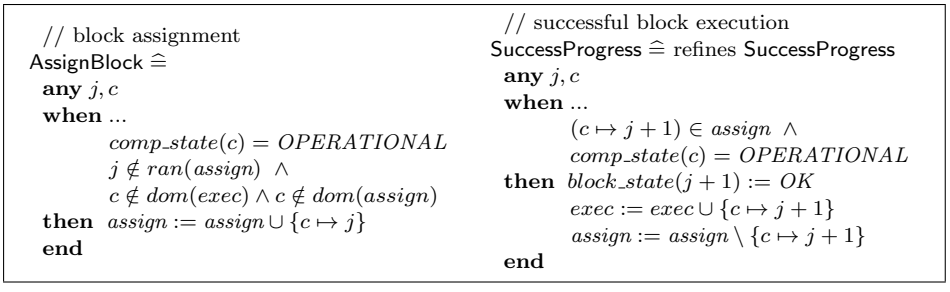


Figure 4.5: Block Reallocation Modelling

SuccessProgress and FailProgress, we add the additional conditions where we check that the corresponding block has been assigned before to the component. In our modelling, we assume that a component may fail only during its block execution.

Let us note that in our specification we model error detection in a highly abstract way. However, we can further refine the model to elaborate on the involved error detection mechanism, for instance, by defining component pinging.

The proposed resilience-explicit refinement process is generic. It abstracts away from the concrete functionality that the system under construction should implement and defines only what kind of the refinement steps should be performed and which types of properties that should be defined and verified. The final refinement step can be seen as a starting point for introducing different reconfiguration strategies and, consequently, employing quantitative assessment technique.

Each reconfiguration alternative (i.e., a different reconfiguration strategy or mechanism) results in creating the corresponding Event-B model. To evaluate the impact of different reconfiguration alternatives, we transform the models into inputs to the PRISM model checker. To achieve this, we augment the corresponding Event-B models with the following probabilistic data:

- the lengths of time delays required by components to execute specific functional blocks;
- the occurrence rates of possible failures of these components.

Moreover, we replace all the local nondeterminism with the (exponential) race conditions. Such a transformation allows us to represent the behaviour of Event-B machines by continuous time Markov chains and use the probabilistic symbolic model checker PRISM to evaluate reliability and performance of the proposed models. Since the quantitative assessment was done primarily by the colleagues of the author of this thesis, the detailed description of this part of approach is omitted in this thesis.

The proposed approach is a generalisation of more detailed formal developments of two case studies presented in Paper I and Paper II of this thesis. In Paper I we undertake modelling and quantitative assessment of an onboard satellite system. We follow the generic resilience-explicit refinement process described above and derive two alternative versions of the reconfigurable system architecture: the triplicated and duplicated ones. From the functional point of view, both architectural alternatives are equivalent, i.e., they implement the same service. To evaluate their resilience aspect, we perform quantitative analysis. The main goal of the analysis is to explore

at design-time whether the duplicated architecture that exploits the dynamic system reconfiguration mechanisms has acceptable performance and reliability characteristics in comparison with the more resource-expensive triplicated alternative.

In the Paper II, to facilitate the formal development process in Event-B, the *event refinement structure* approach is implemented. It augments Event-B refinement with a graphical notation that allows us to explicitly represent the relationships between the events at different abstraction levels as well as define the required event sequence in a model. In this work, we experiment with modelling the system execution flow in the presence of parallelism and block interdependencies.

4.3 Modelling Component Interactions with Multi-Agent Framework

In our resilience-explicit refinement process presented above, we abstracted away from modelling component interactions while performing the predefined functions. Usually, execution of a certain functional block and especially achieving fault tolerance relies on the assumption that the components behave in a cooperative way. For instance, when execution of a functional block is being reallocated from a failed component to a healthy one, the healthy component needs to accept the new responsibility, i.e. behave cooperatively.

The multi-agent modelling paradigm facilitates reasoning about the cooperative component behaviour. We adopt this paradigm to demonstrate how to reason about resilience of complex component interactions. It allows us to treat the components of a resilient distributed system as agents and execution of system functions or services as cooperative agent activities. Next we present our formal approach to resilience-explicit modelling of agent interactions.

We formally reason about agents, their attributes and behaviour as well as agent cooperative activities. The formalisation allow us to establish logical connections between agents and define the conditions under which agents interactions result in correct execution of a cooperative activity. Moreover, the established dynamic connections (called *relationships*) between agents allow us to explicitly reason about resilience of complex agent interactions.

A multi-agent system $\mathcal{MAS}$ *is a tuple* $(\mathcal{A}, \mu, \mathcal{R}, \Sigma, \mathcal{E}, Active, Rel)$*, where* $\mathcal{A}$ *is a set of all the system agents,* μ *is the system middleware,* $\mathcal{R}$ *is a set of all possible relationships between agents in a MAS,* Σ *is the system state space, and* $\mathcal{E}$ *is a collection of system events (reactions). Moreover, the dynamic system attributes Active and Rel map a given system state to a set of the active (healthy) system agents and a set of dynamic relationships*

between the active agents respectively.

The system dynamics is modelled as a set of system events $\mathcal{E}$, where each event $e \in \mathcal{E}$ can be formally represented as a relation on input and output system states, i.e., $e : \Sigma \leftrightarrow \Sigma$. The dynamic system attributes *Active* and *Rel* are then simply functions from Σ, i.e., $Active : \Sigma \to \mathcal{P}(\mathcal{A})$ and $Rel : \Sigma \to \mathcal{P}(\mathcal{R})$, returning respectively the current sets of active system agents and dynamic relationships between them. Intuitively, two or more system agents being in a dynamic relationship means that these agents are currently involved in a specific collaboration needed to provide a predefined system function or service.

Each system agent belongs to a particular agent class. Essentially these classes represent a partitioning of the system agents into different groups according to their capabilities. In general, there can be many agent classes $\mathcal{A}_i,\ i \in 1..n$, such that $\mathcal{A}_i \subseteq \mathcal{A}$. We assume that all of them are disjoint.

The system middleware μ can be considered as a special kind of the system agent that is always present in the system. The main responsibility of the middleware is to ensure communication between different agents, detect appearance of new agents or disappearance (both normal and abnormal) of the existing agents, recover from the situations when the required connections between the agents are lost, etc.

The system state space Σ consists of all possible states of agents and the middleware. The system events $\mathcal{E}$ then include all internal and external system reactions (state transitions). We assume that each agent may have a number of dynamic attributes that can be changed during these transitions. The values of these attributes in a particular state also determine whether a particular agent is currently "eligible", i.e., can be involved in execution of specific system events.

Each *interaction activity* between different agents (or an agent and the middleware) may be composed of a set of events. Moreover, system events may model appearance or disappearance of agents, sending request from one agent to another, recovery of lost connections, etc.

A set $\mathcal{R}$ defines all possible dynamic relationships or connections between agents of the same or different classes. We assume the existence of a number of available data constructor functions to create elements of $\mathcal{R}$. More precisely, for each relationship $r \in \mathcal{R}$, r is modelled as a result of an application of some data constructor function

$$r = R\_Constr_i(a_1, a_2, ..., a_m),$$

where $R\_Constr_i : \mathcal{A}^*_{i_1} \times \mathcal{A}^*_{i_2} ... \times \mathcal{A}^*_{i_m} \rightarrowtail \mathcal{R}$ for some $m \in \mathbb{N}_1$ and each $\mathcal{A}^*_{i_j} = \mathcal{A}_k \cup \{?\}$ for some agent type $\mathcal{A}_k$. Here $\rightarrowtail$ designates an injection function and "?" stands for an unknown agent of the corresponding class.

A relationship can be *pending*, i.e., incomplete. This is indicated by putting the question marks instead of a concrete agent,

e.g., $R\_Constr_i(a_1, a_2, ?, a_4, ?)$. Pending relationships are often caused by disappearance or a failure of the agents previously involved in a relationship. Moreover, an existing active agent may initiate a new pending relationship. Once a pending relationships is resolved (completed), the question mark is replaced by a concrete agent.

While $\mathcal{R}$ represents all possible agent relationships, *Rel* stores the currently active (both complete and pending) relationships. For a relationship to be active, all the involved in it agents should be active as well. In other words, for any $\sigma \in \Sigma$ and $r \in Rel(\sigma)$, if a concrete agent a_i is involved in r, it should be an active one, i.e., $a_i \in Active(\sigma)$.

Let us now consider some expected properties that should be hold for interactions between agents as well as between agents and the middleware.

Property 1. Let $\mathcal{EAA}$ and $\mathcal{EA}\mu$ be all interaction activities (sets of events) defined between agents or between agents and middleware respectively. Moreover, for each agent $a \in \mathcal{A}$, let $\mathcal{E}a$ be a set of events in which the agent a might be involved. Then

$$\forall \sigma, a \cdot\ a \in Active(\sigma) \Rightarrow \mathcal{E}a(\sigma) \in \mathcal{EAA} \cup \mathcal{EA}\mu$$

and

$$\forall \sigma, a \cdot\ a \notin Active(\sigma) \Rightarrow \mathcal{E}a(\sigma) \in \mathcal{EA}\mu.$$

The property restricts agent interactions with respect to the agent activity status. For instance, this property implies that, when an agent is recovering from a failure, it cannot be involved into any cooperative activities with other agents. Therefore, while modelling agent interactions, we have to take into account the agent status.

To represent such a behaviour in Event-B, we define the following events modelling agent activities with the middleware. In particular, the events Appearance and Disappearance model joining and leaving the system by agents (of any classes).

Appearance $\widehat{=}$	Disappearance $\widehat{=}$
any a	**any** a
when $a \in AGENTS \wedge a \notin Active$	**when** $a \in Active$
then $Active := Active \cup \{a\}$	**then** $Active := Active \setminus \{a\}$
end	**end**

Here *AGENTS* defines a set of all system agents (i.e., $\mathcal{A}$), while *Active* represents the subset of active agents.

In a similar way, only active agents can interact with each other as shown by the event Interaction.

Interaction $\widehat{=}$
any $a1, a2$
when $a1 \in Active \wedge a2 \in Active \wedge$
$Elig_1(a1) = TRUE \wedge Elig_2(a2) = TRUE \wedge ...$
then ...
end

Here $Elig_1(a1)$ and $Elig_2(a2)$ abstractly model specific eligibility conditions on the agents that should be checked before their interaction.

The next expected property concerns collaborative activities between the agents and how these activities are linked with the inter-agent relationships.

Property 2. Let $\mathcal{EAA}$ be all the interactions in which active agents may be involved. Moreover, for each active agent a, let $\mathcal{R}_a$ be all the relationships it may be involved in. Finally, for each collaborative activity $CA \in \mathcal{EAA}$, let $\mathcal{A}_{CA}$ be a set of all agents involved in it. Then, for each $CA \in \mathcal{EAA}$ and $a_1, a_2 \in \mathcal{A}_{CA}$,

$$\mathcal{R}_{a_1} \cap \mathcal{R}_{a_2} \neq \varnothing.$$

This property restricts the interactions between the agents: only the agents that are linked by dynamic relationships (some of which may be pending) can be involved into cooperative activities.

To specify abstractly a collaborative activity between agents in Event-B, we define an event CollabActivity. In the event guard, we check that both agents, participating in a collaboration, are active, eligible to be involved, and there is a pre-existing relationships that permits their interactions:

CollabActivity $\widehat{=}$
any $a1, a2$
when
$a1 \in Active \wedge a2 \in Active \wedge$
$Elig_1(a1) = TRUE \wedge Elig_2(a2) = TRUE \wedge$
$RConst_i(a1 \mapsto a2) \in Rel$
then ...
end

Here $RConst_i$ is a data constructor for a specific kind of agent relationships, which is formally specified in the model context. In a similar way, we can model collaborating activities involving any number of agents.

We can specify initiation of a new relationship between agents in two ways. In the case, when all the required agents are active, eligible and ready to enter the relationship, it can be defined by the following event InitiateRelationship.

InitiateRelationship $\widehat{=}$
any $a1, a2$
when $a1 \in Active \wedge a2 \in Active \wedge$
$Ellig(a1) = TRUE \wedge Ellig(a2) = TRUE$
then $Rel := Rel \cup RConst_i(a1 \mapsto a2)$
end

The opposite situation, when some agent of the initiated relationship is still unknown, can be defined by the following event InitiatePendingRelationship. Here the pre-defined element *None*, $None \in AGENTS$, is used to designate a missing agent in the pending relationship (i.e., the special agent "?" in the

above formalisation). In the event shown below, an agent $a1$ initiates a new pending relationship, where the place for a second agent of the particular type is currently vacant (i.e., is marked by *None*). The resulting pending relationships is added to *Rel*.

InitiatePendingRelationship $\widehat{=}$
any $a1$
when $a1 \in Active \ \wedge \ Ellig(a1) = TRUE$
then $Rel := Rel \ \cup \ RConst_i(a1 \mapsto None)$
end

Essentially, all the relationships containing *None* in the place of any their elements denote pending relationships.

To resolve the pending relationship $RConst_i(a1 \mapsto None)$, the corresponding agent has to join this collaborative activity. This situation is abstractly modelled by the event AcceptRelationship.

AcceptRelationship $\widehat{=}$
status *anticipated*
any $a1, a2$
when $a1 \in Active \wedge a2 \in Active \ \wedge \ Ellig(a2) = TRUE \wedge$
$RConst_i(a1 \mapsto None) \in Rel$
then $Rel := (Rel \ \setminus \ RConst_i(a1 \mapsto None)) \ \cup \ RConst\_i(a1 \mapsto a2)$
end

The system middleware μ keeps track of the pending relationships and tries to resolve them by enquiring suitable agents to confirm their willingness to enter into a particular relationship. We can also distinguish a special subset of the pending relationships that have a priority over the others. These relationships are linked with executing critical functions, and hence called critical. A responsibility of the middleware is to detect situations when some of the established or to be established relationships become pending and guarantee eventual resolution of them. Essentially, this means that no pending request is ignored forever and the middleware tries to enforce the given preferences, if possible.

While developing a resilient MAS, we should ensure that all high priority relationships will be established. Therefore, we have to verify that corresponding cooperative activities, establishing these critical relationships, once initiated, are successfully completed. More precisely, we have to verify the following property:

Property 3. *Let $\mathcal{EAA}_{crit}$, where $\mathcal{EAA}_{crit} \subseteq \mathcal{EAA}$, be a subset containing critical collaborative activities. Moreover, let $\mathcal{R}_{pen}$ and $\mathcal{R}_{res}$, where $\mathcal{R}_{pen} \subseteq \mathcal{R}$ and $\mathcal{R}_{res} \subseteq \mathcal{R}$, be the subsets of pending and resolved relationships defined for these activities. Finally, let $\mathcal{R}_{CA}$, where $CA \in \mathcal{EAA}$ and $\mathcal{R}_{CA} \subseteq \mathcal{R}$, be all the relationships the activity CA can affect. Then, for each activity $CA \in \mathcal{EAA}_{crit}$ and relationship $R \in \mathcal{R}_{CA}$,*

$$(R \in \ \mathcal{R}_{pen}) \rightsquigarrow \ (R \in \ \mathcal{R}_{res}),$$

where $\rightsquigarrow$ denotes "leads to" operator.

This property postulates that eventually all the pending relationships should be resolved for each cooperative activity.

To verify this property in Event-B, we have to prove that the event AcceptRelationship converges, i.e., eventually gets enabled. We achieve this by requiring that, at the abstract level, the event AcceptRelationship has the *anticipated* status. This means that "resolving" of a pending relationship is postulated rather than proved. However, at some refinement step, this event status also obliges us to prove that the event or its refinements *converge*, i.e., to prove that the process of resolving a relationship will eventually terminate.

An application of the proposed approach to reasoning about resilient multi-agent systems is illustrated by a case study – development of a hospital MAS. The case study is presented in Papers III-IV included in this thesis.

4.4 Goal-Oriented Modelling of Resilient Systems

In this section, we propose the resilience-explicit refinement process that aims at facilitating development of complex distributed systems whose execution flow is highly non-deterministic, with a loose connection between functional blocks. Typical examples of such systems are multi-agent systems. For such kind of systems, it is convenient to adopt the goal-oriented reasoning style. Goals provide us with a suitable basis for reasoning about system resilience. Indeed, resilience can be considered as an ability of a system to achieve its objectives – goals – despite failures and changes.

4.4.1 Pattern-Based Formal Development of Resilient MAS

To support the goal-oriented development of multi-agent resilient systems in Event-B, we define a set of Event-B *specification* and *refinement patterns* that reflect the main concepts of the goal-oriented engineering. Patterns define generic reusable solutions that facilitate development of complex systems [67, 78, 87].

In the context of formal development in Event-B, patterns represent generic modelling solutions that can be reused in similar developments via instantiation. Usually, an Event-B pattern contains abstract types, constants and variables. The context component of such a model defines the properties that should be satisfied by concrete instantiations of abstract data structures. Moreover, the invariant properties of a pattern, once proven, remain valid for all instantiations.

Let us assume that we have defined a collection of Event-B patterns:

P_1, P_2, ..., P_n that refine each other in the following way:

$$P_1 \textbf{ is refined by } P_2 \textbf{ ,..., } P_{n-1} \textbf{ is refined by } P_n.$$

Such a refinement chain expresses a generic development by refinement. Abstract data structures of all the involved patterns become generic parameters of the development. Each pattern abstractly defines a solution for specifying a certain modelling aspect. The initial pattern P_1 presents a generic model (specification pattern), serving as a starting point of such a development. Each refinement step focuses on formulating specific modelling aspects that should be introduced as a result of the corresponding refinement transformation. The result of such a refinement transformation is called a *refinement pattern*.

Our proposed *specification and refinement patterns* interpret some essential activities of the goal-oriented engineering within the Event-B refinement process:

- *Goal Modelling Pattern*: explicitly defines high-level system goal(s) in Event-B and postulates goal reachability;
- *Goal Decomposition Pattern*: demonstrates how to define the system goals at different levels of abstraction in Event-B (i.e., how to decompose high-level system goal(s) into subgoals). An application of the pattern results in introducing a goal hierarchy;
- *Agent Modelling Pattern*: allows the designers to introduce agents into a specification and associate them with the system goals;
- *Agent Refinement Pattern*: explicitly defines the static and dynamic agent characteristics (attributes).

Next, we describe the proposed patterns in more detail. The full description is presented in Paper V included in this thesis.

Goal Modelling Pattern. We use the concept of a state transition system to reason about the system behaviour. To formulate the *Goal Modelling Pattern*, we start by introducing an abstract type *GSTATE* defining the system state space. Moreover, *Goal* is a non-empty subset of *GSTATE* that abstractly defines the given system goal(s). We say that the system has achieved the desired goals if its current state belongs to *Goal*.

While modelling a system in Event-B, we should ensure that the system under development achieves the desired goal. We can formally express this by requiring that the system terminates in a state belonging to *Goal*. The process of accomplishing such a goal is modelled by the event Reaching_Goal. The event is enabled while the goal is not reached. The variable *gstate* might

eventually change its value from not reached to reached (i.e., $gstate$ becomes $\in Goal$), thus designating achievement of the goal:

Reaching_Goal $\widehat{=}$
  status *anticipated*
  when
    $gstate \in GSTATE \setminus Goal$
  then
    $gstate :\in GSTATE$
  end
end

The system terminates when Reaching_Goal event becomes disabled, i.e., when a state satisfying *Goal* is reached. Note that the event Reaching_Goal has the status *anticipated*. Hence, at this stage reachability is postulated rather than proved, postponing the proof of convergence to some later refinement step. However, later when we introduce more system details, we will be able to prove that the event (or one of its refinements) *converges*.

Goal Decomposition Pattern. The main idea of goal-oriented development is to decompose the high-level system goals into a set of corresponding subgoals. Essentially, the resulting subgoals define intermediate stages of the process of achieving the main goal(s). The objective of the *Goal Decomposition Pattern* is to explicitly introduce such subgoals into the system specification.

While defining the lower-level goals, we should ensure that the high-level goals remain achievable. Hence our refinement pattern should reflect the relation between the high-level goals and their subgoals. Moreover, it should ensure that high-level goal reachability is ensured and can be defined via reachability of the corresponding lower-level subgoals. We assume that the subgoals are independent of each other. This means that reachability of any subgoal does not affect reachability of another one.

To model this pattern in Event-B, we assume (for simplicity, and without losing generality) that the system goal *Goal* is achieved by reaching three subgoals. The subgoals are defined as corresponding variables: $Subgoal1$, $Subgoal2$, and $Subgoal3$. The goal independence assumption allows us to partition the high-level goal state space $GSTATE$ into three non-empty subsets: *SG_STATE1, SG_STATE2* and *SG_STATE3*.

The following mapping function *State_map* establishes the gluing relationship between the new state spaces *SG_STATEi,* $i \in 1..3$, and the abstract state space:

$$State\_map \in SG\_STATE1 \times SG\_STATE2 \times SG\_STATE3 \rightarrowtail\!\!\!\!\!\rightarrow GSTATE.$$

Here $\rightarrowtail\!\!\!\!\!\rightarrow$ designates a bijection function. Essentially it partitions the original goal state space into three independent parts.

To postulate interdependence between reachability of the main goal and that of its subgoals, we rigorous express the following property: the main goal is reached if and only if all three subgoals are reached:

$$\forall sg1, sg2, sg3 \cdot\ sg1 \in Subgoal1 \land sg2 \in Subgoal2 \land sg3 \in Subgoal3 \\ \Leftrightarrow\ State\_map(sg1 \mapsto sg2 \mapsto sg3) \in Goal.$$

In general, we can logically relate the main goal with any expression on its subgoals.

A refinement step performed according to the *Goal Decomposition Pattern* is an example of Event-B data refinement. We replace the abstract variable $gstate$ with the new variables $gstate_i \in SG\_STATEi$, $i \in 1..3$. The new variables model the state of the corresponding subgoals. The following gluing invariant allows us to prove data refinement:

$$gstate = State\_map(gstate1 \mapsto gstate2 \mapsto gstate3).$$

Now the event Reaching_Goal of the abstract machine is decomposed into three similar events $\mathsf{Reaching\_SubGoal_i}, i \in 1..3$, modelling the process of achieving of the corresponding subgoals, as shown below:

Machine M_GD
Reaching_SubGoal$_1$ $\widehat{=}$ refines Reaching_Goal
status *anticipated*
when
$gstate1 \in SG\_STATE1 \setminus Subgoal1$
then
$gstate1 :\in SG\_STATE1$
end
...

The proposed *Goal Decomposition Pattern* can be repeatedly used to refine subgoals into the subgoals of a finer granularity until the desired level of detail is reached.

Agent Modelling Pattern. The proposed *Abstract Goal Modelling* and *Goal Decomposition* patterns allow us to specify the system goal(s) at different levels of abstraction. In multi-agent systems, (sub)goals are usually achieved by system components – agents, which are independent entities that are capable of performing certain tasks. In general, the system might have several types of agents that are distinguished by the type of tasks that they are capable of performing. Our next refinement pattern – *Agent Modelling Pattern* – allows us to model system agents and associate them with goals.

We introduce the set *AGENTS* that abstractly defines the set of system agents. Additionally, we distinguish three non-empty sets *EL_AG1*, *EL_AG2*, and *EL_AG3* of the agents that are capable of achieve the corresponding subgoals.

Agent might fail while trying to achieve a certain subgoal. To reflect this in the specification, we introduce dynamic sets of the eligible agents represented by the variables $elig_i$, $elig_i \subseteq EL\_AGi$, where $i \in 1..3$. We say that an agent is eligible to perform a certain goal if it is active and capable to accomplish it.

Agent failures have direct impact on the process of subgoals achievement, i.e., the goal assigned to the failed agent cannot be reached. To reflect this assumption in our model, we refine the abstract event Reaching_SubGoal$_i$ by two events Successful_Reaching_SubGoal$_i$ and Failed_Reaching_SubGoal$_i$, $i \in 1..3$, which respectively model the successful and unsuccessful reaching of the subgoal by some eligible agent, as shown below:

Machine M_AM
Successful_Reaching_SubGoal$_1$ $\widehat{=}$ refines Reaching_SubGoal$_1$
status *convergent*
any ag
when
$gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge ag \in elig_1$
then
$gstate1 :\in Subgoal1$
end
Failed_Reaching_SubGoal$_1$ $\widehat{=}$ refines Reaching_SubGoal$_1$
status *convergent*
any ag
when
$gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge ag \in elig1 \wedge card(elig1) > 1$
then
$gstate1 :\in SG\_STATE1 \setminus Subgoal1$
$elig1 := elig1 \setminus \{ag\}$
end
...

In the guard of the event Failed_Reaching_SubGoal$_1$, we restrict possible agent failures by postulating that at least one agent associated with the subgoal remains operational: $card(elig1) > 1$. This assumption allows us to change the event status from *anticipated* to *convergent*. In other words, we are now able to prove that, for each subgoal, the process of reaching it eventually terminates. In practice, the constraint to have at least one operational agent associated with our model can be validated by probabilistic modelling of goal reachability, which we discuss later in this chapter.

Agent Refinement Pattern. In the *Agent Modelling Pattern*, we have defined the notion of agent eligibility quite abstractly by formulating the relationship between subgoals and the corresponding agents that are capable of achieving them. Our *Agent Refinement Pattern* aims at elaborating on the notion of agent eligibility. We introduce agent attributes – *agent types* and *agent statuses*, and redefine an eligible agent as an operational agent that belongs to a particular agent type.

We define an enumerated set of agent types AG_TYPE=$\{$ *TYPE1, TYPE2, TYPE3*$\}$ and establish the correspondence between abstract sets of agents and the corresponding agent types by the following axioms:

$$\forall ag \cdot\ ag \in EL\_AGi \ \Leftrightarrow\ atype(ag) = TYPEi,\ i \in 1..3.$$

We consider an agent as capable to perform a certain subgoal if it has the type associated with this subgoal.

To model explicitly the dynamic operational status of each agent, we add a new variable *astatus*:

$$astatus \in AGENTS \rightarrow AG\_STATUS.$$

Here set $AG\_STATUS = \{OK,\ KO\}$, where OK and KO designate operational and failed agents correspondingly.

Now we can data refine the abstract variables $elig_i, i \in 1..3$. The following gluing invariants associate them with the concrete sets:

$$elig_i \ =\ \{a \mid a \in AGENTS \wedge atype(a) = TYPEi \wedge astatus(a) = OK\},$$

for $i \in 1..3$.

In our case, the dynamic set of agents eligible to perform a certain subgoal becomes a set of active agents of the particular type. The event $\mathsf{Failed\_Reaching\_SubGoal}_1$ is now refined to take into account the concrete definition of agent eligibility. The event also updates the status of the failed agent.

$\mathsf{Successful\_Reaching\_SubGoal}_1 \mathrel{\widehat{=}}$ refines $\mathsf{Successful\_Reaching\_SubGoal}_1$
 any ag
 when
  $gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge astatus(ag) = OK \wedge atype(ag) = TYPE1$
 then
  $gstate1 :\in Subgoal_1$
 end
$\mathsf{Failed\_Reaching\_SubGoal}_1 \mathrel{\widehat{=}}$ refines $\mathsf{Failed\_Reaching\_SubGoal}_1$
 any ag
 when
  $gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge astatus(ag) = OK \wedge atype(ag) = TYPE1 \wedge$
  $card(\{a | a \in AGENTS \wedge atype(a) = TYPE1 \wedge astatus(a) = OK\}) > 1$
 then
  $gstate1 :\in SG\_STATE1 \setminus Subgoal_1 \parallel astatus(ag) := KO$
 end

An illustration of our approach based on the proposed patterns is described in the formal development of a multi-robotic system, presented in Paper V and Paper VI of thesis.

As mentioned above, to prove the defined goal reachability property, we had to make the assumptions related to agent reliability, i.e., assume that some agents remain operational to successfully complete the goal achieving

process. To validate this assumption, we can employ quantitative assessment – probabilistic model checking techniques (as described in Paper VII of this thesis). To enable probabilistic analysis of Event-B models in the probabilistic model checker PRISM, we rely on the continuous-time probabilistic extension of the Event-B framework [165]. The idea of this approach is as follows. We annotate actions of all model events with real-valued rates (e.g., failure rate, service rate) and then transform such a probabilistically augmented Event-B specification into a continuous-time Markov chain, which we represent in PRISM. Then we can assess the probability of achieving the goal as well as to compare several alternative system configurations.

The resilience-explicit goal-oriented refinement approach presented above allowed us to identify the key concepts required for formal development of resilient MAS. It has inspired as to propose a conceptual framework for goal-oriented reasoning about resilient MAS that puts a specific emphasis of rigorous definition of system reconfigurability. Next we overview the proposed formalisation.

4.4.2 Formal Goal-Oriented Reasoning About Resilient Reconfigurable MAS

In this section we overview our proposed formalisation of the reconfigurability concept within a multi-agent goal-oriented framework. Our aim is to gradually define the notions of system goals and agents together with their different interrelationships. We systematically introduce the necessary constraints on the system dynamics to facilitate derivation of a necessary reconfiguration mechanism. Here we consider reconfigurability as an ability of agents to redistribute their responsibilities and associations to ensure goal reachability.

To reason about the system behaviour, we rely on the standard concept of a state transition system that we extend with relevant attributes. Next we give a brief overview of the proposed framework. The full description is presented in Paper VIII included in this thesis.

Goal-oriented State Transition System. We start by extending the standard definition of a state transition system (including the set of all system states Σ, the next-state relation *Trans*, and the set of initial system states *Init*) with the notion of goals that a system is trying to accomplish. More specifically, we introduce the set of all possible system goals $\mathcal{G}$ and the function *GMap* mapping a given system goal to a subset of system states:

$$GMap : \mathcal{G} \rightarrow \mathcal{P}(\Sigma).$$

Essentially, the function *GMap* assigns semantics to any goal from $\mathcal{G}$ by associating it with a non-empty set of states (a predicate) of Σ.

Further we extend our goal-oriented state transition system with the notion of subgoals. To introduce inter-relationships between the system goals, e.g., distinguishing particular goals and their subgoals, we define two structures – the relation on goals *G_graph* and the function *SGMap*:

$$G\_graph : \mathcal{G} \leftrightarrow \mathcal{G} \quad and \quad SGMap : \mathcal{G} \rightarrow \mathcal{P}(\Sigma).$$

Essentially, *G_graph* describes relationships between different goals, e.g., how a particular goal can be decomposed into its subgoals and so on. $SGMap(g)$ stands for mapping an arbitrary expression on subgoals of g into a set of states, corresponding to achieving the parent goal g. Intuitively, $SGMap(g)$ stands for the nessessary precondition for achieving goal g.

Essentially, achieving any of subgoals must contribute to reaching the parent goal:

$$\forall g, g' : \mathcal{G}.\ g' \in Subgoals(g) \ \Rightarrow\ SGMap(g) \cap GMap(g') \neq \varnothing,$$

where $Subgoals(g) \quad = \quad \{g' : \mathcal{G} \mid (g \mapsto g') \in G\_graph\}$.

Introducing Agents. Next, we extend a goal-oriented state transition system by introducing agents that can carry out tasks required for achieving the system goals. We introduce the type (set) $\mathcal{A}$ for all possible system agents and define the function *Active* to distinguish a subset of active agents in the current system state:

$$Active : \Sigma \rightarrow \mathcal{P}(\mathcal{A}).$$

By "active" agents we mean the agents that can carry out the tasks in order to achieve the system goals. In its turn, the inactive agents are either the agents which are not currently present in the system or those which failed and thus incapable to carry out any tasks.

To reflect the heterogeneous nature of multi-agent systems, next we introduce possible agent attributes. Namely, we associate certain classes of agents with specific types of system goals they are able to accomplish. To formalise it, we first introduce classifications of system agents and goals and then define relationships between the introduced classes.

The following functions

$$atype : \mathcal{A} \rightarrow AType \quad and \quad gtype : \mathcal{G} \rightarrow GType,$$

associate each agent and goal with their respective type, where *AType* and *GType* are abstract types containing all possible agent and goal types respectively.

The separate goals of the same goal type can be achieved independently, i.e., can be assigned to different agents that work in parallel to accomplish them:

$$\forall g_1, g_2 : \mathcal{G},\ gt : GType.\ gtype(g_1) = gt\ \wedge\ gtype(g_2) = gt\ \wedge\ g_1 \neq g_2 \Rightarrow GMap(g_1) \cap GMap(g_2) = \varnothing,$$

To represent interrelationships between different agent and goal types, we introduce the relation AG_Rel:

$$AG\_Rel : AType \leftrightarrow GType.$$

This formalises a connection between the corresponding agent and goal types clarifies which agents can be given the tasks related to specific system goals.

Agent Subordination and Supervision. Defining agent types and hierarchy of goal types allows us to introduce a subordination structure between agent types. Essentially, subordination means that one agent may be the "master" (manager) of the other agent(s). Naturally, agent subordination supposes that some agents "supervise" activities of other agents. Moreover, a supervising agent can give concrete goal assignments to subordinate agents, which, in turn, should "report" to its supervisors once the assigned goal has been accomplished. The unreached system goals can be also dynamically partitioned among the supervisor agents, essentially modelling accepted responsibilities of those agents for supervision over some goals.

To introduce such subordination, we define a relation on agent types, called A_Sub:

$$A\_Sub : AType \leftrightarrow AType.$$

Moreover, for each pair of subordinated agent types, there should exist (at least one) pair of the related goal types such that goals of the parent goal type can be handled by agents of the "master" agent type, while goals of the subgoal type can be handled by agents of the subordinate agent type.

Let us note that the introduced notions of agent types, subordination, ability to accomplish or supervise a particular goal, constitute *static* properties of a multi-agent goal-oriented system. On the other hand, since agents can change their active/inactive status during system execution, the function *Active* expresses a dynamic system characteristic. To formally define a system configuration and the corresponding reconfiguration mechanism for tolerating system changes, we need to define additional *dynamic* system characteristics. First, in a specific dynamic system state, a particular agent can be *attached* to another agent, which serves as its supervisor. A specific goal that has not been yet reached can be put under *responsibility* of a particular supervisor agent. Moreover, a specific goal can be *assigned* by

a supervisor to one of its subordinate agents. Later, the assigned goal can be executed by the corresponding agent. If the agent fails to achieve the assigned task, its goal can be reassigned to another agent capable to achieve it.

We formulate these dynamic notions formally. For instance, *agent attachment* is defined as the function *Attached*, such that

1. $Attached : \Sigma \to \mathcal{P}(\mathcal{A} \times \mathcal{A})$,

2. $\forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ (a_1 \mapsto a_2) \in Attached(\sigma)\ \Rightarrow$
$a_1 \in Active(\sigma) \wedge a_2 \in Active(\sigma) \wedge atype(a_1) \mapsto atype(a_2) \in A\_Sub \wedge$
$\neg(\exists a_3 : \mathcal{A}.\ a_3 \neq a_1\ \wedge\ (a_3 \mapsto a_2)\ \in\ Attached(\sigma))$.

Therefore, for any agents a_1, a_2 and system state σ, the expression $(a_1 \mapsto a_2) \in Attached(\sigma)$ implies that (i) both agents are active in σ, (ii) the agent type of a_2 is subordinate to that of a_1, and (iii) the agent a_2 is not currently attached to any other supervisor agent.

Moreover, a goal-oriented multi-agents system supports agent attachment if, at any point where the conditions for agent attachment are satisfied, the system has an opportunity (but not an obligation) to do such an action.

Similarly to *agent attachment*, we define *goal responsibility* (the corresponding function called *Responsible*) and *goal assignment* (the corresponding function called *Assigned*) as system dynamic attributes (i.e., they depend on the current system state). Goal responsibility specifies the relationships between certain goals and the agents currently supervising them. In its turn, *goal assignment* defines the relationships between the goals and pair of agents that supervise and perform these goals respectively. Moreover, a goal-oriented multi-agents system supports goal responsibility and goal assignment if, at any point where the conditions for these properties are satisfied, the system is able to do these actions.

Now, the introduced above notions and characteristics allow us to define notion of a reconfigurable system and reason about system reconfigurability.

Reasoning about System Reconfiguration Towards Goal Achievement. Based on our definitions, we can explicitly define multi-agent systems that support system dynamic reconfiguration. Specifically, these are the systems that allow redistributing (unassigned) goals to different responsible agents or reattaching (unassigned) agents to different supervisor agents. Moreover, the following properties must hold

$$\begin{aligned}
&\forall \sigma : \Sigma,\ g : \mathcal{G},\ : a_1, a_2 : \mathcal{A}.\ (g \mapsto a_1) \in Responsible(\sigma)\ \wedge \\
&\quad gtype(g) \in AS_goals(atype(a_2))\ \wedge \\
&\quad \neg(\exists a_3 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_3)) \in Assigned(\sigma))\ \Rightarrow \\
&\qquad \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in Trans\ \wedge\ (g \mapsto a_2) \in Responsible(\sigma')
\end{aligned}$$

and

$$\begin{aligned}\forall \sigma : \Sigma,\ a_1, a_2, a_3 : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge \\ (\mathit{atype}(a_3) \mapsto \mathit{atype}(a_2)) \in \mathit{A\_Sub}\ \wedge \\ \neg(\exists g : \mathcal{G}.\ (g \mapsto a_1 \mapsto a_2)) \in \mathit{Assigned}(\sigma))\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_3 \mapsto a_2) \in \mathit{Attached}(\sigma').\end{aligned}$$

Essentially, these two properties require the existence of state transitions allowing to redistribute goal responsibility and agent attachment. Here the condition $\mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a_2))$ checks that the type of the agent a_2 allows the agent to supervise the goal g, while the condition $(\mathit{atype}(a_3) \mapsto \mathit{atype}(a_2)) \in \mathit{A\_Sub}$ requires that the agent type of a_2 is subordinate to that of a_3.

Finally, we formulate and prove a theorem about goal reachability in a reconfigurable multi-agent system (for proof, see Paper VIII).

Theorem: Goal reachability in a reconfigurable agent system. For a *reconfigurable* goal-oriented multi-agent system $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$, the following property is true:

$$\begin{aligned}\forall \sigma : \Sigma,\ g : \mathcal{G}.\ \sigma \in \mathbf{dom}(\mathit{Trans})\ \wedge\ \sigma \notin \mathit{GMap}(g)\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}^+\ \wedge \sigma' \in \mathit{GMap}(g).\end{aligned}$$

Essentially, theorem states, that for a reconfigurable goal-oriented multi-agent system, any goal that is not yet reached at any (non-final) system state is reachable. Let us note that the theorem is proved to formally demonstrate that all the introduced notions and mechanisms are sufficient to ensure goal reachability in such a system.

The goal-oriented framework provides us with a suitable basis for reasoning about reconfigurability. It allows us to define reconfiguration as an ability of agents to redistribute their responsibilities to ensure goal reachability. The proposed formal systematisation of the involved concepts can be seen as generic guidelines for formal development of reconfigurable systems. In particular, in Paper VIII we show such guidelines can be interpreted within the Event-B framework.

4.5 Modelling and Assessment of Resilient Architectures

In this section, we focus on the problem of formal modelling and quantitative assessment of resilient architectures. In particular, we experiment with different architectural alternatives implementing a well-known fault tolerant

mechanism for distributed systems (WAL). Each alternative provide the developers with different reliability guarantees expressed in the terms of data consistency and data integrity properties. However, since higher reliability usually results in lower performance, it is desirable to quantitatively assess this ratio under different configurations parameters and loads.

We start this section by briefly describing the WAL mechanism. Then we demonstrate how to use the employed refinement approach to derive resilient architectures. Finally, we propose a graphical notation facilitating construction and validation of models for resilience assessment in SimPy – a library and development framework in Python.

WAL mechanism and data base replication. In Paper IX we focus on defining and verifying the data consistency and data integrity properties under possible failure scenarios using the write-ahead logging (WAL) mechanism [121, 133] combined with replication techniques [97, 158, 96, 183]. The WAL mechanism is a standard data base technique for ensuring data integrity. The main principle of WAL is to apply the requested changes to data files only after they have been logged, i.e., recorded into a log file and the file has been stored in a persistent storage. If the system crashes, it can be recovered using the log file. Therefore, the WAL method ensures fault tolerance. Moreover, the WAL mechanism helps to optimise the system performance, since only the log file (rather than all the data changes) should be written to the persistent storage to guarantee that a transaction is (eventually) committed.

However, an implementation of a persistent storage, i.e., the guaranteeing that the node containing the log file never crashes, is hard to achieve. To ensure resilience, the proposed mechanism can be combined with the required replication techniques as follows. In a distributed data store consisting of a number of nodes distributed across different physical locations, one of the nodes, called *master*, is appointed to serve incoming data requests from distributed data store clients and transmit back the outcome of the request, i.e., acknowledge success or failure of a transaction. The remaining nodes, called *standby* or *worker nodes*, contain replicas of the stored data.

Each request is first recorded in the *master log* and then applied to the stored data. After this, an acknowledgement is sent to the client. The standby nodes are constantly monitoring and streaming the master log records into their own logs, before applying them to their persistent data in the same way. Essentially, the standby nodes are continually trying to "catch up" with the master. If the master crashes, one of the standby nodes is appointed to be the master in its stead. At this point, the appointed standby effectively becomes the new master and starts serving all data requests. A graphical representation of the system architecture is shown in Fig 4.6.

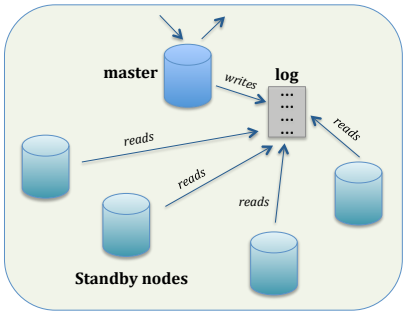


Figure 4.6: Distributed Data Base System Architecture

A distributed data store can implement different models of logging. In the *asynchronous model*, the client request is acknowledged after the master node has performed the required modifications in its persistent storage. The second option – the cascade master-standby – is a *semi-synchronous architecture*. The client receives a response after both the master and its warm standby (called upper standby) has performed the necessary operations. Finally, in the *synchronous* model, only after all replica nodes have written into their persistent storage, i.e., fully synchronised with the master node, the transaction can be committed. Obviously, such logging models deliver different resilience guarantees in the cases of component crashes.

Modelling Distributed Data Store in Event-B. In this thesis, we propose a refinement based approach to deriving various system architectures. For each architecture we formulate and prove system-level *logical* properties – *data consistency* and *data integrity*. Within the described system, the *data consistency* properties express the relationships between the requests handled by the master and those handled by the standby nodes. Since any standby node is continuously copying the master log, we can say that any standby node is logically "behind" the master node. The degree of consistency depends on the chosen architecture.

Within the described system, the *data integrity* property ensures that the corresponding log elements of any two storages (master or standby replicas) are always the same. In other words, all logs are *consistent* with respect to the log records of the master node. Essentially it means, that different replicas all do the same operations according to the log records.

We rely on the Event-B refinement technique to gradually unfold the system architecture and functionality. This allows us to represent the system components, model their change (both normal and abnormal) as well as introduce a generic mechanism for changing the master node. We also mathematically formulate the data consistency and data integrity properties

for different architectural models. Additionally, formal modelling allows us to identify situations, where the desired properties can be violated.

Below, the refinement process is illustrated for the *asynchronous system* architecture. It consists of the abstract model and two refinements as depicted in Fig. 4.7. A brief outline of each step is given as follows:

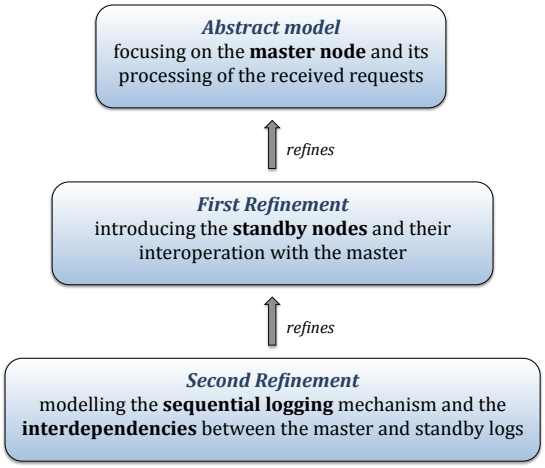


Figure 4.7: Overview of the Development Strategy: Asynchronous model

Initial model. It abstractly represents the overall system architecture. In particular, its describes the behaviour of the master node, which is responsible for receiving and processing of incoming requests. Moreover, here we model a possible change in the set of active nodes and introduce an abstract representation of the procedure of a new master selection.

First refinement. This is a refinement of the abstract specification. Here we introduce the behaviour of the standby nodes and their interactions with the master. We model how the received data requests are transferred through the different processing stages on the master and standby sides. Moreover, we explicitly model possible node failures, and therefore elaborate on the procedure of selection of new master. At this step we are able to formulate the data consistency properties expressing the relationships between the requests handled by the master and those by the standby nodes, respectively. A short transitional period may be needed for the new master to "catch up" with some of the standby nodes that got ahead by handling the requests still not committed by the new master. To address this problem, we introduce an explicit representation of the transition period and redefine the consistency property.

Second refinement. This model explicitly introduces the sequential logging mechanism and the resulting interdependencies between the master and standby logs. The model is obtained as a result of a data refinement. An introduction of the sequential representation of the component log allows us to refine some proven invariants as well as prove some new ones. In particular, we formulate and prove the log data integrity properties as the following model invariants:

$$\forall c1, c2, i \cdot c1 \in comp \wedge c2 \in comp \wedge i \in 1 \mathinner{..} index\_written(c1) \wedge \\ i \in 1 \mathinner{..} index\_written(c2) \Rightarrow log(c1)(i) = log(c2)(i).$$

The property states that the corresponding log elements of any two storages are always the same.

The formal development of the *semi-synchronous* and *synchronous* architectures is essentially repeats the refinement steps presented for the *asynchronous* model. However, in the *semi-synchronous* case, in the abstract model we also introduce the upper standby component and its interoperations with the master node. In both cases, we implement specific architectural solutions for the corresponding architecture and respective restrictions on the component behaviours. Therefore, the data consistency properties for each architecture are reformulated and proved. In its turn, the data integrity property is architectural-independent and remains the same. The resulting Event-B formal models can be served as a starting point for future development of a specific distributed application.

The outlined refinement process supports qualitative reasoning about resilience. However, it is also desirable to quantitatively assess sensitivity of the architecture to changes of its configuration parameters. To enable such quantitative assessment of resilience characteristics, in particular, to analyse the performance/fault tolerance ratio of the architectural alternatives, we integrate formal modelling in Event-B with discrete-event simulation in SimPy. Next we overview the proposed integrated approach.

Quantitative Assessment of Resilient Characteristics. To facilitate integration of the described formal modelling with discrete-event simulation, we introduce an intermediate graphical notation called a *process-oriented model*. The proposed notation contains only the main concepts of the domain together with the key artefacts required for both formal modelling and simulation. It relies on the following assumptions:

- A system consists of a number of parallel processes, interacting asynchronously by means of discrete events;
- System processes can be grouped together into a number of components;

- Within a process, execution follows the pre-defined scenario expressed in terms of functional blocks (activities) and transitions between them. Each such functional block is typically associated with particular incoming events the process reacts to and/or outgoing events it produces;

- A system component can fail and (in some cases) recover. The component failures and recovery mechanisms are described as special component processes simulating different types of failures and recovery procedures of the component;

- Some events (e.g., component failures) should be reacted on immediately upon their occurrence, thus interrupting the process current activities. Such special events (*interrupts*) are explicitly described in the component description.

An example of such a component is graphically presented on Fig.4.8. The component interface consists of one incoming event (*arrival_evn*) and two outgoing events (*rejection_evn* and *completion_evn*). The component itself contains two processes describing its "nominal" behaviour: the first one stores requests to perform a certain service, and the second one performs a requested service and returns the produced results. The internal event *perform_evn* triggers the request execution by the second process. In addition, the component includes the processes Failure and Recovery to simulate possible component failures and its recovery.

Integration Formal Modelling with Simulation in SimPy. A process-oriented model serves as a basis for both Event-B development and system simulation in SimPy. Translating a process-oriented model into Event-B gives us the starting point for formal development with the already fixed system architecture and the control flow between main system components. The corresponding system properties are explicitly formulated and proved as system invariants.

While translating a process-oriented model to SimPy, we augment the resulting code with concrete values for its basic quantitative characteristics,

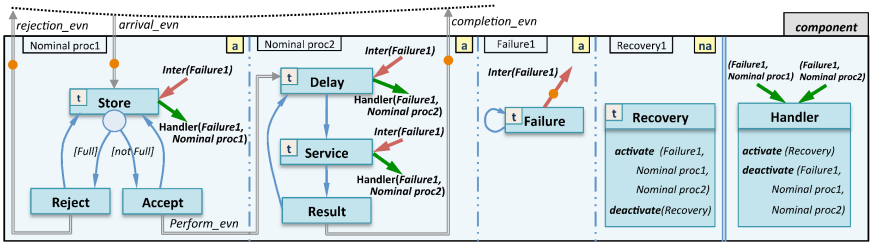


Figure 4.8: Example of a system component

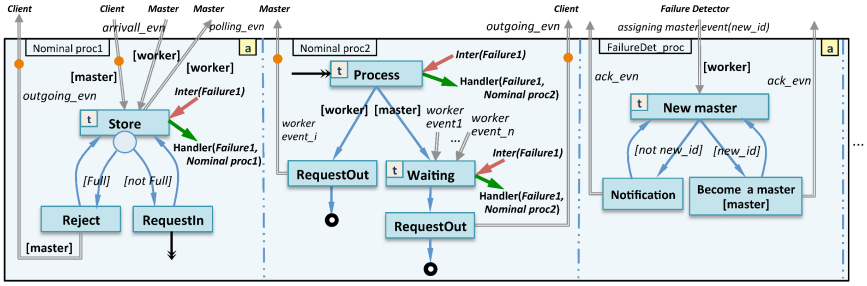


Figure 4.9: Synchronous model

such as data arrival, service, and failure rates. This allows us to compare the system performance and reliability for different system parameter configurations. If satisfactory configuration values can be found and thus re-design of the base process-oriented model is not needed, the simulation results does not affect the Event-B formal development and can be considered completely complementary to it.

We apply the proposed approach to evaluate architectural alternatives combining WAL and replication. We consider two different system architectures: asynchronous and synchronous models. The resulting process-oriented models for the node components of synchronous architecture is presented in Fig. 4.9.

The graphical notation facilitates development of SimPy code. Discrete event simulation in SimPy allow us to evaluate how different parameters affect the results within the considered architecture.

Fig.4.10 and Fig.4.11 show the results of a simulation involving two models – asynchronous and synchronous. With identical operating conditions and parameters, the *asynchronous* model has higher throughput, completing 99.3 % of requests in 1 hour. This is expected, because the *asynchronous* model has shorter delay in comming transactions than the *synchronous* one, which completes 97.2 % of requests in 1 hour (see Table 4.1).

Table 4.1: Results from model comparison

	Completed (%)	Rejected (%)	Failed (%)
asynchronous	99.3	0	0.2
synchronous	97.2	1.6	0.7

Moreover, for each architecture, we can perform sensitivity analysis. Specifically, we can evaluate the impact of the buffer capacity and the mean failure rate on the throughput of the system. Further experiments can reveal more information about the system. For example, we can evaluate how changing the number of standby node affects the performance of the

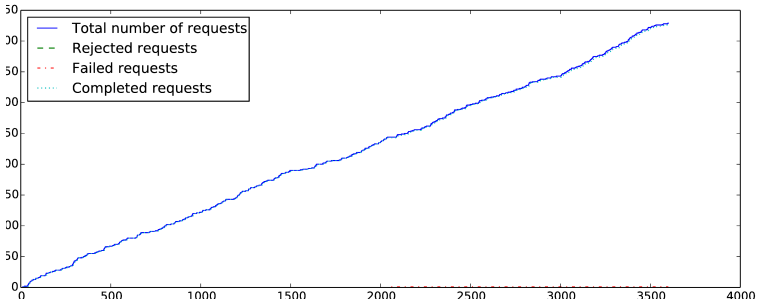


Figure 4.10: Asynchronous model. Mean arrival rate is 7.5/min, service time is 5s, buffer capacity is 5 and mean failure rate is 1.8/h.

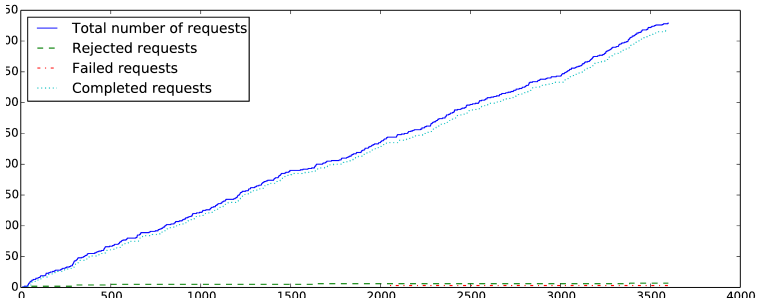


Figure 4.11: Synchronous model. Mean arrival rate is 7.5/min, service time is 5s, buffer capacity is 5 and mean failure rate is 1.8/h.

models and the mean failure rates. In general, the desirable properties and characteristics to be assessed are identified according to the system goals.

To summarise the results of this section, we can conclude that our pragmatic approach to integrating formal modelling in Event-B and discrete-event simulation in SimPy offers a scalable solution to integrated engineering of resilient architectures. Modelling in the Event-B framework allows us to reason about correctness and data integrity properties of the corresponding architectures, while discrete-event simulation in SimPy enables quantitative assessment of performance and reliability. The full description of the proposed approach is presented in Paper IX and Paper X of this thesis.

Chapter 5

Overview of the Original Publications

This thesis is built on the results of ten scientific publications. This chapter gives a short summary of them.

Paper I: Formal Development and Assessment of a Reconfigurable On-board Satellite System

In this paper, we discuss a formal approach to modelling and assessment of a reconfigurable fault tolerant satellite system. Fault tolerance of satellite system is vital for achieving goals of the space mission. Since the use of redundancy is restricted by the size and the weight of the on-board equipment, dynamic system reconfiguration should be employed to achieve the desired level of reliability.

In this work, we define the design rules for stepwise development of a dynamically reconfigurable system in Event-B. Furthermore, we demonstrate how to formally assess the chosen reconfiguration strategy as well as evaluate whether the incorporated fault tolerance mechanism fulfils the reliability and performance objectives. The proposed method is illustrated by a case study – development and assessment of the reconfigurable satellite Data Processing Unit (DPU). We derive a complex system reconfigurable architecture by refinement and formally verify correctness of the established relationships between component failures and goal reachability. Further, we enrich the resulting Event-B specification with explicit probabilistic information about the reliability of its components and the duration of time required to execute the tasks assigned to these components. Then, using the probabilistic model checker PRISM, we compute and compare the reliability and performance measures of a system by employing two different fault tolerance mechanisms: the one that realises a standard redundancy scheme and the other one that is based on dynamic reconfiguration. The proposed approach guarantees

correct design of complex fault tolerance mechanisms as well as facilitates finding suitable trade-offs between system reliability and performance.

Paper II: Formal Derivation of Distributed MapReduce

In this work, we undertake a formal study of the MapReduce framework. MapReduce is a powerful distributed data processing model that is currently adopted in a wide range of domains to efficiently handle large volumes of data. The main principle of MapReduce is to parallelise processing of data by first mapping them to multiple processing nodes (the map stage) and then merging the obtained results (the reduce stage). The goal of this paper is to investigate the applicability of the Event-B framework and its refinement technique for modelling MapReduce.

To facilitate development of the MapReduce computations, we formally model the control flow and mathematically define possible data interdependencies between the map and reduce stages of MapReduce. This formalisation allows us to propose an alternative architectural solution that weakens blocking between the stages and, as a result, achieves a higher degree of parallelisation of the MapReduce computations. Next, we demonstrate that, by relying on the proposed formalisation, we can derive two models of MapReduce – blocking and partially blocking models. The proposed approach relies on stepwise refinement in Event-B and, in particular, the event refinement structure approach – a diagrammatic notation facilitating formal development.

The proposed formalisation of the MapReduce framework is generic, i.e., it can be instantiated by different values for specific configurations of MapReduce. The static part of the modelled system is formally defined in the corresponding context component. The definitions of static data structures in the context are abstract, i.e. they state only the essential properties to be satisfied. This makes them generic parameters of the whole formal development. In its turn, such formal development becomes generic, representing a family of the systems that can be described by providing suitable concrete values for the generic parameters. The proposed formal model can be used then as a starting point for future development of a specific MapReduce application. The actual concrete values can be supplied by either the end-user or the developer of the MapReduce framework.

Paper III: Formal Development of Critical Multi-Agent Systems: A Refinement Approach.

This paper presents an approach for modelling and verification of critical multi-agent systems. In this work, we focus on studying agent interactions and activities whose incorrect execution may jeopardise safety of multi-agent

applications. We demonstrate that the Event-B formal development based on refinement allows the designers to rigorously specify complex agent interactions and verify their correctness and safety properties.

In the first part of the paper, we define general principles of formal reasoning about MAS. We define notions and properties that allow us to represent agent classes and attributes as well as agent activities and their interactions. In particular, the proposed properties determine the rules for regulating interactions between the agents and define properties of collaborative agent activities. Next, we demonstrate by a case study how our formalisation can facilitate a rigorous development of a complex hospital system. Specifically, we investigate safety of the *doctor-patient* agent interactions. In our development, we explicitly introduce a fault tolerance mechanism that guarantees that the system functions correctly even in the presence of doctor and patient failures or their disconnections. Specifically, we verify by proofs correctness and safety of two agent activities – handling patient emergencies and consistent updates of patient data records. The formal verification process based on Event B refinement allows us to systematically capture complex requirements. Moreover, the proposed refinement approach facilitates derivation of the constrains that should be imposed on the system to guarantee its safety.

Paper IV: A Refinement-Based Approach to Developing Critical Multi-Agent Systems

Paper IV can be considered as a revised and expanded version of the Paper III. Here, we further demonstrate the benefits of a refinement-based approach to modelling and verification of critical multi-agent systems. We further expand the obtained formal development of the considered hospital system and derive a distributed architecture by a special kind of refinement – decomposition.

Following the system approach [75], we start our development from an abstract centralised model and then gradually unfold the system functionality by augmenting the specification with detailed design decisions. At a certain refinement step, when the required system functionality is introduced and required safety properties are proved, we explicitly define the communication between agents and then perform the system decomposition as a special form of refinement. We distribute agent activities between different modules which correspond to agent classes (i.e., patients and doctors). By decomposing the system-level model, we derive the required interfaces of the resulting system components and guarantee that their activities do not violate the safety requirements imposed on the system. As a result of the decomposition step, we arrive at a specification of a distributed hospital MAS.

We argue that a formal approach based on refinement in Event-B advocated in this paper provides us with a suitable and scalable method for development of complex multi-agent systems. Firstly, the reliance on the proof-based verification does not impose any restrictions on the size of the model, number of agents, etc. Secondly, the proposed method gives full benefits of the systems approach and allows us to reason about such system-level properties as safety.

Finally, the adopted top-down development technique allows us to efficiently cope both with the complexity of requirements as well as the complexity of modelling and verification processes by adopting an incremental development approach. Thereby, we conclude that the refinement in Event-B constitutes a suitable scalable technique for formal development of critical MAS.

Paper V: Formal Goal-Oriented Development of Resilient MAS in Event-B

In this work, we investigate the use of the goal-oriented requirement analysis with combination of a refinement-based approach to develop a resilient system. Specifically, we reason about system resilience in terms of goals. To ensure system resilience, we should guarantee that the system achieves its goals despite system changes. This work demonstrates how to rigorously define goals in Event-B and ensure goal reachability "by construction" using refinement.

We formalise goal-oriented development by defining a set of modelling and refinement patterns that interpret essential activities of the goal-oriented engineering in terms of the Event-B refinement. Specifically, we demonstrate how to define goals at different levels of abstraction (e.g., using decomposition) and then gradually introduce the desire system functionality. Rigorous modelling of the impact of agent failures on goal achievement allows us to derive a dynamic goal reallocation mechanism. It guarantees system resilience in the presence of agent failures. The defined set of modelling and refinement patterns represents generic solutions common to formal modelling of a multi-agent system and can be applied to a wide spectrum of such systems.

We illustrate our approach by a case study – a development of a cleaning multi-robotic system. We perform its formal development via instantiation of the proposed patterns. While modelling the behaviour of a multi-robotic system, we show that the refinement process allows us to discover the restrictions that we have to impose on the system behaviour to guarantee its resilience. Our approach demonstrates a good scalability and facilitates development of such complex distributed systems as multi-robotic systems.

Paper VI: A Case Study in Formal development of a Fault Tolerant Multi-Robotic System

The Paper VI further exploits a combination of the goal-oriented development and the Event-B refinement for modelling complex distributed systems. The aim of this work is to perform a large-scale validation by extending our multi-robotic cleaning system case study with new functionalities and focus on modelling component cooperation and system reconfigurability. The resulted system has a heterogeneous architecture and consists of different types of components that might fail. The goal of our development is to formally derive a specification of a robotic system and verify that the proposed design ensures goal reachability despite unreliability of the system components. By relying on the results discussed in Paper V, here we demonstrate how to define a system goal and decompose it into the corresponding subgoals of finer granularity until the desired level of details is reached.

While modelling the chosen system, we gradually introduce a representation of the main system elements and dependencies between them. Moreover, we introduce the notions of local and global knowledge to capture a discrepancy between agent perception of its goal and the actual global system state. We propose an advanced reconfiguration mechanism that relies on goal reallocation to guarantee goal reachability. During the development process, we identify the main system-level properties (e.g., consistency between the global and local knowledge) and demonstrate how to formally specify and verify them as a part of the refinement process.

We find the refinement approach to be beneficial for deriving precise requirements of the multi-robotic system and finding appropriate modelling solutions. During the refinement process, we discovered a number of subtleties in the system requirements. In particular, we had to impose additional restrictions on the behaviour of a base station when it takes a new responsibility for other subgoals and robots.

Paper VII: Formal Development and Quantitative Assessment of a Resilient Multi-robotic System

This paper present an integrated approach to formal development and assessment of a resilient system. Specifically, we demonstrate how to integrate probabilistic analysis into the refinement process. This work complements our previous works on modelling and verification of multi-robotic systems. To prove the goal reachability (and therefore guarantee system resilience), we have made artificial assumptions concerning component failures. In this paper, we employ quantitative analysis techniques to evaluate these artificial assumptions. To facilitate an integrated development and assessment, we augment our formal models with statistical data and rely on probabilis-

tic verification. Namely, we use the probabilistic model checker PRISM to assess the probability of achieving the system goal in the presence of component failures.

The general idea of the proposed approach is as follows. In a number of Event-B refinement steps we gradually define the system functionality and prove the desired system properties. When a detailed logical architecture is derived by refinement, we augment the obtained Event-B model with the probabilistic information required to conduct the probabilistic resilience assessment. In particular, we annotate actions of all model events with the real-valued rates and then transform a probabilistically augmented Event-B specification into a continuous-time Markov chain. Then, we use the probabilistic model checker PRISM to compute the probability of achieving the global system goal in the presence of components failures. Moreover, we show how we can conduct different assessment of the sources of system failures, system performance, etc. As a result, our approach allows us to compare several alternative system configurations.

Paper VIII: Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems

This paper presents a formalisation of a resilient goal-oriented multi-agent system and its essential properties. Our formalisation covers the notions of system goals and agents as well as various formal structures defining different interrelationships between these notions. Moreover, our formalisation defines constraints on the system dynamics allowing a multi-agent system to become more reconfigurable and thus resilient in order to achieve the system goals. We rely on the goal-oriented paradigm because it provides us with an especially suitable basis for reasoning about reconfigurability. In particular, it allows us to define reconfigurability as an ability of agents to redistribute their responsibilities to ensure goal reachability. Our proposed formal systematisation of the involved concepts can be seen as generic guidelines for formal development of reconfigurable systems. Specifically, in the second part of the paper, we demonstrate how such guidelines can be defined within the Event-B framework.

Paper IX: Formal Modelling of Resilient Data Storage in Cloud

Reliable and highly performant handling of large data stores constitutes one of the major challenges of cloud computing. The paper presents a formalisation of an industrial approach to implementing a resilient cloud data store. The solution combines the write-ahead-logging (WAL) mechanism with log replication.

In our work, we rely on the Event-B refinement to derive specifications of the synchronous, semi-synchronous and asynchronous replication architec-

tures. Event-B allows us to exlicitly express the required data consistency and data integrity properties as model invariants and compare them in all three models. During the development, we identify situations that may lead to data loss and introduce an explicit representation of the transition period to mitigate these losses. In general, these properties inform the industry practitioners on the resilience guarantees provided by each solutions. The proposed modelling approach can facilitate early design exploration and evaluate benefits of different fault tolerance mechanisms in implementing resilience requirements.

Paper X: Integrating Event-B Modelling and Discrete-Event Simulation to Analyse Resilience of Data Stores in the Cloud

Paper X is a logical continuation of Paper IX. The approach proposed in Paper IX allows us represent and verify logical system-level properties and qualitatively evaluate them. In its turn, discrete-event simulation in SimPy enables perform the quantitative system analysis. In particular, it allow us to evaluate some performance and reliability characteristics that are crucial for services in the cloud.

To facilitate integration with SimPy and discussions with the industrial engineers, we have created a simple graphical notation – a process-oriented model. The proposed notation introduces only the core concepts of the domain together with key artefacts required for formal modelling and simulation. Such a graphical model defines the component interactions, representation of statistical parameters, reactions on different stimulus (e.g., faults).

The proposed process-oriented model plays the role of an unifying blueprint of the system and allows us to define the required structure of the Event-B and simulation models as well as provide an easy-to-comprehend visual representation to the engineers. Once the initial models are derived from the created process-oriented model, the resulting Event-B model is refined to represent and verify data integrity properties, while the corresponding simulation model is executed to analyse the performance/reliability ratio, e.g., under different service and failure rates.

Author's contribution: The author of this thesis was responsible for the Event-B development of the all case studies presented in Papers I-X. The author has also made contribution to investigating the theoretical results proposed in the papers under supervision of Assoc. Prof. Elena Troubitsyna and Docent Linas Laibinis. Probabilistic modelling and verification in PRISM presented in Paper I and Paper VII was conducted by Dr. Anton Tarasyuk. The discrete-event simulation in SimPy presented in Paper X was done by M.Sc. Benjamin Byholm.

Chapter 6

Related Work

In this chapter we survey the fields of research relevant to the objectives of this thesis. Specifically, we start by giving an overview of the research done on goal-oriented development. Next, we summarise the methods for modelling and verification multi-agent systems. Then, we briefly review the works done on integration of simulation techniques with formal modelling. Finally, we consider the field of self-management, system adaptation and dynamic reconfiguration, listing several research directions and works on the frameworks and techniques for managing these aspects.

6.1 Goal-Oriented Development

Significant work has been done on goal-oriented requirement engineering approaches. The foundational work on *goal-oriented development* belongs to van Lamsweerde [40, 171, 172]. The proposed KAOS framework [40] introduces a goal-oriented approach for requirements modelling, specification, and analysis as well as addresses both functional and non-functional system requirements. Based on the KAOS framework, Lamsweerde [170] has proposed a method for deriving the software architecture from its requirements. Specifically, according to the method, a software specification is developed from the given system requirements and then used to build the architectural system design. The design is developed by consecutive refinements, which take into account the system constraints and non-functional goals. The KAOS approach is supported by the GRAIL tool [40].

Over the last decade, the goal-oriented approach has also received several extensions that allow the designers to link it with formal modelling [103, 131]. In particular, the work [103] presents a translation technique of KAOS operational models into event-based tabular specifications, which can be then analysed by the SCR* toolset [76]. The technique consists of a number of transformation steps, each of which solves semantic, structural

or syntactic differences between the KAOS and SCR (Software Cost Reduction) languages.

A significant body of research has been also devoted for translating formal specifications built according to the KAOS goal-oriented method into event-based transition systems. For example, the work [107] presents an approach to use the formal analysis capabilities of LTSA (Labelled Transition System Analyser) to analyse and animate KAOS operational models. The mapping allows the designers to translate goal-oriented operational requirements into a black-box event-based model of the software behaviour, expressed in a formalism appropriate to reason about system behaviours at the architectural level.

One of the first attempts to bridge the KAOS goal-oriented framework with the B formalism was presented in [132]. More recently, the study to formalise KAOS requirements in Event-B was attempted in [11]. The paper proposes a constructive approach that allows linking of high-level system requirements expressed as linear temporal logic formulae to the corresponding Event-B elements. The notion of a triggered event is used to translate time operators that are used in KAOS models. Similar, Matoussi et al. [116, 117] present works on coupling requirements engineering methods with formal methods. In contrast, in our work we have relied on goals to facilitate structuring of the system behaviour, while connecting them with the required agent collaboration and system reconfiguration mechanisms.

The goal behind our research is to both formally model and verify systems with intricate relationships between agent and goal structures, that are able to dynamically reconfigure themselves in order to tolerate various failures or changes.

6.2 Modelling and Verification of Multi-Agent Systems

The field of design of multi-agent systems has considerable evolved over the last decade. Surveying the literature on MAS reveals a significant amount of research devoted to different agent organisation concepts, agent specification languages and platforms, modelling and verification of the agent behaviour, etc. The resulting approaches vary significantly in terms of the covered topics, such as agent interoperability, communication, roles, goals and beliefs. Below we outline only a few works most relevant to our research.

The Tropos methodology [32] supports analysis and design in the development of agent-based software systems. UML diagrams are used to represent the system goals, agents, their capabilities and interdependencies, as well as system properties and agent interactions. An extension of this work [123] also supports modelling of agent errors and recovery activities.

Another proposed methodology – Multi-Agent System Engineering (MaSe) [46] – guides the designer through the software lifecycle of a multi-agent system. It allows a graphical representation of the system goals, the associated use cases and agent roles. Finite state automata are used to express communications between agent classes. The accompanied tool, the Agent Tool, supports the agent system development following the MaSe methodology. An extension of this work, Organization-based MaSE (O-MaSe) [47], provides a mechanism for defining agent interactions with the environment via external actors as well as defining interaction protocols between the system and the actors. O-MaSE makes use of UML class diagrams and does not support formal notation.

Formal modelling of agent systems has been undertaken by [147, 146, 144, 145]. The authors have proposed an extension of the UNITY framework to explicitly define such concepts as mobility and context-awareness. The mobile UNITY [147] extension proposes a notation to express mobile computations and a logic for reasoning about component temporal properties. It also supports formal reasoning about mobile components and their behaviour. On the other hand, the Context UNITY extension [145] formalises context-aware computing, with a proposed notation to represent the system context. The sensed aspects of the environment are used by the system to adjust its behaviour. In our formalisation we have pursued a different goal – we aimed at formally guaranteeing that the specified agent behaviour with the incorporated mechanisms facilitates achieving the defined system goals. We also have studied the problem of ensuring access to the fresh context. However, in [145] this problem is solved at the level of the matching agent attributes, while in our approach we rely on the scoping mechanism [110, 102, 85] to achieve this.

Formal modelling of fault tolerant MAS in Event-B has been also undertaken by Ball and Butler [19]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach we consider fault tolerance as a part of ensuring dependability and safety of MAS. Moreover, we have formalised a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability. Application of Event-B in a formal development of situated multi-agent systems has been investigated in work [104]. In particular, the Event-B framework has been used to develop a case study focused on a platoon of vehicles and verify the safety properties related to their simultaneous movement.

Significant amount of works is dedicated to the specification and verification of multi-agent systems based on the Belief-Desire-Intention (BDI) model of agency [41]. The research is mostly focus on studying the mental agent characteristics such as agent beliefs, desired and intentions.

An approach to formal development of multi-agent systems based on temporal logic is presented in the work [61]. The paper describes how formal specification, verification and refinement techniques can be used to represent and analyse both the behaviour of individual agents and the behaviour of multi-agent systems. The approach mostly focused on representation of agent's beliefs, desires and intentions introduced in BDI framework [70]. In our work we have not aimed at modelling the mental attributes of system agents. In contrast, we use an agent concept as a powerful and expressive abstraction for expressing inter-connections and collaborations of different system components.

The use of model-checking techniques for reasoning about MAS properties has been actively researched as well (see, e.g., [28, 29, 111, 74, 112]). In particular, [29] presents a framework for verification of agent programs with the respect to given BDI agent specifications. In the proposed approach, an agent system is first programmed using the logic-based agent oriented programming language AgentSpeak(F). Then the obtained AgentSpeak(F) programs are translated into Promela – the specification language of the SPIN model checker – to verify the resulting system. Ferrari et al. [59] describe verification of the chosen π-calculus based process algebra for mobile agents, while [112] presents modelling of fault-tolerant agents by stochastic Petri nets. The paper [111] describes the symbolic model checker MCMAS, specifically tailored for verification of MAS. The MCMAS tool takes as inputs models describing both agents and working environment of a multi-agent system and applies the epistemic logic to analyse it. However, model checking approaches typically suffer from the state space explosion problem, which is especially acute for large systems. Since Event-B is based on theorem proving, this helps us to avoid the mentioned problem.

There is a number of papers that discuss the use of probabilistic model techniques for analysis of multi-agents systems. For example, in [83], the authors address model-checking of probabilistic knowledge (relative to the agent knowledge) by developing an algorithm in the MCK model checker, while in [74], the authors represent a MAS as a discrete-time Markov chain and verify such system properties as convergence and convergence rate in the PAT model checker. Yet, to the best of our knowledge, the approaches combining both theorem proving and verification via model checking are still missing.

Proactive agent behaviour has been traditionally modelled in terms of the system goals that agents aim to achieve in order to meet their objectives. There are many works focussed on the notion of a goal from the agent perspective, e.g., [31, 173, 166, 184]. For example, the work [173] presents a unifying framework for goals, treating them as LTL formulae that describe desired goal progressions: adoption, pursuit, and dropping of goals. Later, in [184] the Winikoff investigates the required interactions between goals and

provides a framework for reasoning about such interactions. In our work, we focus on studying inter-connections between the system goals as well as between goals and agents required to perform such goals.

6.3 System Resilience

The research investigating software architectures with respect to their dependability properties has received recent attention [72, 44, 43]. Guerra et al. propose in [72] the concept of a idealised fault-tolerant component that can be employed as a building block within the architectural description of a system. The concept allows the designers to identify the system parts that are responsible for coping with faults. Therefore, the system in general is viewed as a set of components interacting within the architecture designed to handle software faults, thus providing a higher level of dependability. However, these approaches are characterised by a high degree of uncertainty in achieving fault tolerance that is unsuitable for class of the safety-critical systems that we focus on.

The problem of system reconfiguration and its connection to predictable dynamic resilience is presented in works [154, 153]. The proposed architectural framework – MetaSelf – is suitable to development of dynamically resilient systems using a specific architectural model. The main idea of the approach is to separate the functional and non-function descriptions of the system components, formulate necessary resilience policies and reason about system reconfiguration based on these policies. The key feature of the proposed solution is the need for metadata – information about system components – sufficient to enable decision-making about dynamic system reconfiguration. The metadata are used to guide the system reconfiguration with respect to the given reconfiguration policies. In [153] the authors present a system development method for the MetaSelf framework. They identify the phases and tasks that should be performed during the MetaSelf development process and illustrate the method application on two examples: dependable Web services and industrial assembly systems. Despite the provided evidence of the applicability of the proposed framework, there is a lack of architectural models, languages, and notations for the description of system configuration changes.

A body of research done on quantitative assessment of system resilience aspects is presented in works [159, 169, 6]. The paper [6] discusses benchmarking of resilience of self-adaptive systems, while [169] and [159] deal with resilience in the context of network systems. The measurable aspects of network resilience as a function of time have been studied in [21]. In this paper, resilience is intertangled with the notions of system vulnerability and recoverability. The authors study the resilience-based component importance measures. Specifically, they investigate which components are most

influential regarding the performance and reliability issues, and therefore affect the resilience of the entire network. The problem of empirical assessment of resilience is discussed in work [130]. Patariczа et al. review the main requirements on the statistical background needed for resilience characterisation and present an approach based on Exploratory Data Analysis (EDA) helping to understand the impact of changes and their quantitative characterisation.

It is also worth mentioning the ReSIST project [139] that addresses achieving sufficient resilience in the complex systems of ever evolving networks of computers and mobile devices. The research conducted within this project covers such system characteristics as evolvability, assessability, usability and diversity. These characteristics are studied in the context of fault tolerant agreement protocols [114, 122], wireless sensor network algorithms [25], testing in mobile settings [175, 124], evolving resilient usable systems [115, 125], and others.

The similar idea of modelling a dynamic system architecture with the integrated reconfiguration mechanisms and assessing system resilience characteristics is presented in paper [101]. Using Event-B and its refinement technique, the authors derive a complex architecture of data processing capabilities of CPS. To assess the resilience of the obtained data processing architecture, the authors rely on the statistical model checking UPPAAL.

6.4 Simulation and Formal Modelling

There is a significant amount of work on the topic of modelling and simulation of distributed systems. The problem of integration of model-based formal methods with discrete-event models in the development of complex embedded systems is addressed within the DESTECS project [52]. The research proposes an approach to the model-based design through co-simulation of discrete-event models in the Vienna Development Method (VDM) and continuous-time models in 20-sim [1]. These models are coupled by a co-simulation tool that coordinates execution of the developed models in their respective simulators. The resulting models can be also augmented with descriptions of potential failures and fault tolerance mechanisms [64, 63, 65].

The problem of formal verification and simulation-based validation is also addressed in the ADVANCE project [5]. However, the main focus of the proposed methodologies is related to cyber-physical systems, which are characterised by a mixture of discrete-event and continuous-time components. The proposed simulation based approach combines the Event-B development and co-simulation with tool independent physical components via the FMI interface [151]. An approach on combining formal modelling with a simulation technique for Wireless Sensor Network (WSN) is presented

in works [91, 92]. The proposed approach is based on co-simulation using Event-B models of WSN and the MiXiM environment simulation engines. In contrast, in our work we deal with discrete-event systems and focus on integrating separate approaches for qualitative and quantitative reasoning about such systems.

The Ptolemy project [135] studies modelling, simulation, and design of concurrent, real-time, embedded systems. It proposes a component-based, actor-oriented approach in which components are concurrent and interact by message-passing [42], and which supports heterogeneous modelling and design for Modelica [56]. The key underlying focus in the project is the use of well-defined models of computation that govern interactions between components. However, the dependability issues, e.g., fault tolerance modelling, are not properly supported within the proposed framework.

The problem of inadequate integration between formal reasoning about correctness and simulation has also been identified by Boer et al [27]. They propose to explicitly represent the notion of resources into an abstract behaviour specification language. In our case, Event-B allows us to formally represent and verify system-level properties, while in [27] the stress is put on creating executable specifications and analysis of the corresponding traces.

Our proposed process-oriented model described in Paper X is similar to Activity Cycle Diagrams (ACD) [60, 53] – a graphical notation to model discrete events and interactions. In particular, [60] presents an extension of ACS to enable automatic translation of them to Java programs, while [53] proposes extended ACD to represent the relationship between conditions and events in a discrete event system that are not covered by the classical ACD. In contrast, our process-oriented models allow us to represent a high level system architecture in terms of components, processes and their interactions. Moreover, our proposed models can be both used as a basis to formal modelling and simulation at the same time.

6.5 Self-*, Adaptive and Reconfigurable Systems.

The topic of system adaptability has been significantly studied since the middle of 60's years [113]. Different interpretations and concepts have been proposed by the research community since then, e.g., self-organising, self-adaptive and reconfigurable systems [49, 84]. These paradigms are all intrinsically intertangled and act to enable system *autonomy*, while maintain system capabilities. MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) control-loop model was proposed for autonomous architectures [98]. The idea of autonomic control loop is to collect and evaluate system parameters, analyse them (i.e., determine whether adaptation actions are required), plan mitigation actions if needed and execute them. The planning component is intended to identify alternative configurations that satisfy current

contextual requirements and constraints.

Self-adaptation has been widely recognised as an effective approach to deal with the complexity and dynamicity of modern software systems [182]. Formal methods provide the means to rigorously specify and verify the behaviour of self-adaptive systems. Formal methods have been applied during system development, but also during runtime to provide guarantees about the required properties of self-adaptive systems [186, 174, 163].

Software architecture-based self-adaptation, employing the MAPE loops into system design, is presented in works [68, 69, 37, 36]. The proposed Rainbow framework provides a reusable architectural model as well as the means to reason about constraint violation and system adaptation capabilities at runtime. The framework adopts the standard component-connector paradigm of the software architecture which allows the designers to identify system elements as well as the relationships between them as well as their interactions. Moreover, the additional values and constrains might be annotated on the architectural elements such as throughput, latency. The obtained model is extended with adaptation operators and strategies. The adaptation operators define a set of actions that might change the system configuration, while the adaptation strategies define the rules and properties for changing the current configuration. Though various implementations of the proposed framework are presented, no rigorous formal technique is used in the construction of the resulting architecture.

In work [176] the authors propose a goal-oriented approach for self-reconfiguration by employing the idea of monitoring feedback loop. The proposed framework uses goal models as software requirements models, and employs SAT solvers to check the current execution records against the models to diagnose task failures. The monitoring component monitors requirements and record data, while the diagnostic component analyses the recorded data and identifies failures of the system behaviour. In our work, we focus on modelling the system behaviour and dependencies between components, while not distinguishing the system components to be the monitoring or diagnostic ones. However, this idea can be incorporated into our system design.

A number of architectural patterns and design choices for self-adaptation via feedback loops are proposed in [186, 136, 181]. In works [136, 186] authors propose new design solutions to support monitoring, decision making, and reconfiguration of adaptive systems. In [180] the authors introduce the FORMS model (FOrmal Reference Model for Self-adaptation) that allows designers to describe and evaluate alternative design choices for self-adaptive systems. The FORMS model defines a shared vocabulary of adaptive primitives that can be used to precisely define complex self-adaptive systems.

A large variety of the *dynamic reconfiguration* aspects has been studied in the last decade. For instance, Wermelinger et al. [179] have proposed a

high-level language for specifying dynamically reconfigurable architectures. They focus on possible modifications of architectural components and model reconfiguration by the algebraic graph rewriting. In our work, we see reconfigurability as an ability of components to redistribute their responsibilities to ensure system goals. Therefore, the proposed reconfiguration mechanisms are build by changing links and associations between components. In addition, we also focus on the functional aspects of reasoning about reconfiguration.

Significant research efforts are invested in finding suitable models of triggers for run-time adaptation. Such triggers monitor performance [35] or integrity [177] of the application under consideration and initiate reconfiguration when the desired characteristics are not achieved. In our work, we consider system reconfiguration as means to achieve system resilience. We perform the assessment of possible reconfiguration strategies at the development phase that allows us to rely on the existing error detection mechanisms to trigger dynamic reconfiguration. Moreover, to analyse non-functional characteristics such as reliability and performance, we integrate formal development with quantitative analysis.

An extensive body of research investigates the quality of service characteristics of dynamically reconfigurable service-oriented systems. Among the most prominent works in this area is the approach proposed by Calinescu et al. [34]. It aims at defining the optimal configuration with respect to quality of service by assessing the quality of service attributes of various service components that are available at run-time. We use similar probabilistic verification techniques to assess system reliability and performance. However, in our work we consider a system with the predefined set of components. This allows us to assess possible reconfiguration strategies at the development phase and thus simplify the reconfiguration process.

The idea to employ agent-based architectures for building autonomous systems has been realised in studies [50, 108, 178, 109]. According to the proposed research, an autonomous system has a distinguished decision-making component – agent, which behaviour can be formally verified by model-checking techniques. In particular, it allows one to explore all possible choices the agent might make and why [51, 178]. The works adopt the BDI approach [137, 138] and use the GWENDOLEN language [48] for programming rational agents.

The idea of achieving system dependability via reconfiguration is described in work [161]. The authors present a method for constructing systems where general properties of reconfiguration can be ensured via formal proofs. The idea of the proposed approach is to introduce a formal definition of reconfiguration as well as a set of hight-level properties. Then a system architecture is introduced which guarantees those reconfiguration properties. In our research, we follow the same idea to enable the system to

be reconfigurable already at a hight-level system specification.

In [93], Inverardi et al. investigate system adaptation based on the assume-guarantee concept. In particular, they propose a framework that allows the developers to efficiently define under which conditions adaptation can be performed by still preserving the desired system invariant properties. The framework also allows the designers to split the system into parts that can be substituted. In order to guarantee the correctness of adaptation, the special conditions are formulated and have to be proven at run-time. In our approach, the reconfiguration strategies are already defined at development phase and are incorporated into the system architecture. In the case of failures or changes, the system is able to reconfigure by changing interdependencies among components, as well as between the components and the system goals.

Chapter 7

Conclusions and Future Work

In this chapter we summarise the main achievements of this thesis, discuss the limitations of the proposed approaches and point out future research directions.

7.1 Research Conclusions

Formal methods are traditionally used in the development and verification of critical computer-based systems to ensure system correctness and trustworthiness. However, a highly dynamic nature of modern software-intensive systems requires further advances of formal techniques to facilitate reasoning about system resilience – an ability of the system to adapt to changes while remaining trustworthy. In this thesis, we propose a formal integrated approach that addresses this challenge.

Our approach relies on formal modelling and refinement in Event-B. The Event-B framework offers a powerful scalable basis for correct-by-construction development of complex distributed systems. In this thesis, we demonstrate how to specialise a generic process of distributed system development by refinement to explicitly address system resilience at different levels of abstraction.

We also discuss how, by modelling of not only the nominal system behaviour but also failures of the system components, rigorous systematic derivation of the required fault tolerance and reconfiguration mechanisms can be facilitated. Moreover, we investigate how the employed refinement approach can be used as a basis for the gradual unfolding of component-based system architectures where dynamic system reconfiguration is used as the means to enhance system resilience. As another source for system resilience, complex component interactions and their cooperative behaviour

are formally studied as a part of this thesis.

In the thesis, we have aimed at proposing a scalable formal development approach that can facilitate development of different types of resilient distributed systems. To achieve this objective, we combine refinement with goal-oriented development and define the resilience-explicit goal-oriented refinement process to facilitate development of complex multi-agent systems. Our proposed approach allows the developers to systematically introduce the necessary reconfiguration mechanisms to ensure that the system progresses towards achieving its goals despite agent failures or becomes more performant by using its agents more efficiently. Since reconfiguration is a powerful technique for achieving resilience, we propose a formalisation of the reconfigurability concept, by connecting it with the system goals, agents, and their inter-relationships.

Since quantitative assessment of different resilience characteristics is an essential part of the design for resilience, in the thesis we also show how Event-B models can be augmented with quantitative data and serve as a basis for quantitative resilience assessment during the design time.

We have validated our approach by a number of case studies from different domains, such as robotics, space, healthcare and cloud. However, we are also well aware of some limitations of the presented research. The formal models resulting from the proposed refinement-based development are still very abstract, with most of the system complexity hidden. Even if they allow us to investigate some essential properties of reconfigurable distributed systems, further refinement steps, bringing the models closer to a prototype implementation of the suggested resilience mechanisms in a chosen domain, are required to fully evaluate the approach potential. Moreover, to exploit the benefits of the proposed integration of formal modelling and quantitative assessment, the automatic support for translation between the involved modelling languages is sorely needed. The mentioned limitations also lead us to several possible future research directions that we discuss next.

7.2 Future Work

Resilience is a rapidly developing area that offers many opportunities for the research. One possible research direction is to experiment with deriving resilience monitors (i.e., monitoring components) from the formally developed system models to enable proactive resilience. Moreover, it will be interesting to focus on a specific domain and specialise our resilience-modelling approach for the concrete domain needs.

To facilitate the quantitative system assessment of Event-B models, it is necessary to develop of a plug-in for the Rodin platform that would support Event-B models augmented with probabilistic information. Such a plug-in

could be used to derive the corresponding Markov model from an augmented Event-B specification. This would make it possible to automatically generate PRISM specifications for performing further quantitative analysis.

Moreover, it would be beneficial to create a tool for automatic translation of Event-B models in SimPy. Such a tool would allow the designers to automatically generate a code skeleton for SimPy models. Finally, it would be advantageous to enhance our proposed intermediate graphical notation for visualisation of system components, processes and their interactions as well as provide tool support for it.

Bibliography

[1] 20sim. Modeling and simulation program for mechatronic systems, online at http://www.20sim.com/.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[3] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[4] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[5] Advanced Design and Verification Environment for Cyber-physical System Engineering (Advance). FP7 Information and Communication Technologies (ICT) Programme. online at http://www.advance-ict.eu/.

[6] R. Almeida and M. Vieira. Benchmarking the resilience of self-adaptive software systems: Perspectives and challenges. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 190–195, New York, NY, USA, 2011. ACM.

[7] R. Alur and T. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[8] A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and its Threats - A taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[10] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying Continuous Time Markov Chains. In *CAV'96, International Conference on Computer Aided Verification*, pages 269–276. Springer, 1996.

[11] B. Aziz, A. Arenas, J. Bicarregui, Ch. Ponsard, and Ph. Massonet. From goal-oriented requirements to event-b specifications. In *First NASA Formal Methods Symposium - NFM 2009*, pages 96–105, 2009.

[12] S. Bacherini, A. Fantechi, M.Tempestini, and N.Zingoni. A story about formal methods adoption by a railway signaling manufacturer. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2006.

[13] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM, 1983.

[14] R. J. R. Back and K. Sere. Stepwise Refinement of Action Systems. *Structured Programming*, 12(1):17–30, 1991.

[15] R. J. R. Back and K. Sere. From Action Systems to Modular Systems. *Software – Concepts and Tools*, 17(1):26–39, 1996.

[16] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2005.

[17] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.

[18] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate Symbolic Model Checking of Continuous-Time Markov Chains. In *CONCUR'99, International Conference on Concurrency Theory*, pages 146–161. Springer, 1999.

[19] E. Ball and M. Butler. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In *Methods, Models and Tools for Fault Tolerance*, pages 104–129. Springer, 2009.

[20] Jerry Banks. Principles of simulation. In *Handbook of Simulation*, pages 3–30. John Wiley & Sons, Inc., 2007.

[21] K. Barker, J. E.Ramirez-Marquez, and C. M. Rocco Sanseverino. Resilience-based network component importance measures. *Rel. Eng. & Sys. Safety*, 117:89–97, 2013.

[22] C. Barrett and C. Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[23] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[24] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR '04)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 13–23. Elsevier, 2005.

[25] C. Bernardeschi, P. Masci, and H. Pfeifer. Analysis of wireless sensor network protocols in dynamic scenarios. In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009*, volume 5873 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2009.

[26] M. Bernardo, F. Corradini, and K. G. Larsen, editors. *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004.

[27] Frank S. Boer, Reiner Hähnle, Einar B. Johnsen, Rudolf Schlatte, and Peter Y. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In *Service-Oriented and Cloud Computing*, LNCS 7592, pages 91–106. Springer, 2012.

[28] R. Bordini, M.Fisher, C. Pardavila, and M. Wooldridge. Model Checking AgentSpeak. In *AAMAS 2003*, pages 409–416. ACM Press, 2003.

[29] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[30] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993.

[31] L. Braubach, A. Pokahr, Daniel Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In *Programming Multi-Agent Systems, Second International Workshop ProMAS*, volume 3346 of *Lecture Notes in Computer Science*, pages 44–65. Springer, 2004.

[32] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[33] M. J. Butler. Decomposition structures for event-b. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, pages 20–38, 2009.

[34] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. volume 37, pages 387–409. IEEE Computer Society, 2011.

[35] M. Caporuscio, A. Di Marco, and P. Inverardi. Model-Based System Reconfiguration for Dynamic Performance Management. *J. Syst. Softw.*, 80:455–473, 2007.

[36] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009*, pages 132–141. IEEE, 2009.

[37] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *1st International Conference on Autonomic Computing (ICAC 2004)*, pages 276–277. IEEE Computer Society, 2004.

[38] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking - History, Achievements, Perspectives*, pages 1–26. Springer-Verlag Berlin Heidelberg, 2008.

[39] Coq. The Coq Proof Assistant, online at https://coq.inria.fr/what-is-coq.

[40] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 612–613. ACM, 1997.

[41] M. Dastani, K. V. Hindriks, and J.-J. Meyer. *Specification and Verification of Multi-Agent Systems.* Springer US, 1st edition, 2010.

[42] Galicia R. Goel M. Hylands C. Lee E. Liu J. Liu X. Muliadi L. Neuendorffer S. Reekie J. Smyth N. Tsay J. Davis, J. and Y. Xiong. Ptolemy-ii: Heterogeneous concurrent modeling and design in java. In *Technical Memorandum UCB/ERL No. M99/40*. University of California at Berkeley, 1999.

[43] R. de Lemos, C. Gacek, and A. Romanovsky. Architecting dependable systems. *Journal of Systems and Software*, 79(10):1359–1360, 2006.

[44] Rogério de Lemos, Paulo Asterio de Castro Guerra, and Cecília Mary F. Rubira. A Fault-Tolerant Architectural Approach for Dependable Systems. *Software, IEEE*, 23:80–87, 2006.

[45] L. Mendonça de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[46] S. A. DeLoach. The mase methodology. 11:107–125, 2004.

[47] Scott A. DeLoach and J. C. García-Ojeda. O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

[48] L. A. Dennis and B. Farwer. Gwendolen: A bdi language for verifiable agents. In *University of Aberdeen*, 2008.

[49] L. A. Dennis, M. Fisher, J. M. Aitken, S. M. Veres, Y. Gao, A. Shaukat, and G. Burroughes. Reconfigurable autonomy. *KI*, 28(3):199–207, 2014.

[50] L. A. Dennis, M. Fisher, A. Lisitsa, N. Lincoln, and S. M. Veres. Satellite control using rational agent programming. *IEEE Intelligent Systems*, 25(3):92–97, 2010.

[51] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Autom. Softw. Eng.*, 19(1):5–63, 2012.

[52] Design Support and Tooling for Embedded Control Software(DESTECS). FP-7 project. online at http://www.destecs.org.

[53] D.Kang and B. K. Choi. The extended activity cycle diagram and its generality. *Simulation Modelling Practice and Theory*, 19(2):785 – 800, 2011.

[54] A. Edmunds. Templates for event-b code generation. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 284–289. Springer, 2014.

[55] A. Edmunds, A. Rezazadeh, and M. J. Butler. Formal modelling for ada implementations: Tasking event-b. In *Reliable Software Technologies - Ada-Europe 2012 - 17th Ada-Europe International Conference on Reliable Software Technologies.*, volume 7308 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2012.

[56] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[57] M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein. Formal correctness, safety, dependability, and performance analysis of a satellite. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1022–1031, Piscataway, NJ, USA, 2012. IEEE Press.

[58] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[59] G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, October 2003.

[60] W. De Lara Araújo Filho and C. M. Hirata. Translating activity cycle diagrams to java simulation programs. In *ANSS'04*, pages 157–164. IEEE, 2004.

[61] M. Fisher. Temporal development methods for agent-based. *Autonomous Agents and Multi-Agent Systems*, 10(1):41–66, 2004.

[62] M. Fisher, L. A. Dennis, and M. P. Webster. Verifying autonomous systems. *Commun. ACM*, 56(9):84–93, 2013.

[63] J. S. Fitzgerald, P. G. Larsen, K. G. Pierce, and M. Verhoef. A formal approach to collaborative modelling and co-simulation for embedded systems. *Mathematical Structures in Computer Science*, 23(4):726–750, 2013.

[64] J. S. Fitzgerald, K. G. Pierce, and C. Gamble. A rigorous approach to the design of resilient cyber-physical systems through co-simulation. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN 2012*, pages 1–6. IEEE, 2012.

[65] J. S. Fitzgerald, K. G. Pierce, and P. G. Larsen. Co-modelling and co-simulation in the engineering of systems of cyber-physical systems. In *9th International Conference on System of Systems Engineering, SoSE 2014*, pages 67–72. IEEE, 2014.

[66] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, ECAI '96, pages 21–35. Springer-Verlag, 1997.

[67] Helm R. Johnson R.E. Vlissides J. Gamma, E. *Design Patterns.Elements of Reusable Object-Oriented Software*. Addison-Wesley Reading, 1995.

[68] D. Garlan, S.-W. Cheng, and B. R. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems [the book is a result of the ICSE 2002 Workshop on Software Architectures for Dependable Systems]*, volume 2677 of *Lecture Notes in Computer Science*, pages 61–89. Springer, 2003.

[69] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct 2004.

[70] Michael P. Georgeff, Barney Pell, Martha E. Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, ATAL '98, pages 1–10, London, UK, UK, 1999. Springer-Verlag.

[71] N. Guelfi, P. Pelliccione, H. Muccini, and A. Romanovsky. *An Introduction to Software Engineering and Fault Tolerance*, chapter Software Engineering of Fault Tolerant Systems. Series on Software Engineering and Knowledge Eng., 2007.

[72] P. A. Castro Guerra, C. Rubira, and R. de Lemos. A Fault-Tolerant Software Architecture for Component-Based Systems. In *Architecting Dependable Systems*, pages 129–143. Springer, 2003.

[73] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. In *Formal Aspects of Computing*, pages 512–535, 1994.

[74] J. Hao, S. Song, Y. Liu, J. Sun, L. Gui, J. Song Dong, and H. Leung. Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In *PRIMA 2012*, volume 7455 of *LNCS*, pages 16–30. Springer, 2012.

[75] Ian J. Hayes, Michael A. Jackson, and Cliff B. Jones. Determining the Specification of a Control System from That of Its Environment. In *FM 2003*.

[76] C. L. Heitmeyer, J. Kirby, B. G. Labaw, and R. Bharadwaj. Scr*: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification, 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 526–531. Springer, 1998.

[77] T. S. Hoang and J.-R. Abrial. Event-b decomposition for parallel programs. In *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 319–333, 2010.

[78] T. S. Hoang, A. Fürst, and J.-R. Abrial. Event-b patterns and their tool support. *Software and System Modeling*, 12(2):229–244, 2013.

[79] T. S. Hoang, A. Iliasov, R.Silva, and W. Wei. A survey on event-b decomposition. *ECEASST*, 46, 2011.

[80] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[81] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[82] J. Hörl and B. K. Aichernig. Formal specification of a voice communication system used in air traffic control. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, page 1868. Springer, 1999.

[83] X. Huang, C. Luo, and R. van der Meyden. Symbolic Model Checking of Probabilistic Knowledge. In *TARK 2011*, pages 177–186. ACM, 2011.

[84] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[85] A. Iliasov, V. Khomenko, M. Koutny, and A. Romanovsky. On specification and verification of location-based fault tolerant mobile systems. In *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*, pages 168–188. Springer, 2006.

[86] A. Iliasov, L. Laibinis, E. Troubitsyna, and A. Romanovsky. Support of Indexed Modules in Event-B. In *Proceedings of the 4th Rodin User and Developer Workshop*, pages 29–30. TUCS Lecture Notes, 2013.

[87] A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky. Patterns for refinement automation. In *Formal Methods for Components and Objects - 8th International Symposium,FMCO 2009*, volume 6286 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2010.

[88] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In *Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ 2010)*, volume 5977, pages 174–188. Lecture Notes in Computer Science, Springer, 2010.

[89] Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). IST FP7 IP Project. online at http://www.deploy-project.eu/.

[90] Industrial Use of the B Method. http://www.methode-b.com/wp-content/uploads/2012/08/ClearSy-Industrial_Use_of_B1.pdf.

[91] A. Intana, M. Poppleton, and G. V. Merrett. Adding value to WSN simulation through formal modelling and analysis. In *4th International Workshop on Software Engineering for Sensor Network Applications, SESENA 2013, San Francisco, CA, USA, May 21, 2013*, pages 24–29. IEEE, 2013.

[92] A. Intana, M. Poppleton, and G. V. Merrett. A formal co-simulation approach for wireless sensor network development. *ECEASST*, 70, 2014.

[93] P. Inverardi, P. Pelliccione, and M. Tivoli. Towards an assume-guarantee theory for adaptable systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009*, pages 106–115. IEEE, 2009.

[94] N. R. Jennings. On agent-based software engineering. *Artif. Intell.*, 117(2):277–296, 2000.

[95] R. Praful Jetley, C. Carlos, and S. Purushothaman Iyer. A case study on applying formal methods to medical devices: computer-aided resuscitation algorithm. *STTT*, 5(4):320–330, 2004.

[96] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 156–163. IEEE Computer Society, 1998.

[97] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TRANS. KNOWL. DATA ENG*, 15:2003, 1999.

[98] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[99] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11, International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[100] M. Kwiatkowska and D. Parker. Advances in Probabilistic Model Checking. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 126–151. IOS Press, 2012.

[101] L. Laibinis, D. Klionskiy, E. Troubitsyna, A. Dorokhov, J. Lilius, and M. Kupriyanov. Modelling resilience of data processing capabilities of CPS. In *Software Engineering for Resilient Systems - 6th International Workshop, SERENE 2014*, volume 8785 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2014.

[102] L. Laibinis, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Rigorous development of fault-tolerant agent systems. In *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2006.

[103] R. De Landtsheer, E. Letier, and A. van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. *Requir. Eng.*, 9(2):104–120, 2004.

[104] A. Lanoix. Event-b specification of a situated multi-agent system: Study of a platoon of vehicles. In *Theoretical Aspects of Software En-*

gineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on, pages 297–304, 2008.

[105] J.-C. Laprie. Resilience for the Scalability of Dependability. In *Fourth IEEE International Symposium on Network Computing and Applications*. IEEE, 2005.

[106] J.-C. Laprie. From Dependability to Resilience. In *DSN 2008, Dependable systems and Networks*. IEEE Computer Society, 2008.

[107] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206, 2008.

[108] N. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher, and A. Lisitsa. An agent based framework for adaptive control and decision making of autonomous vehicles. In *10th IFAC International Workshop on the Adaptation and Learning in Control and Signal Processing, ALCOSP 2010*, pages 310–317. International Federation of Automatic Control, 2010.

[109] Nick K. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher, and A. Lisitsa. Autonomous asteroid exploration by rational agents. *IEEE Comp. Int. Mag.*, 8(4):25–38, 2013.

[110] L.Laibinis, E.Troubitsyna, A. Iliasov, and A. Romanovsky. Fault tolerant middleware for agent systems: A refinement approach, 2009.

[111] A. Lomuscio, H. Qu, and F.Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *CAV 2009*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.

[112] Michael R. Lyu, Xinyu Chen, and Tsz Yeung Wong. Design and evaluation of a fault-tolerant mobile-agent system. *IEEE Intelligent Systems*, 19(5):32–38, 2004.

[113] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267 – 7279, 2013.

[114] P. Masci, H. Moniz, and A. Tedeschi. Services for fault-tolerant conflict resolution in air traffic management. In *SERENE 2008,*, pages 121–125. ACM, 2008.

[115] M. Massink, D. Latella, M. H. ter Beek, M. D. Harrison, and M. Loreti. A fluid flow approach to usability analysis of multi-user systems.

In *Engineering Interactive Systems, Second Conference on Human-Centered Software Engineering, HCSE 2008*, volume 5247 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2008.

[116] A. Matoussi, F. Gervais, and R. Laleau. A first attempt to express KAOS refinement patterns with event B. In *Abstract State Machines, B and Z, First International Conference, ABZ 2008*, page 338, 2008.

[117] A. Matoussi, F. Gervais, and R. Laleau. A goal-based approach to guide the design of an abstract event-b specification. In *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, pages 139–148, 2011.

[118] D. Mery and R. Monahan. Transforming event b models into verified c# implementations. In Alexei Lisitsa and Andrei Nemytykh, editors, *VPT 2013*, volume 16 of *EPiC Series*, pages 57–73, 2013.

[119] D. Méry and N. K. Singh. Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188. ACM, 2011.

[120] S. P Miller. Specifying the mode logic of a flight guidance system in core and scr. In *In Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, volume 4916 of *Lecture Notes in Computer Science*, page 44–53. ACM, 1998.

[121] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17:94–162, 1992.

[122] H. Moniz, A. Tedeschi, N. Ferreira Neves, and M. Correia. A distributed systems approach to airborne self-separation. In *Computational Models, Software Engineering and Advanced Technologies in Air Transportation: Next Generation Applications*. IGI Global, 2010.

[123] M. Morandini, L. Penserini, and A. Perini. Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 9–16. ACM, 2008.

[124] Minh Duc Nguyen, Hélène Waeselynck, and Nicolas Rivière. Testing mobile computing applications: Toward a scenario language and tools. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08. ACM, 2008.

[125] R. De Nicola, D. Latella, M. Loreti, and M. Massink. Marcaspis: A markovian extension of a calculus for services. *Electron. Notes Theor. Comput. Sci.*, 229(4), August 2009.

[126] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[127] OMG Mobile Agents Facility (MASIF). http://www.omg.org.

[128] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: combining specification, proof checking, and model checking. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[129] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[130] A. Pataricza, I. Kocsis, Á. Salánki, and L. Gönczy. Empirical assessment of resilience. In *Software Engineering for Resilient Systems, 5th International Workshop, SERENE 2013,*, volume 8166 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[131] C. Ponsard, G. Dallons, and M. Philippe. From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In *ERCIM News (75)*, pages 22–23, 2008.

[132] C. Ponsard and E. Dieul. From requirements models to formal specifications in B. In *Proceedings of the CAISE*06 Workshop on Regulations Modelling and their Validation and Verification ReMo2V '06*, 2006.

[133] PostgreSQL: WAL. http://www.postgresql.org/docs/9.2/static/wal-intro.html.

[134] PRISM. Probabilistic Symbolic Model Checker. online at http://www.prismmodelchecker.org/.

[135] Ptolemy Project. The Berkeley Ptolemy Project. online at http://ptolemy.eecs.berkeley.edu/.

[136] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010*, pages 49–58. ACM, 2010.

[137] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 439–449. Morgan Kaufmann, 1992.

[138] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319. The MIT Press, 1995.

[139] Resilience for Survivability in IST (ReSIST). ReSIST. online at http://www.resist-noe.org/.

[140] A. Riazanov and A. Voronkov. Vampire. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[141] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[142] Rodin. Event-B Platform. online at http://www.event-b.org/.

[143] RODIN Modularisation Plug-in. Documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.

[144] G.-C. Roman, C. Julien, and J. Payton. A formal treatment of context-awareness. In *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 12–36. Springer, 2004.

[145] G.-C. Roman, C. Julien, and J. Payton. Modeling adaptive behaviors in context UNITY. *Theor. Comput. Sci.*, 376(3):185–204, 2007.

[146] G.-C. Roman and P. J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20(1):47–68, 2002.

[147] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. *ACM Trans. Softw. Eng. Methodol.*, 6(3):250–282, 1997.

[148] J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, 1993.

[149] J. M. Rushby. Analyzing cockpit interfaces using formal methods. *Electr. Notes Theor. Comput. Sci.*, 43:1–14, 2001.

[150] M. Y. Said, M. J. Butler, and C. F. Snook. Language and tool support for class and state machine refinement in UML-B. volume 5850 of *Lecture Notes in Computer Science*, pages 579–595. Springer, 2009.

[151] V. Savicks, M. Butler, J. Colley, and J. Bendisposto. Rodin multi-simulation plug-in. In *5th Rodin User and Developer Workshop*, 2014.

[152] T. J. Schriber and D. T. Brunner. How discrete-event simulation software works. In Jerry Banks, editor, *Handbook of Simulation*, pages 765–812. John Wiley & Sons, 2007.

[153] G. Di Marzo Serugendo, J. S. Fitzgerald, and A. Romanovsky. Meta-self: an architecture and a development method for dependable self-* systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 457–461. ACM, 2010.

[154] G. Di Marzo Serugendo, J. S. Fitzgerald, A. Romanovsky, and N. Guelfi. A metadata-based architectural model for dynamically resilient systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 566–572. ACM, 2007.

[155] R. Silva, C. Pascal, T. S. Hoang, and M. J. Butler. Decomposition tool for event-b. *Softw., Pract. Exper.*, 41(2):199–208, 2011.

[156] SimPy. Simulation framework in Python, online at http://simpy.readthedocs.org/.

[157] C. F. Snook and M. J. Butler. UML-B: formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[158] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 148–155. IEEE Computer Society, 1998.

[159] J. P.G. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, and P. Smith. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Comput. Netw.*, 54(8):1245–1265, June 2010.

[160] L. Strigini. Resilience: What is it, and how much do we want? *IEEE Security & Privacy*, 10(3):72–75, 2012.

[161] E. A. Strunk and J.C. Knight. Dependability through assured reconfiguration in embedded system software. *IEEE Trans. Dependable Sec. Comput.*, 3(3):172–187, 2006.

[162] A. Stump, D. L. Dill, C. W. Barrett, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the* 16^{th} *IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 29–37. IEEE Computer Society, June 2001. Boston, Massachusetts.

[163] Gabriel Tamura, NorhaM. Villegas, HausiA. Müller, JoãoPedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. Towards practical runtime verification and validation of self-adaptive software systems. In Rogério de Lemos, Holger Giese, HausiA. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 108–132. Springer Berlin Heidelberg, 2013.

[164] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In *IFM 2012, Integrated Formal Methods*, volume 7321 of *LNCS*, pages 237–252. Springer, 2012.

[165] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Integrating stochastic reasoning into event-b development. *Formal Asp. Comput.*, 27(1):53–77, 2015.

[166] J. Thangarajah, J. Harland, D. N. Morley, and N. Yorke-Smith. Operational behaviour for executing, suspending, and aborting goals in BDI agent systems. volume 6619 of *Lecture Notes in Computer Science*. Springer, 2011.

[167] The HOL System. online at http://www.cl.cam.ac.uk/research/hvg/HOL/.

[168] The ProB Animator and Model Checker. online at http://www.stups.uni-duesseldorf.de/ProB/index.php5/.

[169] K.S. Trivedi, D. S. Kim, and R. Ghosh. Resilience in computer systems and networks. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 74–77, New York, NY, USA, 2009. ACM.

[170] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures, Third International*

School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, Advanced Lectures, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2003.

[171] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering*, pages 249–263, 2001.

[172] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[173] M. B. van Riemsdijk and and M. Winikoff M. Dastani. Goals in agent systems: A unifying framework. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '08, pages 713–720. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[174] E. Vassev and M. Hinchey. ASSL: A software engineering approach to autonomic computing. *IEEE Computer*, 42(6):90–93, 2009.

[175] H. Waeselynck, Z. Micskei, M. Duc Nguyen, and N. Riviere. Mobile systems from a validation perspective: a case study. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 85–92. IEEE Computer Society, 2007.

[176] Y. Wang, S.A. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 293–302. ACM, 2007.

[177] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An Automated Formal Approach to Managing Dynamic Reconfiguration. In *ASE 2006*, pages 18–22. Springer, 2006.

[178] Matthew P. Webster, M. Fisher, N. Cameron, and M. Jump. Formal methods for the certification of autonomous unmanned aircraft systems. In *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011*, volume 6894 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2011.

[179] M. Wermelinger, A. Lopes, and J.L. Fiadeiro. A Graph Based Architectural Reconfiguration Language. *SIGSOFT Softw. Eng. Notes*, 26:21–32, 2001.

[180] D. Weyns, S. Malek, and J. Andersson. FORMS: unifying reference model for formal specification of distributed self-adaptive systems. *TAAS*, 7(1):8, 2012.

[181] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II - International Seminar*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer, 2013.

[182] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. A survey of formal methods in self-adaptive systems. In *Fifth International C* Conference on Computer Science & Software Engineering, C3S2E '12*, pages 67–79. ACM, 2012.

[183] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *In Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000*, pages 264–274, 2000.

[184] M. Winikoff. An integrated formal framework for reasoning about goal interactions. In *Declarative Agent Languages and Technologies IX - 9th International Workshop,*, pages 16–32.

[185] J. C. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and Experience. *ACM Comput. Surv.*, 41(4), 2009.

[186] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE 2006)*, pages 371–380. ACM, 2006.

Part II

Original Publications

Paper I

Formal Development and Assessment of a Reconfigurable On-board Satellite System

Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, Timo Latvala and Laura Nummila

Originally published in: Frank Ortmeier, Peter Daniel (Eds.), *Proceedings of 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2012)*, LNCS 7612, 210–222, Springer, 2012.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-33678-2_18

Formal Development and Assessment of a Reconfigurable On-board Satellite System

Anton Tarasyuk[1,2], Inna Pereverzeva[1,2], Elena Troubitsyna[1], Timo Latvala[3], and Laura Nummila[3]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] Space Systems Finland, Espoo, Finland
{inna.pereverzeva, anton.tarasyuk, elena.troubitsyna}@abo.fi
{timo.latvala, laura.nummila}@ssf.fi

Abstract. Ensuring fault tolerance of satellite systems is critical for achieving goals of the space mission. Since the use of redundancy is restricted by the size and the weight of the on-board equipments, the designers need to rely on dynamic reconfiguration in case of failures of some components. In this paper we propose a formal approach to development of dynamically reconfigurable systems in Event-B. Our approach allows us to build the system that can discover possible reconfiguration strategy and continue to provide its services despite failures of its vital components. We integrate probabilistic verification to evaluate reconfiguration alternatives. Our approach is illustrated by a case study from aerospace domain.

Keywords: Formal modelling, fault tolerance, Event-B, refinement, probabilistic verification.

1 Introduction

Fault tolerance is an important characteristics of on-board satellite systems. One of the essential means to achieve it is redundancy. However, the use of (hardware) component redundancy in spacecraft is restricted by the weight and volume constraints. Thus, the system developers need to perform a careful cost-benefit analysis to minimise the use of spare modules yet achieve the required level of reliability.

Despite such an analysis, Space System Finland has recently experienced a double-failure problem with a system that samples and packages scientific data in one of the operating satellites. The system consists of two identical modules. When one of the first module subcomponents failed, the system switched to the use of the second module. However, after a while a subcomponent of the spare has also failed, so it became impossible to produce scientific data. To not lose the entire mission, the company has invented a solution that relied on healthy subcomponents of both modules and a complex communication mechanism to restore system functioning. Obviously, a certain amount of data has been lost before a repair was deployed. This motivated our work on exploring proactive

solutions for fault tolerance, i.e., planning and evaluating of scenarios implementing a seamless reconfiguration using a fine-grained redundancy.

In this paper we propose a formal approach to modelling and assessment of on-board reconfigurable systems. We generalise the ad-hoc solution created by Space Systems Finland and propose an approach to formal development and assessment of fault tolerant satellite systems. The essence of our modelling approach is to start from abstract modelling functional goals that the system should achieve to remain operational, and to derive reconfigurable architecture by refinement in the Event-B formalism [1]. The rigorous refinement process allows us to establish the precise relationships between component failures and goal reachability. The derived system architecture should not only satisfy functional requirements but also achieve its reliability objective. Moreover, since the reconfiguration procedure requires additional inter-component communication, the developers should also verify that system performance remains acceptable. Quantitative evaluation of reliability and performance of probabilistically augmented Event-B models is performed using the PRISM model checker [8].

The main novelty of our work is in proposing an integrated approach to formal derivation of reconfigurable system architectures and probabilistic assessment of their reliability and performance. We believe that the proposed approach facilitates early exploration of the design space and helps to build redundancy-frugal systems that meet the desired reliability and performance requirements.

2 Reconfigurable Fault Tolerant Systems

2.1 Case Study: Data Processing Unit

As mentioned in the previous section, our work is inspired by a solution proposed to circumvent the double failure occurred in a currently operational on-board satellite system. The architecture of that system is similar to Data Processing Unit (DPU) – a subsystem of the European Space Agency (ESA) mission BepiColombo [2]. Space Systems Finland is one of the providers for BepiColombo. The main goal of the mission is to carry out various scientific measures to explore the planet Mercury. DPU is an important part of the Mercury Planetary Orbiter. It consists of four independent components (computers) responsible for receiving and processing data from four sensor units: SIXS-X (X-ray spectrometer), SIXS-P (particle spectrometer), MIXS-T (telescope) and MIXS-C (collimator).

The behaviour of DPU is managed by telecommands (TCs) received from the spacecraft and stored in a circular buffer (TC pool). With a predefined rate, DPU periodically polls the buffer, decodes a TC and performs the required actions. Processing of each TC results in producing telemetry (TM). Both TC and TM packages follow the syntax defined by the ESA Packet Utilisation Standard [12]. As a result of TC decoding, DPU might produce a housekeeping report, switch to some mode or initiate/continue production of *scientific data*. The main purpose of DPU is to ensure a required rate of producing TM containing scientific data. In this paper we focus on analysing this particular aspect of the system behaviour. Hence, in the rest of the paper, TC will correspond to the telecommands requiring production of scientific data, while TM will designate packages containing scientific data.

2.2 Goal-Oriented Reasoning about Fault Tolerance

We use the notion of a goal as a basis for reasoning about fault tolerance. Goals – the functional and non-functional objectives that the system should achieve – are often used to structure the requirements of dependable systems [7, 9].

Let $\mathcal{G}$ be a predicate that defines a desired goal and $\mathcal{M}$ be a system model. Ideally, the system design should ensure that the goal can be reached "infinitely often". Hence, while verifying the system, we should establish that

$$\mathcal{M} \models \Box\Diamond\,\mathcal{G}.$$

The main idea of a goal-oriented development is to decompose the high-level system goals into a set of subgoals. Essentially, subgoals define the intermediate stages of achieving a high-level goal. In the process of goal decomposition we associate system components with tasks – the lowest-level subgoals. A component is associated with a task if its functionality enables establishing the goal defined by the corresponding task.

For instance, in this paper we consider *"produce scientific TM"* as a goal of DPU. DPU sequentially enquires each of its four components to produce its part of scientific data. Each component acquires fresh scientific data from the corresponding sensor unit (SIXS-X, SIXS-P, MIXS-T or MIXS-C), preprocesses it and makes available to DPU that eventually forms the entire TM package. Thus, the goal can be decomposed into four similar tasks *"sensor data production"*.

Generally, the goal $\mathcal{G}$ can be decomposed into a finite set of tasks:

$$\mathcal{T} = \{task_j \mid j \in 1..n \wedge n \in \mathbb{N}_1\},$$

Let also $\mathcal{C}$ be a finite set of components capable of performing tasks from $\mathcal{T}$:

$$\mathcal{C} = \{comp_j \mid j \in 1..m \wedge m \in \mathbb{N}_1\},$$

where $\mathbb{N}_1$ is the set of positive integers. Then the relation Φ defined below associates components with the tasks:

$$\Phi \in \mathcal{T} \leftrightarrow \mathcal{C}, \;\; \text{such that} \;\; \forall t \in \mathcal{T} \cdot \exists c \in \mathcal{C} \cdot \Phi(t, c),$$

where $\leftrightarrow$ designates a binary relation.

To reason about fault tolerance, we should take into account component unreliability. A failure of a component means that it cannot perform its associated task. Fault tolerance mechanisms employed to mitigate results of component failures rely on various forms of component redundancy. Spacecraft have stringent limitations on the size and weight of the on-board equipment, hence high degree of redundancy is rarely present. Typically, components are either duplicated or triplicated. Let us consider a duplicated system that consists of two identical DPUs – DPU_A and DPU_B. As it was explained above, each DPU contains four components responsible for controlling the corresponding sensor.

Traditionally, satellite systems are designed to implement the following simple redundancy scheme. Initially DPU_A is active, while DPU_B is a cold spare. DPU_A allocates tasks on its components to achieve the system goal $\mathcal{G}$ – processing of a TC and producing the TM. When some component of DPU_A fails, DPU_B is activated to achieve the goal $\mathcal{G}$. Failure of DPU_B results in failure of

the overall system. However, even though none of the DPUs can accomplish $\mathcal{G}$ on its own, it might be the case that the operational components of both DPUs can together perform the entire set of tasks required to reach $\mathcal{G}$. This observation allows us to define the following dynamic reconfiguration strategy.

Initially DPU_A is active and assigned to reach the goal $\mathcal{G}$. If some of its components fails, resulting in a failure to execute one of four scientific tasks (let it be $task_j$), the spare DPU_B is activated and DPU_A is deactivated. DPU_B performs the $task_j$ and the consecutive tasks required to reach $\mathcal{G}$. It becomes fully responsible for achieving the goal $\mathcal{G}$ until some of its component fails. In this case, to remain operational, the system performs *dynamic reconfiguration*. Specifically, it reactivates DPU_A and tries to assign the failed task to its corresponding component. If such a component is operational then DPU_A continues to execute the subsequent tasks until it encounters a failed component. Then the control is passed to DPU_B again. Obviously, the overall system stays operational until two identical components of both DPUs have failed.

We generalise the architecture of DPU by stating that essentially a system consists of a number of modules and each module consists of n components:

$$\mathcal{C} = \mathcal{C}_a \cup \mathcal{C}_b, \quad \text{where} \quad \mathcal{C}_a = \{a\_comp_j \mid j \in 1..n \wedge n \in \mathbb{N}_1\} \quad \text{etc.}$$

Each module relies on its components to achieve the tasks required to accomplish $\mathcal{G}$. An introduction of redundancy allows us to associate not a single but several components with each task. We reformulate the goal reachability property as follows: a goal remains reachable while there exists at least one *operational* component associated with each task. Formally, it can be specified as:

$$\mathcal{M} \models \Box \mathcal{O}_s, \quad \text{where} \quad \mathcal{O}_s \equiv \forall t \in \mathcal{T} \cdot (\exists c \in \mathcal{C} \cdot \Phi(t, c) \wedge \mathcal{O}(c))$$

and $\mathcal{O}$ is a predicate over the set of components $\mathcal{C}$ such that $\mathcal{O}(c)$ evaluates to $TRUE$ if and only if the component c is operational.

2.3 Probabilistic Assessment

If a duplicated system with the dynamic reconfiguration achieves the desired reliability level, it might allow the designers to avoid module triplication. However, it also increases the amount of intercomponent communication that leads to decreasing the system performance. Hence, while deciding on a fault tolerance strategy, it is important to consider not only reachability of functional goals but also their performance and reliability aspects.

In engineering, reliability is usually measured by the probability that the system remains operational under given conditions for a certain time interval. In terms of goal reachability, the system remains operational until it is capable of reaching targeted goals. Hence, to guarantee that system is capable of performing a required functions within a time interval t, it is enough to verify that

$$\mathcal{M} \models \Box^{\leq t} \mathcal{O}_s. \tag{1}$$

However, due to possible component failures we usually cannot guarantee the absolute preservation of (1). Instead, to assess the reliability of a system, we need to show that the probability of preserving the property (1) is sufficiently high.

On the other hand, the system performance is a reward-based property that can be measured by the number of successfully achieved goals within a certain time period.

To quantitatively verify these quality attributes we formulate the following CSL (Continuous Stochastic Logic) formulas [6]:

$$\mathbf{P}_{=?}\{\mathbf{G} \leq t\ \mathcal{O}_s\} \quad \text{and} \quad \mathbf{R}(|goals|)_{=?}\{\mathbf{C} \leq t\}.$$

The formulas above are specified using PRISM notation. The operator **P** is used to refer to the probability of an event occurrence, **G** is an analogue of $\Box$, **R** is used to analyse the *expected values* of rewards specified in a model, while **C** specifies that the reward should be cumulated only up to a given time bound. Thus, the first formula is used to analyse how likely the system remains operational as time passes, while the second one is used to compute the expected number of achieved goals cumulated by the system over t time units.

In this paper we rely on modelling in Event-B to formally define the architecture of a dynamically reconfigurable system, and on the probabilistic extension of Event-B to create models for assessing system reliability and performance. The next section briefly describes Event-B and its probabilistic extension.

3 Modelling in Event-B and Probabilistic Analysis

3.1 Modelling and Refinement in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [1], which encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the behaviour of a modelled system. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms. The model variables are strongly typed by the constraining predicates. These predicates and the other important properties that must be preserved by the model constitute model *invariants*.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of the event is defined by the next-state relation R_e. In Event-B, R_e is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. The guard defines the conditions under which the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce nondeterminism and introduce detailed design

decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we should define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables. The proof of data refinement is often supported by supplying *witnesses* – the concrete values for the replaced abstract variables and parameters. Witnesses are specified in the event clause **with**.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations generated by the Rodin platform [11]. The platform provides an automated tool support for proving.

3.2 Augmenting Event-B Models with Probabilities

Next we briefly describe the idea behind translating of an Event-B machine into continuous time Markov chain – CTMC (the details can be found in [15]). To achieve this, we augment all events of the machine with information about the probability and duration of all the actions that may occur during their execution, and refine them by their probabilistic counterparts.

Let Σ be a state space of an Event-B model defined by all possible values of the system variables. Let also $\mathcal{I}$ be the model invariant. We consider an event e as a binary relation on Σ, i.e., for any two states $\sigma, \sigma' \in \Sigma$:

$$e(\sigma, \sigma') \stackrel{def}{=} G_e(\sigma) \wedge R_e(\sigma, \sigma').$$

Definition 1. *The behaviour of an Event-B machine is fully defined by a transition relation* $\rightarrow$*:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \rightarrow \sigma'},$$

where $\mathsf{before}(e) = \{\sigma \in \Sigma \mid \mathcal{I}(\sigma) \wedge G_e(\sigma)\}$, $\mathcal{E}_\sigma = \{e \in \mathcal{E} \mid \sigma \in \mathsf{before}(e)\}$ *and* $\mathsf{after}(e) = \{\sigma' \in \Sigma \mid \mathcal{I}(\sigma') \wedge (\exists \sigma \in \Sigma \cdot \mathcal{I}(\sigma) \wedge G_e(\sigma) \wedge R_e(\sigma, \sigma'))\}$.

Furthermore, let us denote by $\lambda_e(\sigma, \sigma')$ the (exponential) transition rate from σ to σ' via the event e, where $\sigma \in \mathsf{before}(e)$ and $R_e(\sigma, \sigma')$. By augmenting all the event actions with transition rates, we can modify Definition 1 as follows.

Definition 2. *The behaviour of a probabilistically augmented Event-B machine is defined by a transition relation* $\xrightarrow{\Lambda}$*:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \xrightarrow{\Lambda} \sigma'}, \quad \textit{where } \Lambda = \sum_{e \in \mathcal{E}_\sigma} \lambda_e(\sigma, \sigma').$$

Definition 2 allows us to define the semantics of a probabilistically augmented Event-B model as a probabilistic transition system with the state space Σ, transition relation $\xrightarrow{\Lambda}$ and the initial state defined by model initialisation (for probabilistic models we require the initialisation to be deterministic). Clearly, such a transition system corresponds to a CTMC.

In the next section we demonstrate how to formally derive an Event-B model of the architecture of a reconfigurable system.

4 Deriving Fault Tolerant Architectures by Refinement in Event-B

The general idea behind our formal development is to start from an abstract goal modelling, decompose it into tasks and introduce an abstract representation of the goal execution flow. Such a model can be refined into different fault tolerant architectures. Subsequently, these models are augmented with probabilistic data and used for the quantitative assessment.

4.1 Modelling Goal Reaching

Goal Modelling. Our initial specification abstractly models the process of reaching the goal. The progress of achieving the goal is modelled by the variable *goal* that obtains values from the enumerated set $STATUS = \{not\_reached, reached, failed\}$. Initially, the system is not assigned any goals to accomplish, i.e., the variable *idle* is equal to $TRUE$. When the system becomes engaged in establishing the goal, *idle* obtains value $FALSE$ as modelled by the event *Activation*. In the process of accomplishing the goal, the variable *goal* might eventually change its value from *not_reached* to *reached* or $failed$, as modelled by the event *Body*. After the goal is reached the system becomes idle, i.e., a new goal can be assigned. The event $Finish$ defines such a behaviour. We treat the failure to achieve the goal as a permanent system failure. It is represented by the infinite stuttering defined in the event *Abort*.

Activation $\widehat{=}$
when $idle = TRUE$
then $idle := FALSE$
end

Body $\widehat{=}$
when $idle = FALSE \wedge goal = not\_reached$
then $goal :\in STATUS$
end

Finish $\widehat{=}$
when $idle = FALSE \wedge goal = reached$
then $goal, idle := not\_reached, TRUE$
end

Abort $\widehat{=}$
when $goal = failed$
then $skip$
end

Goal Decomposition. The aim of our first refinement step is to define the goal execution flow. We assume that the goal is decomposed into n tasks, and can be achieved by a sequential execution of one task after another. We also assume that the id of each task is defined by its execution order. Initially, when the goal is assigned, none of the tasks is executed, i.e., the state of each task is "not defined" (designated by the constant value ND). After the execution, the state of a task might be changed to success or failure, represented by the constants OK and NOK correspondingly. Our refinement step is essentially data refinement that replaces the abstract variable *goal* with the new variable *task* that maps the id of a task to its state, i.e., $task \in 1..n \rightarrow \{OK, NOK, ND\}$.

We omit showing the events of the refined model (the complete development can be found in [13]). They represent the process of sequential selection of one task after another until either all tasks are executed, i.e., the goal is reached, or execution of some task fails, i.e., goal is not achieved. Correspondingly, the guards ensure that either the goal reaching has not commenced yet or the execution of all previous task has been successful. The body of the events nondeterministically

changes the state of the chosen task to OK or NOK. The following invariants define the properties of the task execution flow:

$$\forall l \cdot l \in 2 \mathinner{..} n \wedge task(l) \neq ND \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} l-1 \Rightarrow task(i) = OK),$$
$$\forall l \cdot l \in 1 \mathinner{..} n-1 \wedge task(l) \neq OK \Rightarrow (\forall i \cdot i \in l+1 \mathinner{..} n \Rightarrow task(i) = ND).$$

They state that the goal execution can progress, i.e., a next task can be chosen for execution, only if none of the previously executed tasks failed and the subsequent tasks have not been executed yet.

From the requirements perspective, the refined model should guarantee that the system level goal remains achievable. This is ensured by the gluing invariants that establish the relationship between the abstract goal and the tasks:

$$task[1 \mathinner{..} n] = \{OK\} \Rightarrow goal = reached,$$
$$(task[1 \mathinner{..} n] = \{OK, ND\} \vee task[1 \mathinner{..} n] = \{ND\}) \Rightarrow goal = not\_reached,$$
$$(\exists i \cdot i \in 1 \mathinner{..} n \wedge task(i) = NOK) \Rightarrow goal = failed.$$

Introducing Abstract Communication. In the second refinement step we introduce an abstract model of communication. We define a new variable ct that stores the id of the last achieved task. The value of ct is checked every time when a new task is to be chosen for execution. If task execution succeeds then ct is incremented. Failure to execute the task leaves ct unchanged and results only in the change of the failed task status to NOK. Essentially, the refined model introduces an abstract communication via shared memory. The following gluing invariants allow us to prove the refinement:

$$ct > 0 \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} ct \Rightarrow task(i) = OK), \quad ct < n \Rightarrow task(ct+1) \in \{ND, NOK\},$$
$$ct < n-1 \Rightarrow (\forall i \cdot i \in ct+2 \mathinner{..} n \Rightarrow task(i) = ND).$$

As discussed in Section 2, each task is independently executed by a separate component of a high-level module. Hence, by substituting the id of a task with the id of the corresponding component, i.e., performing a data refinement with the gluing invariant

$$\forall i \in 1..n \cdot task(i) = comp(i),$$

we specify a *non-redundant* system architecture. This invariant trivially defines the relation Φ. Next we demonstrate how to introduce either a triplicated architecture or duplicated architecture with a dynamic reconfiguration by refinement.

4.2 Reconfiguration Strategies

To define triplicated architecture with static reconfiguration, we define three identical modules A, B and C. Each module consists of n components executing corresponding tasks. We refine the abstract variable $task$ by the three new variables a_comp, b_comp and c_comp:

$$a\_comp \in 1..n \rightarrow STATE,\ b\_comp \in 1..n \rightarrow STATE,\ c\_comp \in 1..n \rightarrow STATE.$$

To associate the tasks with the components of each module, we formulate a number of gluing invariants that essentially specify the relation Φ. Some of these invariants are shown below:

$$\forall i \cdot i \in 1 \mathinner{..} n \wedge module = A \wedge a\_comp(i) = OK \Rightarrow task(i) = OK,$$
$$module = A \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow b\_comp(i) = ND \wedge c\_comp(i) = ND),$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge module = A \wedge a\_comp(i) \neq OK \Rightarrow task(i) = ND,$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge module = B \wedge b\_comp(i) \neq OK \Rightarrow task(i) = ND,$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge module = C \Rightarrow c\_comp(i) = task(i),$$
$$module = B \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow c\_comp(i) = ND).$$

Here, a new variable $module \in \{A, B, C\}$ stores the id of the currently active module. The complete list of invariants can be found in [13]. Please note, that these invariants allows us to mathematically prove that the Event-B model preserves the desired system architecture.

An alternative way to perform this refinement step is to introduce a duplicated architecture with dynamic reconfiguration. In this case, we assume that our system consists of two modules, A and B, defined in the same way as discussed above. We replace the abstract variable $task$ with two new variables a_comp and b_comp. Below we give an excerpt from the definition of the gluing invariants:

$$module = A \wedge ct > 0 \wedge a\_comp(ct) = OK \Rightarrow task(ct) = OK,$$
$$module = B \wedge ct > 0 \wedge b\_comp(ct) = OK \Rightarrow task(ct) = OK,$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge a\_comp(i) = NOK \wedge b\_comp(i) = NOK \Rightarrow task(i) = NOK,$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge a\_comp(i) = NOK \wedge b\_comp(i) = ND \Rightarrow task(i) = ND,$$
$$\forall i \cdot i \in 1 \mathinner{..} n \wedge b\_comp(i) = NOK \wedge a\_comp(i) = ND \Rightarrow task(i) = ND.$$

Essentially, the invariants define the behavioural patterns for executing the tasks according to dynamic reconfiguration scenario described in Section 2.

Since our goal is to study the fault tolerance aspect of the system architecture, in our Event-B model we have deliberately abstracted away from the representation of the details of the system behaviour. A significant number of functional requirements is formulated as gluing invariants. As a result, to verify correctness of the models we discharged more than 500 proof obligations. Around 90% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment.

Note that the described development for a generic system can be easily instantiated to formally derive fault tolerant architectures of DPU. The goal of DPU – handling the scientific TC by producing TM – is decomposed into four tasks that define the production of data by the satellite's sensor units – SIXS-X, SIXS-P, MIXS-T and MIXS-C. Thus, for such a model we have four tasks ($n = 4$) and each task is handled by the corresponding computing component of DPU. The high-level modules A, B and C correspond to three identical DPUs that control handling of scientific TC – DPU_A, DPU_B and DPU_C, while functions a_comp, b_comp and c_comp represent statuses of their internal components.

From the functional point of view, both alternatives of the last refinement step are equivalent. Indeed, each of them models the process of reaching the goal by a fault tolerant system architecture. In the next section we will present a quantitative assessment of their reliability and performance aspects.

5 Quantitative Assessment of Reconfiguration Strategies

The scientific mission of BepiColombo on the orbit of the Mercury will last for one year with possibility to extend this period for another year. Therefore, we should assess the reliability of both architectural alternatives for this period of time. Clearly, the triplicated DPU is able to tolerate up to three DPU failures within the two-year period, while the use of a duplicated DPU with a dynamic reconfiguration allows the satellite to tolerate from one (in the worst case) to four (in the best case) failures of the components.

Obviously, the duplicated architecture with a dynamic configuration minimises volume and the weight of the on-board equipment. However, the dynamic reconfiguration requires additional inter-component communication that slows down the process of producing TM. Therefore, we need to carefully analyse the performance aspect as well. Essentially, we need to show that the duplicated system with the dynamic reconfiguration can also provide a sufficient amount of scientific TM within the two-year period.

To perform the probabilistic assessment of reliability and performance, we rely on two types of data:

- probabilistic data about lengths of time delays required by DPU components and sensor units to produce the corresponding parts of scientific data
- data about occurrence rates of possible failures of these components

It is assumed that all time delays are exponentially distributed. We refine the Event-B specifications obtained at the final refinement step by their probabilistic counterparts. This is achieved via introducing probabilistic information into events and replacing all the local nondeterminism with the (exponential) race conditions. Such a refinement relies on the model transformation presented in Section 3. As a result, we represent the behaviour of Event-B machines by CTMCs. This allows us to use the probabilistic symbolic model checker PRISM to evaluate reliability and performance of the proposed models.

Due to the space constraints, we omit showing the PRISM specifications in the paper, they can be found in [13]. The guidelines for Event-B to PRISM model transformation can be found in our previous work [14].

The results of quantitative verification performed by PRISM show that with probabilistic characteristics of DPU presented, in Table 1[1], both reconfiguration strategies lead to a similar level of system reliability and performance with insignificant advantage of the triplicated DPU. Thus, the reliability levels of both systems within the two-year period are approximately the same with the difference of just 0.003 at the end of this period (0.999 against 0.996). Furthermore, the use of two DPUs under dynamic reconfiguration allows the satellite to handle only 2 TCs less after two years of work – 1104 against 1106 returned TM packets in the case of the triplicated DPU. Clearly, the use of the duplicated architecture with dynamic reconfiguration to achieve the desired levels of relia bility and performance is optimal for the considered system.

[1] Provided information may differ form the characteristics of the real components. It is used merely to demonstrate how the required comparison of reliability/performance can be achieved

Table 1. Rates (time is measured by minutes)

TC access rate when the system is idle	λ	$\frac{1}{12 \cdot 60}$	SIXS-P work rate	α_2	$\frac{1}{30}$
TM output rate when a TC is handled	μ	$\frac{1}{20}$	SIXS-P failure rate	β_2	$\frac{1}{10^6}$
Spare DPU activation rate (power on)	δ	$\frac{1}{10}$	MIXS-T work rate	α_3	$\frac{1}{30}$
DPUs "communication" rate	τ	$\frac{1}{5}$	MIXS-T failure rate	β_3	$\frac{1}{9 \cdot 10^7}$
SIXS-X work rate	α_1	$\frac{1}{60}$	MIXS-C work rate	α_4	$\frac{1}{90}$
SIXS-X failure rate	β_1	$\frac{1}{8 \cdot 10^7}$	MIXS-C failure rate	β_4	$\frac{1}{6 \cdot 10^7}$

Finally, let us remark that the goal-oriented style of the reliability and performance analysis has significantly simplified the assessment of the architectural alternatives of DPU. Indeed, it allowed us to abstract away from the configuration of input and output buffers, i.e., to avoid modelling of the circular buffer as a part of the analysis.

6 Conclusions and Related Work

In this paper we proposed a formal approach to development and assessment of fault tolerant satellite systems. We made two main technical contributions. On the one hand, we defined the guidelines for development of the dynamically reconfigurable systems. On the other hand, we demonstrated how to formally assess reconfiguration strategy and evaluate whether the chosen fault tolerance mechanism fulfils reliability and performance objectives. The proposed approach was illustrated by a case study – development and assessment of the reconfigurable DPU. We believe that our approach not only guarantees correct design of complex fault tolerance mechanisms but also facilitates finding suitable trade-offs between reliability and performance.

A large variety of aspects of the dynamic reconfiguration has been studied in the last decade. For instance, Wermelinger et al. [17] proposed a high-level language for specifying the dynamically reconfigurable architectures. They focus on modifications of the architectural components and model reconfiguration by the algebraic graph rewriting. In contrast, we focused on the functional rather than structural aspect of reasoning about reconfiguration.

Significant research efforts are invested in finding suitable models of triggers for run-time adaptation. Such triggers monitor performance [3] or integrity [16] of the application and initiate reconfiguration when the desired characteristics are not achieved. In our work we perform the assessment of reconfiguration strategy at the development phase that allows us to rely on existing error detection mechanisms to trigger dynamic reconfiguration.

A number of researchers investigate self* techniques for designing adaptive systems that autonomously achieve fault tolerance, e.g., see [4, 10]. However, these approaches are characterised by a high degree of uncertainty in achieving fault tolerance that is unsuitable for the satellite systems. The work [5] proposes an interesting conceptual framework for establishing a link between changing environmental conditions, requirements and system-level goals. In our approach we were more interested in studying a formal aspect of dynamic reconfiguration.

In our future work we are planning to further study the properties of dynamic reconfiguration. It particular, it would be interesting to investigate reconfiguration in the presence of parallelism and complex component interdependencies.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. BepiColombo: ESA Media Center, Space Science, online at http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html
3. Caporuscio, M., Di Marco, A., Inverardi, P.: Model-Based System Reconfiguration for Dynamic Performance Management. J. Syst. Softw. 80, 455–473 (2007)
4. de Castro Guerra, P.A., Rubira, C.M.F., de Lemos, R.: A Fault-Tolerant Software Architecture for Component-Based Systems. In: Architecting Dependable Systems. pp. 129–143. Springer (2003)
5. Goldsby, H., Sawyer, P., Bencomo, N., Cheng, B., Hughes., D.: Goal-Based Modeling of Dynamically Adaptive System Requirements. In: ECBS 2008. pp. 36–45. IEEE Computer Society (2008)
6. Grunske, L.: Specification Patterns for Probabilistic Quality Properties. In: ICSE 2008. pp. 31–40. ACM (2008)
7. Kelly, T.P., Weaver, R.A.: The Goal Structuring Notation – A Safety Argument Notation. In: DSN 2004, Workshop on Assurance Cases (2004)
8. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV'11. pp. 585–591. Springer (2011)
9. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: RE'01. pp. 249–263. IEEE Computer Society (2001)
10. de Lemos, R., de Castro Guerra, P.A., Rubira, C.M.F.: A Fault-Tolerant Architectural Approach for Dependable Systems. Software, IEEE 23, 80–87 (2006)
11. Rodin: Event-B Platform, online at http://www.event-b.org/
12. Space Engineering: Ground Systems and Operations – Telemetry and Telecommand Packet Utilization: ECSS-E-70-41A. ECSS Secretariat, 30.01.2003, online at http://www.ecss.nl/
13. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T., Nummila, L.: Formal Development and Assessment of a Reconfigurable On-board Satellite System. Tech. Rep. 1038, Turku Centre for Computer Science (2012)
14. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Quantitative Reasoning about Dependability in Event-B: Probabilistic Model Checking Approach. In: Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 459–472. IGI Global (2011)
15. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In: IFM 2012. pp. 237–252. Springer (2012)
16. Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T.: An Automated Formal Approach to Managing Dynamic Reconfiguration. In: ASE 2006. pp. 18–22. Springer (2006)
17. Wermelinger, M., Lopes, A., Fiadeiro, J.: A Graph Based Architectural Reconfiguration Language. SIGSOFT Softw. Eng. Notes 26, 21–32 (2001)

Paper II

Formal Derivation of Distributed MapReduce

Inna Pereverzeva, Michael J. Butler, Asieh Salehi Fathabadi, Linas Laibinis, Elena Troubitsyna

Originally published in: Yamine Aït Ameur, Klaus-Dieter Schewe (Eds.), *Proceedings of 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, LNCS 8477, 238–254, Springer-Verlag Berlin Heidelberg, 2014.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-662-43652-3_21

Formal Derivation of Distributed MapReduce

Inna Pereverzeva[1,2], Michael Butler[3], Asieh Salehi Fathabadi[3], Linas Laibinis[1], and Elena Troubitsyna[1]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] University of Southampton, UK
{inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi
{mjb, asf08r}@ecs.soton.ac.uk

Abstract. MapReduce is a powerful distributed data processing model that is currently adopted in a wide range of domains to efficiently handle large volumes of data, i.e., cope with the big data surge. In this paper, we propose an approach to formal derivation of the MapReduce framework. Our approach relies on stepwise refinement in Event-B and, in particular, the *event refinement structure* approach – a diagrammatic notation facilitating formal development. Our approach allows us to derive the system architecture in a systematic and well-structured way. The main principle of MapReduce is to parallelise processing of data by first mapping them to multiple processing nodes and then merging the results. To facilitate this, we formally define interdependencies between the map and reduce stages of MapReduce. This formalisation allows us to propose an alternative architectural solution that weakens blocking between the stages and, as a result, achieves a higher degree of parallelisation of MapReduce computations.

Keywords: formal modelling, Event-B, refinement, event refinement structure, MapReduce

1 Introduction

MapReduce is a widely used framework for handling large volumes of data [5]. It allows the users to automatically parallelise computations and execute them on large clusters of computers. Essentially, the computation is performed in two stages – map and reduce. The first stage maps the input data to multiple processing nodes, while the second stage performs parallel computations to merge the obtained results. Typically, execution of the map stage is blocking, i.e., execution of the reduce stage does not start until the map stage is completed. Though MapReduce is already a highly performant framework, to keep pace with the drastically increasing volume of data, it would be desirable to loosen the coupling between the stages and hence exploit the potential for parallelisation to the fullest.

In this paper, we undertake a formal study of the MapReduce framework. We formally model the control flow and data interdependencies between the map and

reduce tasks, as well as derive the conditions under which the execution of the reduce stage can overlap with the execution of the map stage. Our formalisation of the (generic) MapReduce framework relies on the Event-B method and the associated Rodin platform. Event-B [1] is a formal approach that is particularly suitable for the development of distributed systems. The system development in Event-B starts from an abstract specification that is transformed into a detailed specification in a number of correctness-preserving refinement steps. In this paper, the Event Refinement Structure approach [3, 6] is used to facilitate the refinement process. The technique provides us with an explicit graphical representation of the relationships between the events at different levels of abstraction and helps to gradually derive the complex MapReduce architecture.

Event-B relies on proof-based verification that is integrated into the development process. The Rodin platform [10] automates development in Event-B by generating the required proof obligations and automatically discharging a part of them. Via abstraction, proof and decomposition, Event-B enables reasoning about system-level properties of complex distributed systems. In particular, it allows us to explicitly define interdependencies between the processed data and derive the conditions under which an execution of the reduce stage can start before completion of the map stage. We believe that the proposed approach provides the designers with a formally grounded insight on the properties of MapReduce and enables fine-tuning of the framework to achieve a higher degree of parallelisation.

The rest of the paper is organised as follows. In Section 2 we describe the generic MapReduce framework and our formalisation of it. In Section 3 we give an overview of the Event-B formalism and the Event Refinement Structure (ERS) approach. In Section 4 we present our formal derivation of the MapReduce framework in Event-B using the ERS approach. As a result, we derive two alternative architectures of the MapReduce framework – blocking and partially blocking. In Section 5 we overview the related work and present some concluding remarks.

2 MapReduce

2.1 Overview of MapReduce

MapReduce is a programming model for processing large data sets. It has been originally proposed by Google [5]. The framework is designed to orchestrate the work on distributed nodes, run various computational tasks in parallel, providing at the same time for redundancy and fault tolerance. Distributed and parallelised computations are the key mechanisms that make the MapReduce framework very attractive to use in a wide range of application areas: data mining, bioinformatics, business intelligence, etc. Nowadays it is becoming increasingly popular in cloud computing. There exist different implementations of MapReduce, among them open-source Hadoop [2], Hive [11], and others.

The MapReduce computational model was inspired by the *map* and *reduce* functions widely used in functional programming. A MapReduce computation is composed of two main steps: the *map stage* and the *reduce stage*. During the map stage, the system inputs are divided into smaller computational tasks,

which are then performed in parallel (provided there are enough processors in the cluster). The obtained collective results then become the inputs for the reduce stage, which combines them in some way to produce the overall output. Once again, the reduce inputs are split into smaller computational tasks that can be executed in parallel.

The MapReduce framework can be tuned to perform different data transformations by the user-supplied map and reduce functions. These functions encode basic mapping and reduction tasks to be performed in single nodes. The MapReduce framework then incorporates the provided functions and orchestrates the overall distributed computations based on them.

A typical example illustrating MapReduce computations is counting the word occurrences in a large set of documents. The input data set is split into smaller portions and the user-provided map function is applied to each such data block. The *map* function simply assigns to each word it encounters the value equal to 1. Overall, the map stage produces a collection of (word,1) pairs as intermediate results. Then, during the reduce stage, the user-supplied reduce function takes a portion of these intermediate data related to a particular word and sums all the occurrences of that word. Such a computation is done for each encountered word. The overall result is a set of (word,number) pairs.

2.2 Towards Formal Reasoning about MapReduce

In this section, we present a formalisation of the MapReduce framework. Specifically, we mathematically represent all MapReduce execution stages, i.e., the required data and control flow, and identify the computational (map and reduce) tasks that can be executed in parallel. Moreover, we formally define possible data interdependencies between the map and reduce tasks. The latter allows us to propose an alternative architectural solution, which weakens blocking between the MapReduce phases and, as a result, achieves a higher degree of parallelisation of MapReduce computations. In Section 4, we will propose two alternative formal developments of the MapReduce framework in Event-B, both of which rely on the formalisation presented below.

Let *IData* be an abstract type defining the input data to be processed within the MapReduce framework and *OData* be an abstract type defining the resulting output data. In a nutshell, a MapReduce computation processes the given input data and generates some result. Thus, it can be formally represented as a function:

$$MapReduce \in IData \rightarrow OData.$$

More specifically, a MapReduce computation can be defined as a functional composition of the following phases: *MSplit*, *Map*, *RSplit*, *Reduce*, and *Combine*:

$$MapReduce = MSplit;\ Map;\ RSplit;\ Reduce;\ Combine.$$

Let us note that the phases *MSplit* and *Map* together correspond to the *map stage* mentioned in Section 2.1, while the phases *RSplit* and *Reduce* belong to the *reduce stage*.

The MapReduce process starts with the *MSplit* phase. During this phase, the input data are split into a number of blocks (portions of the input data),

which can be handled independently of each other. In the following *Map* phase, the user-provided *map* function is applied to each such input block. Next, in the *RSplit* phase, the MapReduce framework groups together all the intermediate results obtained after the Map phase to prepare for the reduce computations. Similarly to the *MSplit* phase, the data are divided into blocks that can be handled separately. After that, the *Reduce* phase is executed, during which the user-supplied *reduce* function is repeatedly applied (once per each block). Finally, in the *Combine* phase, all the obtained results are combined into the final output.

Formalisation of the MapReduce execution phases. Next we define all the MapReduce execution phases in more detail. In the *MSplit* phase, the input data are split into a number of blocks that are later supplied to the *map* function. To emphasise the independent nature of map computations, we associate the notion of a *map task* with such a portion of the input data to be processed separately.

Let *MTask* be a set of all possible map tasks and *MData* be an abstract type defining the data obtained after the splitting. Then the *MSplit* phase can be mathematically represented as follows:

$$MSplit \in IData \to (MTask \nrightarrow MData).$$

Essentially, *MSplit* produces a partitioning of the input data to be used in the *Map* phase among different map tasks. Note that the result of *MSplit* is a partial function since only a subset of *MTask* may be needed for particular input data.

We assume that the input data fully determines the number and the subset of involved map tasks.[4] To extract this information, we use the following functions

$$mtasks \in IData \to \mathbb{P}_1(MTask), \quad mnum \in IData \to \mathbb{N}_1$$

defined as

$$\forall idata \in IData \cdot\ mtasks(idata) = \mathsf{dom}(MSplit(idata)),$$
$$\forall idata \in IData \cdot\ mnum(idata) = \mathsf{card}(MSplit(idata)),$$

where dom and card are the function domain and set cardinality operators.

The *Map* phase involves transformation of all the data obtained by the *MSplit* phase into the intermediate form to be used in the later phases. Let $RData$ be an abstract type defining the intermediate data obtained after the *Map* phase. Then *Map* phase can be mathematically represented as the following function:

$$Map \in (MTask \nrightarrow MData) \to \mathbb{P}_1(MTask \times RData).$$

Therefore, *Map* takes the map data partitioning produced by *MSplit* and returns the transformed data associated with the map tasks that produced them. These results then become the input data for the following reduce computations.

In our formalisation the *Map* results consist of a set of (*mtask, rdata*) pairs, without assuming any further structure among them. This is done intentionally, since grouping and partitioning of these data will be performed in the *RSplit* phase.

[4] This applies only to the involved computational tasks. Actual software components that will be employed to carry out the necessary computations can be dynamically assigned and re-assigned for a specific map or reduce task.

All the involved map tasks should be performed within the *Map* phase. Formally, this requirement can be formulated as follows:

$$\forall f \in MTask \pfun MData \cdot f \neq \varnothing \Rightarrow \mathsf{dom}(f) = \mathsf{dom}(Map(f)).$$

Next the results obtained by the *Map* phase are grouped together to prepare for reduce computations. Similarly to the *MSplit* phase, they should be first partitioned among the individual *reduce tasks*.

Let *RTask* be a set of all possible reduce tasks. Then the *RSplit* phase can be formally defined as the following function:

$$RSplit \in \mathbb{P}_1(MTask \times RData) \to (RTask \pfun \mathbb{P}_1(RData)).$$

Essentially, the function takes the intermediate results produced by the *Map* phase and produces data partitioning among the involved reduce tasks.

We can reason about the actual number and the subset of the involved reduce tasks. Once again, this is determined by the original input data. Formally, we introduce the functions

$$rtasks \in IData \to \mathbb{P}_1(RTask), \quad rnum \in IData \to \mathbb{N}_1$$

defined as

$$\forall idata \in IData \cdot rtasks(idata) = \mathsf{dom}(RSplit(Map(MSplit(idata)))),$$
$$\forall idata \in IData \cdot rnum(idata) = \mathsf{card}(RSplit(Map(MSplit(idata)))).$$

The *RSplit* phase only rearranges the intermediate data, producing their partitioning among the reduce tasks. Therefore, neither new data should appear nor any of the existing data can disappear during this transformation. Mathematically, this can be formulated as the following property:

$$\forall f \in \mathbb{P}_1(MTask \times RData) \cdot \mathsf{ran}(f) = (\bigcup rt \in \mathsf{dom}(RSplit(f)) \mid RSplit(f)(rt)),$$

where ran is the function range operator.

The *Reduce* phase is similar to the *Map* phase – it takes as input a data partitioning produced by *RSplit* and returns transformed data:

$$Reduce \in (RTask \pfun \mathbb{P}_1(RData)) \to \mathbb{P}_1(OData),$$

where *OData* is an abstract type defining the resulting output data.

Finally, the last *Combine* phase can be simply defined as follows:

$$Combine \in \mathbb{P}_1(OData) \to OData.$$

Formalisation of the `map` and `reduce` functions. The *Map* phase is based on repeated invocations of the user-supplied function `map`. The `map` function can be formally represented in the following way:

$$\texttt{map} \in MData \to \mathbb{P}_1(RData).$$

Thus, it takes an input data from *MData* and produces some intermediate data to be used in reduce computations. The `map` function and the *Map* phase are tightly linked. To be precise, the union of all the results obtained from all the `map` function applications should be equal to the overall result of the *Map* phase:

$$Map = \{f \cdot f \in MTask \pfun MData \mid f \mapsto (\bigcup mt \cdot mt \in dom(f) \mid \{mt\} \times map(f(mt)))\}.$$

The user-supplied `reduce` function can be specified as follows:

$$\texttt{reduce} \in \mathbb{P}_1(RData) \rightarrow \mathbb{P}_1(OData).$$

It takes as an input a subset of the reduce data *RData* and produces some subset of output data from *OData*.

Finally, the overall result of the *Reduce* phase should be equal to the combined results obtained by repeated application of the `reduce` function:

$$Reduce = \{f \cdot f \in RTask \nrightarrow \mathbb{P}_1(RData) \mid f \mapsto (\bigcup rt \cdot rt \in dom(f) \mid \texttt{reduce}(f(rt)))\}.$$

Essentially, the *Reduce* definition is directly based on the user-supplied `reduce` function.

Formalisation of interdependencies between the map and reduce tasks. The main principle of MapReduce is that all the map and reduce computations are distributed to multiple independent processing nodes. The reduce inputs are based on the previously produced map outputs. However, in some cases, the reduce inputs might depend on only particular map outputs. Therefore, the reduce stage can be initiated before all the map computations are finished. To relax the limitation of the original MapReduce computation flow, requiring that the reduce stage starts only after completing the map stage, we formally define the *dependence relation* between the map and reduce tasks as the following function *dep*:

$$dep \in IData \rightarrow \mathbb{P}(RTask \times MTask),$$

with the following property:

$$\begin{aligned} \forall\, idata \in IData,\, rt \in RTask,\, mt \in MTask \cdot\ & rt \mapsto mt \in dep(idata) \quad \Leftrightarrow \\ & mt \in \mathsf{dom}(MSplit(idata)) \wedge \\ & (\exists rd \in RData \cdot rt \in \mathsf{dom}(RSplit(Map(MSplit(idata))))) \ \wedge \\ & rd \in RSplit(Map(MSplit(idata)))(rt) \ \wedge\ mt \mapsto rd \in Map(MSplit(idata))). \end{aligned}$$

The property states that for any input data *input*, a map task *mt* and a reduce task *rt* are in *dependence relation* (i.e., a reduce task depends on a map task), if and only if some intermediate data *rd* has been generated for this reduce task *rt* by the computations of the map task *mt* during the *Map* phase. Essentially, the relation *dep* defines the data interdependencies between the map and reduce stages. This formalisation allows us to propose (in Section 4) an alternative architectural solution that weakens blocking between the stages.

Finally, to make it possible for a particular reduce task to start immediately after all the necessary data have been produced by the map tasks related by *dep*, we need a version of *RSplit*, defining a partial split related with a specific reduce task. For a given reduce task, it produces the grouped together results obtained within the Map phase:

$$\texttt{rsplit} \ \in\ RTask \nrightarrow (\mathbb{P}_1(MTask \times RData) \rightarrow \mathbb{P}_1(RData)).$$

Again, the union of the results obtained from all the `rsplit` function applications should be the result of the *RSplit* phase:

$$\forall f \cdot f \in \mathbb{P}_1(MTask \times RData) \Rightarrow$$
$$RSplit(f) = (\bigcup rt \cdot rt \in dom(rsplit) | \{rt \mapsto \texttt{rsplit}(rt)(f)\}).$$

In Section 4 we will demonstrate that, by relying on the proposed formalisation, we can derive a formal model of the MapReduce framework. There we will propose two models of MapReduce – *blocking* and *partially blocking* models.

3 Formal Development by Refinement: Background

3.1 Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving [1]. In Event-B, a system model is specified using the notion of an *abstract state machine*. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that, together with other important system properties, are defined as model *invariants*. Usually, a machine has an accompanying component, called a *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$e \;\widehat{=}\; \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, a is the list of local variables, and (the event *guard*) G_e is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation R_e. The event guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. The consistency of Event-B models, i.e., verification of model well-formedness, invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform [10] provides an automated support for modelling and verification. In particular, it automatically generates the required proof obligations and attempts to discharge them.

3.2 Event Refinement Structure

The Event Refinement Structure (ERS) [3, 6] approach augments Event-B refinement with a graphical notation that allows us to explicitly represent the relationships between the events at different abstraction levels as well as define the required event sequence in a model. ERS is illustrated by example in Figure 1. The diagram explicitly shows that *AbstractEvent* is refined by *Event2*,

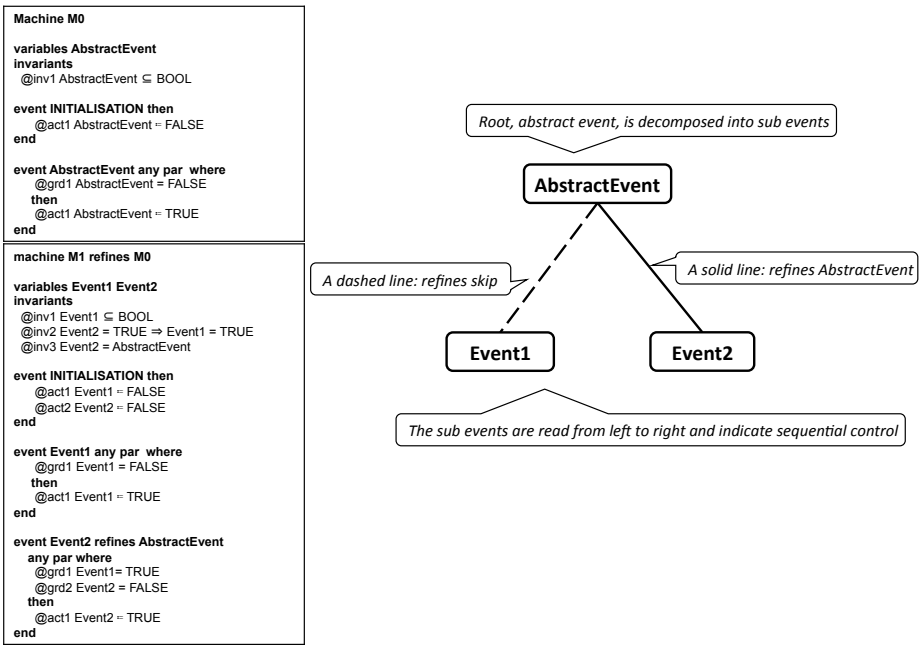


Fig. 1. Event Refinement Structure (ERS) Diagram

while *Event1* is a new event that refines *skip*. Moreover, the diagram shows that the effect achieved by *AbstractEvent* in the abstract machine is realised in the refining machine by the occurrence of *Event1* followed by *Event2*.

In ERS, the sequential execution of the leaf events is depicted from left to right. The event sequencing is managed by additional control variables introduced into the underlying Event-B model. For instance, for each leaf event (node) represented in Fig. 1, there is one boolean control variable with the same name as the event. When the event Event1 occurs, the corresponding control variable is set to *TRUE*. The following event, *Event2*, can occur only after *Event1*. This is achieved by checking the value of the *Event1* control variable in the guard of *Event2*.

Boolean variables only allow controlling single execution of events. When multiple executions of an event are needed, the event is parameterised and set control variables are used instead of boolean ones. This allows the event to occur many times with different values of its parameter. A parameter can be introduced in an event by the ERS constructors. The ERS constructors used in this paper are illustrated by two simple examples in Fig. 2. The use of *all* constructor indicates that *Event1* is executed for all instances of the p parameter before execution of *Event2*, while the use of the constructor *some* indicates that *Event1* is executed for some of instances of the p parameter before execution of *Event2*. The corresponding control variables for *Event1* and *Event2* are defined as sets in the model.

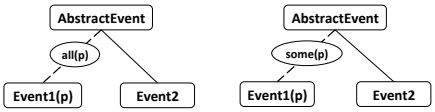


Fig. 2. ERS *all* /*some* Constructors

Event-B adopts an event-based modelling style that facilitates the correct-by-construction development of complex distributed systems. Since MapReduce is a framework designed for large-scale distributed computations, Event-B is a natural choice for its formal modelling and verification.

4 Formal Development with Event Refinement Structure

In this section, we rely on our formalisation presented in Section 2.2 to develop two alternative Event-B models of the MapReduce framework: *blocking* and *partially blocking*. The presented formal developments make use of the Event Refinement Structure (ERS) approach, presented in Section 3.2. Our development strategy is based on gradually unfolding all the MapReduce computational phases by refinement. Such small model transformation steps allow us to efficiently handle the complexity of the MapReduce framework.

Let us note that our development of the MapReduce framework is generic. It relies on the use of abstract functions to represent essential data transformations of MapReduce. These abstract functions can be treated as generic system parameters that can be later instantiated with their concrete instances for the specific MapReduce implementations.

4.1 Blocking Model of MapReduce

The mathematical data structures and their properties from our MapReduce formalisation constitute the basis for defining the Event-B *context* component that is used throughout the whole formal development. Essentially, the whole presented formalisation is incorporated as the context, e.g.

$\texttt{axm8:}\ MSplit \in IData \rightarrow (MTask \nrightarrow MData)$

$\texttt{axm9:}\ Map \in (MTask \nrightarrow MData) \rightarrow \mathbb{P}_1(MTask \times RData)$

$\texttt{axm10:}\ RSplit \in \mathbb{P}_1(MTask \times RData) \rightarrow (RTask \nrightarrow \mathbb{P}_1(RData)),\ ...$

We will constantly rely on these definitions to ensure the correctness of the overall data transformation process within our MapReduce models. Since the formalised definitions are still abstract (generic), our presented development essentially formally describes a family of possible MapReduce implementations. Due to space limit, we do not present the complete development but rather give its graphical representation using the ERS graphical notation. The full Event-B models of this development can be found in [8].

Abstract model of MapReduce. We start with an abstract model in which the whole MapReduce computation is done in one atomic step. This behaviour is modelled by the event OutputMapReduce:

OutputMapReduce $\widehat{=}$
any $t1,\ t2,\ t3,\ t4$
when $t1 = MSplit(idata) \wedge t2 = Map(t1) \wedge t3 = RSplit(t2) \wedge t4 = Reduce(t3)$
then $output := Combine(t4) \parallel done := TRUE$ **end**

With help of the ERS approach, we decompose the atomicity of OutputMapReduce into smaller steps. Verification of the refinement proof obligations ensures that the decomposition preserves correctness. Specifically, in the next several consecutive refinement steps, we break the atomicity of the OutputMapReduce event by introducing explicit events for the following MapReduce phases: *MSplit*, *Map*, *RSplit*, and *Reduce*. Fig. 3 presents the ERS diagram of the model.

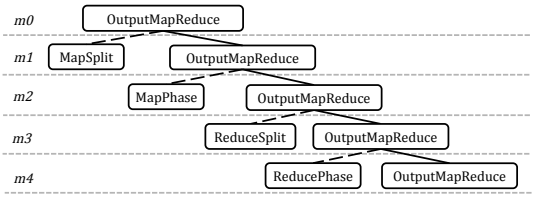


Fig. 3. Blocking model: ERS diagram (for OutputMapReduce)

The new model events MapSplit, MapPhase, ReduceSplit and ReducePhase specify the sequential execution of the MapReduce phases. The sequence between the events is enforced by following the rules given in Section 3.2. It is also specified by the invariant properties on the control variables:

$$OutputMapReduce = TRUE \Rightarrow ReducePhase = TRUE,$$
$$ReducePhase = TRUE \Rightarrow ReduceSplit = TRUE,$$
$$ReduceSplit = TRUE \Rightarrow MapPhase = TRUE,$$
$$MapPhase = TRUE \Rightarrow MapSplit = TRUE.$$

Moreover, to store the intermediate results of separate phases, we introduce a number of variables ($msplit$, map_result, $rsplit$ and $reduce\_result$) that are updated during execution of the corresponding events. The variable updates are also performed according to the formalisation given in Section 2.2. For instance, the variable $msplit$ is introduced to store the result of the *MSplit* phase. After the execution of the MapSplit event, $msplit$ gets the value equal to *MSplit(idata)*.

Machine MapReduce1_m1 **refines** MapReduce1_m0
Variables $idata,\ output,\ msplit,\ MapSplit,\ ...$
Invariants $OutputMapReduce = TRUE \Rightarrow MapSplit = TRUE \wedge$
$MapSplit = TRUE \Rightarrow msplit = MSplit(idata)\ \wedge ...$
MapSplit $\widehat{=}$
when $MapSplit = FALSE$
then $MapSplit := TRUE$
$msplit := MSplit(idata)$
end
OutputMapReduce **refines** OutputMapReduce $\widehat{=}$
any $t2,\ t3,\ t4$
where $MapSplit = TRUE \wedge OutputMapReduce = FALSE \wedge$
$t2 = Map(msplit) \wedge t3 = RSplit(t2) \wedge t4 = Reduce(t3)$
with $t1 = msplit$
then $output := Combine(t4)$
$OutputMapReduce := TRUE$
end

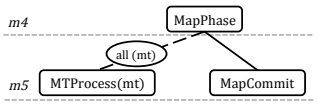


Fig. 4. Blocking model: ERS diagram (for MapPhase)

Breaking atomicity of the Map phase. In the second refinement step, we introduce the event MapPhase that abstractly models the *Map* phase. Essentially, the *Map* phase involves parallel execution of all the map tasks. To introduce such a behaviour, we use the constructor "*all* constructor", which is applied to the MTProcess event that models the execution of a particular map task (see Fig.4). The expression "*all(mt)*" means that the MTProcess event can be enabled for multiple values of $mt \in \mathsf{dom}(msplit)$. On the other hand, the MapCommit event can only occur when all the map computations of map tasks have been finished. In Event-B, we model this by adding a variable $MTProcess$, which is a set containing all possible map tasks that should be processed. The order between the events is ensured by the invariants on the control variables, e.g.,

$$\texttt{inv4:}\ \ MapCommit = TRUE \Rightarrow MTProcess = \mathsf{dom}(msplit),$$

where $\mathsf{dom}(msplit)$ defines the set of all current map tasks. The invariant states that if the MapCommit event has been executed, then all the map tasks have been completed before it. While specifying the MTProcess event, we rely on the definition of the `map` function, given in Section 2.2.

Machine MapReduce1_m5 **refines** MapReduce1_m4
MTProcess $\widehat{=}$
  any mt
  where $MapSplit = TRUE \wedge mt \in dom(msplit) \wedge mt \notin MTProcess$
  then $MTProcess := MTProcess \cup \{mt\}$
    $MTProcess\_result(mt) := map(msplit(mt))$
  end
MapCommit **refines** MapPhase $\widehat{=}$
  when $MapSplit = TRUE \wedge MapCommit = FALSE \ \wedge MTProcess{=}dom(msplit)$
  then $MapCommit = TRUE$
    $map\_result := (\bigcup mt \cdot mt \in dom(msplit) | \{mt\} \times MTProcess\_result(mt))$
  end

Further refinements of the Map phase. During the MapReduce execution, all the map and reduce tasks are parallelised and distributed to multiple processing nodes – the actual software components that carry out the computations. We name these components as *map* and *reduce workers*. Moreover, there is a special component – *master* – that controls all the computations and assigns the map and reduce tasks to the workers. The master periodically pings every worker. In case of a worker failure, the master re-assigns tasks from the failed worker to a healthy one. This procedure can be repeated until the master gets the result for a particular map or reduce task from some worker. To introduce such functionality, we carry out several further refinements focusing on the *Map* phase. These refinements elaborate on modelling of map task execution.

Fig.5 illustrates the event MTProcess and its several consecutive levels of atomicity decomposition. First, the abstract event MTProcess is broken into two concrete events, MTok and MTSuccess correspondingly. The MTok event models the execution of the map task mt by a particular map worker mw. The result

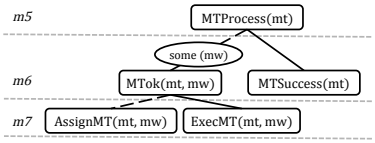


Fig. 5. Blocking model: ERS diagram (for MTProcess)

of this computation should be approved by the master side, which is modelled by execution of the MTSuccess event. The "*some*" constructor indicates that the event MTok may be executed only for some instances of the *mw* parameter before the MTSuccess event becomes enabled. The *MTSuccess* and *MTok* control variables are defined as sets, which allows for multiple executions of the MTSuccess and MTok events. Later on, in the next refinement step, the atomicity of the MTok event is broken into two events AssignMT and ExecMT. The event AssignMT models an assignment of a map task *mt* to a particular map worker *mw*, while ExecMT models the successful execution of the task by this worker.

Similarly to the *Map* phase, we refine the *Reduce* phase by gradually unfolding its computations. The overall refinement structure is presented on Fig.6.

Let us note that the proposed architecture is *blocking* in the sense that the reduce computations can be only started after all the map computations have been finished. The formal derivation of the blocking model and its dynamics is performed under this condition. Next we propose an alternative architectural solution of the MapReduce framework that weakens blocking between the map and reduce stages and, as a result, achieves a higher degree of parallelisation of the MapReduce computations. For this purpose, we will make use of the *dependence relation* between map and reduce tasks introduced in the Section 2.2. We call this model *partially blocking model*.

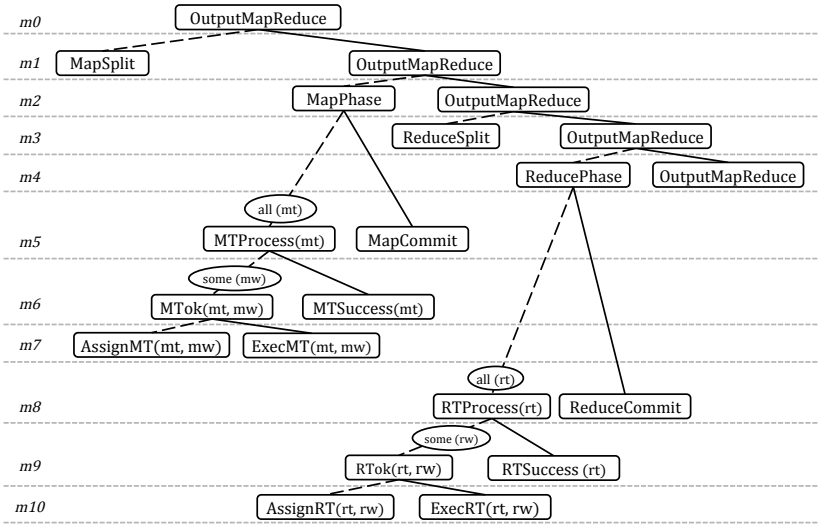


Fig. 6. MapReduce ERS Diagram: blocking model

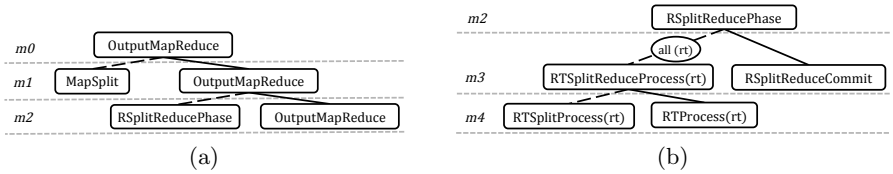


Fig. 7. Partially blocking model: ERS diagrams

4.2 Partially Blocking Model of MapReduce

We start from the same initial specification as for the blocking model, in which the whole MapReduce computation is done in one atomic step, and then refine it in order to introduce the *MSplit* phase. Next, in contrast to the previous derivation, we separate the phase that combines executions of the *RSplit* and *Reduce* phases – RSplitReducePhase. Fig.7 (a) presents the ERS diagram of the refined model.

RSplitReducePhase involves executions of the *RSplit* and *Reduce* phases for all reduce tasks. Essentially, these computations are parallelised. To introduce such behaviour, we use the"*all*" constructor applied to the RTSplitReduceProcess event that, for a particular reduce task rt, performs split and then reduce computations (see. Fig.7 (b)). Next, we separate these split and reduce executions of the particular reduce task rt. Namely, the event RTSplitReduceProcess is split into two concrete events, RTSplitProcess and RTProcess. Here we again rely on the *rsplit* and *reduce* functions formalised in Section 2.2.

Up to now we did not introduce the Map phase explicitly. However, the results of *MapPhase* are simulated internally, by storing the intermediate results in the local variables of the RTSplitProcess event. To explicitly model the Map phase, the event RTSplitProcess is now split into two events MTProcess and RSplit (see Fig.8). The constructor "all" is parameterised by $(mt \in dep[\{rt\}])$". It means that the event MTProcess is executed for all those map tasks, mt, that are in data dependency with the reduce task rt. Therefore, to start the *RSplit* phase, we do not need to wait until all the map tasks are completed. Here we are relying on the definition of data interdependency *dep* between the map and reduce stages, formalised in Section 2.2. Finally, the MTProcess and RTProcess events are refined in the same manner as in the blocking model presented in the Section 4.1.

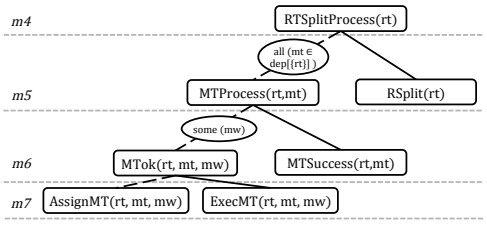


Fig. 8. Partially blocking model: ERS diagram (for RTSplitProcess)

Let us note that the proposed partially blocking model allows us to achieve a higher degree of parallelisation of MapReduce computations. Indeed, for a particular reduce task, when the dependent map tasks have already been executed, the *RSplit* phase for this reduce task can be performed, and then reduce computations can be started. In other words, the computations from three different phases – *Map*, *RSplit*, and *Reduce* – can be performed in parallel, provided the involved data are independent. Therefore, the proposed architectural solution weakens blocking between the stages and, as a result, achieves a higher degree of parallelisation. The overall refinement structure of the partially blocking model is presented on Fig.9.

4.3 Discussion and Future Work

To verify correctness of the presented models, we have discharged around 270 proof obligations for the first formal development, as well as more than 300 for the second one. Approximately 93% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. With help of the ERS approach, we have decomposed the atomicity of the MapReduce framework and hereby achieved a higher degree of automation in proving. Moreover, the ERS diagrammatic notation has provided us with additional support to represent the model control flow at different abstraction levels and also simplified reasoning about possible refinement strategies. The whole development and proving effort has taken about one person-month.

As a result of the presented refinement chains, we have arrived at two different centralised Event-B models of the distributed MapReduce framework. As a part of the future work, we are planing to derive distributed models by employing the existing decomposition mechanisms of Event-B. This would result in

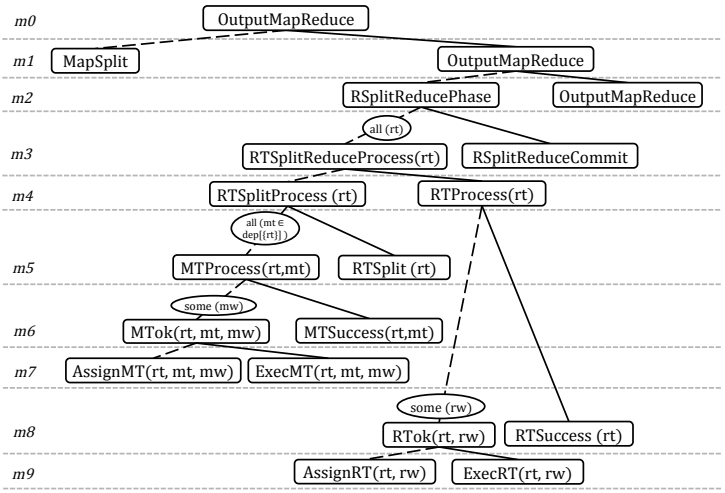


Fig. 9. MapReduce Event Refinement Structure: partially blocking model

creating separate formal specifications of the involved software components of the MapReduce framework (such as master, map worker, reduce worker, etc.).

The static part of the modelled system is formally defined in the corresponding context component. The definitions of static data structures in the context are mostly very abstract, i.e. they state only essential properties to be satisfied. This makes them generic parameters of the whole formal development. In its turn, such formal development becomes generic, representing a family of the systems that can be described by providing suitable concrete values for the generic parameters. The proposed formal model can be used then as a starting point for future development of a specific MapReduce application. The actual concrete values can be supplied by either the end-user (e.g., the `map` and `reduce` functions) or the developer of the MapReduce framework (e.g., the *MSplit* or *RSplit* transformations).

As a continuation of this work, it would be interesting to create formal models for a concrete MapReduce implementation, e.g., the word counting example, by using the Event-B generic instantiation plug-in. Moreover, to analyse the quantitative characteristics of the proposed models, we are planing to use the Uppaal-SMC model checker. This would allow us to, e.g., assign different data processing rates for the map and reduce tasks and then compare the execution time estimations of two considered architectures.

5 Related Work and Conclusions

The problem of formalisation of the MapReduce framework has been studied in [12]. The authors present a formal model of MapReduce using the CSP method. In their work, they focus on formalising the essential components of the MapReduce framework: the master, mapper, reducer, the underlying file system, and their interactions. In contrast, our focus is on modelling the overall flow of control as well as the data interdependencies between the MapReduce computational phases. Moreover, our approach is based on the stepwise refinement technique that allowed us to gradually unfold the complexity of the MapReduce framework.

Formalisation of MapReduce in Haskel is presented in [9]. Similarly to our approach, it focuses on the program skeleton that underlies MapReduce computations and considers the opportunities for parallelism in executing MapReduce computations. However, in addition to that, we also reason about the involved software components – the master, map and reduce workers – that are associated with the respective map and reduce tasks.

The work [7] presents two approaches based on Coq and JML to formally verify the actual running code of the selected Hadoop MapReduce application. In our work we are more interested in formalisation of MapReduce computations and gradual building of different MapReduce models that are correct-by-construction. The performance issues of MapReduce computations have been studied in the paper [4], focusing on one particular implementation of the MapReduce – Hadoop. In contrast, we have tried to formally investigate the data interdependencies between the MapReduce phases and their effect on the degree of parallelisation, independently of a concrete MapReduce implementation.

In this paper we have proposed an approach to formalising the MapReduce framework. Our main technical contribution of this paper is two-fold. On the one hand, based on our definition of interdependencies between the processed data as well as the map and reduce stages, we have derived the conditions under which blocking between the stages can be relaxed. Therefore, we have rigorously derived constraints for implementing MapReduce with a higher degree of parallelisation. On the other hand, we have demonstrated how to use the Event Refinement Structure (ERS) technique to formally derive and verify a model of a complex system with a massively parallel architecture and complex dynamic behaviour.

The stepwise refinement approach to deriving a complex system model has demonstrated good scalability and allowed us to express system properties at different levels of abstraction and with a different degree of granularity. Moreover, combining the refinement technique with tool-assisted mathematical proofs have provided us with a scalable approach to verification of a complex system model.

Acknowledgements. The authors would like to thank the reviewers for their valuable comments. Pereverzeva's work is partly supported by the STV Grant. Butler and Salehis work is partly funded by the FP7 ADVANCE Project (http://www.advance-ict.eu).

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design. In: The Apache Software Foundation (2007)
3. Butler, M.: Decomposition Structures for Event-B. In: Integrated Formal Methods, 7th International Conference, IFM 2009. pp. 20–38. Springer Heidelberg (2009)
4. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M.: MapReduce Online. In: NSDI 2010. pp. 20–20. USENIX Association (2010)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation. pp. 137–150. USENIX Association (2004)
6. Fathabadi, A.S., Butler, M., Rezazadeh, A.: A Systematic Approach to Atomicity Decomposition in Event-B. In: SEFM 2012. LNCS, vol. 7504, pp. 78–93. Springer-Verlag Berlin Heidelberg (2012)
7. Ono, K., Hirai, Y., Tanabe, Y., Noda, N., Hagiya, M.: Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications. In: SEFM 2011. pp. 350–365. Springer Berlin Heidelberg (2011)
8. Pereverzeva, I., Butler, M., Fathabadi, A.S., Laibinis, L., Troubitsyna, E.: Formal Derivation of Distributed MapReduce. Tech. Rep. 1099, TUCS (2014)
9. Ralf Lämmel: Google's MapReduce programming model. vol. 70, pp. 1–30 (2008)
10. Rodin: Event-B Platform, online at http://www.event-b.org/
11. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive - A Warehousing Solution Over a Map-Reduce Framework. In: Proc. VLDB Endowment. vol. 2, pp. 1626–1629 (2009)
12. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing MapReduce with CSP. In: 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. pp. 358–367. IEEE computer society (2010)

Paper III

Formal Development of Critical Multi-Agent Systems: A Refinement Approach

Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis

Originally published in: Cristian Constantinescu, Miguel P. Correia (Eds.), *Proceedings of 9th European Dependable Computing Conference (EDCC 2012)*, 156–161, IEEE Computer Society, 2012
The final publication is available at http://dx.doi.org/10.1109/EDCC.2012.24

Formal Development of Critical Multi-Agent Systems: A Refinement Approach

Inna Pereverzeva*†, Elena Troubitsyna* and Linas Laibinis*
*Åbo Akademi University
†Turku Centre for Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland
Email: {inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi

***Abstract*—Multi-agent systems (MAS) are increasingly used in critical applications. To ensure dependability of MAS, we need powerful development techniques that would allow us to master complexity inherent to MAS and formally verify correctness and safety of collaborative agent activities. In this paper we present a development of hospital MAS by refinement in Event-B. We demonstrate that Event-B allows the developers to rigorously specify complex agent interactions and verify their correctness and safety.**

***Keywords*-Event-B; refinement; formal modelling; formal verification; multi-agent system; safety**

I. Introduction

Mobile multi-agent systems (MAS) are complex decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programs acting autonomously on behalf of a person or organisation, while coordinating their activities by communication [1]. MAS are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas, etc. However, widespread use of MAS is currently hindered by the lack of methods for ensuring their dependability.

In this paper we focus on studying complex agent interactions. In a critical MAS, incorrect execution of these activities might have devastating consequences. However, ensuring correctness of complex interactions is a challenging issue due to faults caused by agent disconnections, dynamic role allocation and autonomy of the agent behaviour. To address these challenges, we need the system-level modelling approaches that would support formal verification of correctness and facilitate discovery of restrictions that should be imposed on the system to guarantee its safety.

In this paper we develop a hospital MAS by refinement in Event-B [2]. Refinement is a top-down approach to formal development of systems that are correct by construction. The system development starts from an abstract specification which defines the main behaviour and properties of the system. The abstract specification is gradually transformed (refined) into a more concrete specification directly translatable into a system implementation. Correctness of each refinement step is verified by proofs. The Rodin platform [3] provides the developers with automated tool support for constructing and verifying system models in Event-B.

In our development we demonstrate how to gradually *derive* a system implementation that satisfies the desired safety properties. It is different from traditional approaches to verification of MAS that extract a model from a system implementation and verify the desired properties by state-exploration. Our approach is not only free of the state explosion problem but also allows the designers to discover restrictions that should be imposed on the system environment to guarantee system safety. We argue that the formal development in Event-B offers a useful technique for development and verification of complex critical MAS.

The paper is structured as follows. In Section II we describe our formal modelling framework – Event-B. In Section III we propose main principles for formal reasoning about MAS and their properties. Section IV presents our case study – a hospital MAS. We show here how to abstractly model a MAS, introduce complex collaborative agent interactions by refinement, as well as verify safety properties. Finally, in Section V we overview the related work and discuss the achieved results.

II. Formal Modelling and Refinement in Event-B

We start by briefly describing our formal development framework. The Event-B formalism is a variation of the B Method [4], a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems. Currently Event-B is actively used within the FP7 ICT project DEPLOY to develop dependable systems from various domains [5].

In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the model state represented as a collection of variables, and defines operations on this state, i.e., it describes the *behaviour* of the modelled system. A machine may also have the accompanying component, called *context*. A context may include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. In Event-B, the model variables are strongly typed by the constraining predicates called *invariants*. Moreover, the invariants specify important properties that should be preserved during the system execution.

The dynamic behaviour of the system is defined by the set of atomic *events*. Generally, an event can be defined as

$$\textbf{evt} \mathrel{\widehat{=}} \textbf{any } vl \textbf{ where } g \textbf{ then } S \textbf{ end}$$

where vl is a list of new local variables (parameters), g is the event *guard*, and S is the event *action*. The guard is a state predicate that defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments.

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness. The consistency of Event-B models, i.e., invariant preservation, correctness of refinement steps, should be formally demonstrated by discharging relevant proof obligations [2]. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the Rodin platform [3]. Proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of such complex distributed systems as multi-agent systems.

Recently the Event-B language and Rodin have been extended with a possibility to define modules [6], [7] – components containing groups of callable atomic operations. Modules can have their own (external and internal) state and invariant properties. An important characteristic of modules is that they can be developed separately and, when needed, composed with the main system.

In the next section, we outline main principles of formal reasoning about MAS and their properties.

III. Formal Reasoning about MAS

Let us start by defining a MAS formally.

Definition 1. *A multi-agent system $\mathcal{MAS}$ is a tuple $(\mathcal{A}, \mu, \mathcal{E}, \mathcal{R})$, where $\mathcal{A}$ is a collection of different classes of agents, μ is the system middleware, $\mathcal{E}$ is a collection of system events and $\mathcal{R}$ is a set of dynamic relationships between agents in a MAS.*

Each agent belongs to a particular class or type of agents $A_i,\ i \in 1..n$, such that $A_i \in \mathcal{A}$. This class is dynamic, i.e., new class agents can spontaneously appear or the existing agents may disappear (both normally or abnormally). A particular agent $a_{ij} \in A_i$ is characterised by its local state consisting from agent variables and static agent attributes.

The system middleware μ can be considered as a special agent that is always present in the system. The responsibility of the middleware is to ensure communication between agents, to detect agent appearance or disappearance, recover from the situations when the required connections between the system agents are lost, etc.

The system events $\mathcal{E}$ include all internal and external system reactions. An execution of an event may change the state of the middleware or agents. Each collaborative activity between different agents (or an agent and the middleware) may be composed of a set of events. Moreover, system events may model appearance or disappearance of mobile agents, sending request from one agent to another, recovery of lost agent connections, etc. The collaborative action should preserve the following property:

Property 1. *Let $\mathcal{A}_{act}$ and $\mathcal{A}_{ina}$ be sets of active and inactive agents correspondingly, where $\mathcal{A} = \mathcal{A}_{act} \cup \mathcal{A}_{ina}$ and $\mathcal{A}_{act} \cap \mathcal{A}_{ina} = \varnothing$. Let $\mathcal{EAA}$ and $\mathcal{EA}\mu$ be all the collaborative activities (sets of events) between agents and agents and between agents and middleware respectively. Moreover, for each $A \in \mathcal{A}$, let $\mathcal{E}A$ be a set of events in which the agent A is involved. Then*

$$\forall A \cdot\ A \in \mathcal{A}_{act} \Rightarrow \mathcal{E}A \in \mathcal{EAA}$$

and

$$\forall A \cdot\ A \in \mathcal{A}_{ina} \Rightarrow \mathcal{E}A \in \mathcal{EA}\mu.$$

For instance, this property implies that while an agent is recovering from a failure it cannot be involved into cooperative activities with other agents.

A collection of system events $\mathcal{R}$ consists of dynamic relationships or connections between active agents of the same or different classes. An agent relationship is modelled as a relation

$$R(a_1, a_2, ..., a_m) \subseteq C_1^* \times C_2^* ... \times C_m^*,$$

where $C_j^* = C_j \cup \{?\}$, $C_j \subset \mathcal{A}_{act}$, $j \in 1..m$. A relationship can be *pending*, i.e., incomplete. This is indicated by question marks in the corresponding places of R, e.g., $R(a_1, a_2, ?, a_4, ?)$. Pending relationships are usually caused by appearance of new agents or disappearance of the existing ones. An existing agent may initiate a new relationship.

Property 2. *Let $\mathcal{A}_{act}$ be a set of active agents. Let $\mathcal{EAA}$ be all the collaborative activities in which these active agents are involved. Moreover, for each agent $A \in \mathcal{A}_{act}$, let $\mathcal{R}_A$ be all the relationships it is involved. Finally, for each collaborative activity $CA \in \mathcal{EAA}$, let $\mathcal{A}_{CA}$ be a set of the involved agents in this activity. Then, for each $CA \in \mathcal{EAA}$ and $A_1, A_2 \in \mathcal{A}_{CA}$, $\mathcal{R}_{A_1} \cap \mathcal{R}_{A_2} \neq \varnothing$.*

This property restricts the interactions between the agents – only the agents that are linked by relationships (including the pending ones) can be involved into cooperative activities.

The system middleware μ keeps a track of pending relationships and tries to resolve them by enquiring suitable agents to confirm their willingness to enter into a particular relationships. Additional data structure $Pref_R$ associated with a relationship $R \in \mathcal{R}$ can be used to express a specific preference of one agents over the other ones. The middleware then enforces this preference by enquiring the preferred agents first. Formally, $Pref_R$ is an ordering relation over the involved agent classes. Thus, for $R \subseteq C_1^* \times ... \times C_m^*$,

$$Pref_R \ \in\ C_1 \times ... \times C_m \ \leftrightarrow\ C_1 \times ... \times C_m.$$

A responsibility of the middleware is to detect situations when some of the established or to be established relationships become pending and guarantee "fairness", i.e., no pending request will be ignored forever, as well as try to enforce the given preferences, if possible.

While developing a critical MAS, we should ensure that certain cooperative activities, once initiated, are successfully completed. Those are the activities that implement safety requirements. The ensure safety we have to verify the following property:

Property 3. *Let $\mathcal{EAA}_{crit}$, where $\mathcal{EAA}_{crit} \subseteq \mathcal{EAA}$, be a subset containing critical collaborative activities. Moreover, let $\mathcal{R}_{pen}$ and $\mathcal{R}_{res}$, where $\mathcal{R}_{pen} \subseteq \mathcal{R}$ and $\mathcal{R}_{res} \subseteq \mathcal{R}$, be subsets of pending and resolved relationships defined for these activities. Finally, let $\mathcal{R}_{CA}$, where $CA \in \mathcal{EAA}$ and $\mathcal{R}_{CA} \subseteq \mathcal{R}$, be all the relationships the activity CA can affect. Then, for each activity $CA \in \mathcal{EAA}_{crit}$ and relationship $R \in \mathcal{R}_{CA}$,*

$$\Box((R \in \mathcal{R}_{pen}) \rightsquigarrow (R \in \mathcal{R}_{res})),$$

where $\Box$ designates "always" and $\rightsquigarrow$ denotes "leads to".

This property postulates that eventually all pending relationships should be resolved for each critical cooperative activity. In the next section we will present modelling a hospital MAS in Event-B.

IV. Abstract Modelling of a Hospital MAS

A. Requirements

The hospital MAS consists of two types of agents – patients and medical personnel (called doctors for simplicity). The condition of each patient is monitored by the corresponding medical equipment – an agent representing a patient. The doctor agents are running on Pocket PC-based devices – Personal Digital Assistants (PDA). The hospital provides the wireless connectivity to the doctor agents. Each doctor is associated with one agent.

The medical equipment continuously updates the patient's medical record consisting of different medical measurements as well as detects emergencies – dangerous changes of critical parameters (e.g., blood pressure, pulse rate, etc.). In case of an emergency, the patient agent generates an emergency call that is communicated to the doctor(s) treating the patient. An important safety requirement imposed on the system is that *all emergencies should be promptly handled by the responsible doctors*. In spite of its seeming simplicity, this requirement is hard to ensure. Indeed, a MAS operates in a volatile communication environment, i.e., agents might experience temporal disconnections. Hence the design of our system should incorporate certain fault tolerance mechanisms that would guarantee that each emergency call is eventually handled. Moreover, different doctors can be associated with the same patient during different shifts. Therefore, we have to ensure that at the end of a doctor's shift all his/her patients are handed over to another doctor.

Another important safety requirement associated with the system is to guarantee that *a doctor always accesses the most recent patient record and the patient's data are always kept in a consistent state*. We assume that the patient record is stored at the equipment associated with her. To ensure that these requirements are satisfied, we should regulate the access to the patient's data. A specific delivery of a medicine, prescription of a treatment and so on are introduced into the patient's log by the medical personnel via their PDAs. To ensure that the data are updated consistently, we only allow the doctor to modify the patient's data when he or she is in a close proximity to the patient. Specifically, the patient data become available at the doctor's PDA only when she/he arrives to the patient location. All the modifications are synchronised with the data stored by the patient's equipment. When the doctor finishes examining the patient or delivering a medicine and leaves, the connection to the patient's data is lost. Such a restriction allows us to ensure that only a doctor who is in a close proximity to a patient is allowed to modify the patient's record. Moreover, it ensures that the doctor has the access to the freshest info about the patient.

The safety-critical requirements imposed on the system should be fulfilled in the course of complex agent interactions. Hence, the properties defined in the previous section are put in the context of doctor-patients interactions. Indeed, each patient should be assigned to a certain doctor.

While the doctor is inactive (because the shift is over or a failure of the corresponding PDA), doctor-patient interactions cannot occur. Finally, each emergency call should be eventually served. Next we demonstrate how to model and verify agent interactions in Event-B.

B. Towards modelling agent interdependencies

Our abstract specification models the behaviour of the entire hospital MAS in a highly abstract way. We define the set of active doctor agents as $med\_agents \subseteq MEDSTAFF$. The events *Activate* and *Deactivate* model joining and leaving hospital location by the agents. While an agent is active, it can perform certain activities, which is abstractly modelled by the event *Activity*.

In our first refinement step we augment our model with a representation of patients, $patients \subseteq PATIENTS$, and introduce the events that abstractly model interactions between the introduced patients and the doctors.

Each patient arriving at the hospital is associated with a doctor who has a primary responsibility for treating the patient:

$$assigned\_doctor \in patients \rightarrow med\_agents.$$

We omit showing complete specifications of the subsequent refinement steps; complete specifications of the entire development can be found at [8].

At the first refinement step, resulting in the model *Hospital1*, we add the new events *PatientArrival* and *PatientDischarge* to model a patient arrival and patient discharge from the hospital correspondingly. The event *PatientArrival*

ensures the following property, which can be considered as an instance of the *Property 3*:

$$\square(\text{new patient} \rightsquigarrow \text{assigned doctor}),$$

while, similarly, the event *PatientDischarge* ensures that

$$\square(\text{a patient leaves the hospital} \rightsquigarrow \text{all his/her relationships are removed}).$$

In the refined specification we also elaborate on the event *Activity*. Essentially, the medical personnel should examine the patients and deliver the prescribed medicine. We generalise these actions under the general term "visiting a patient". Events *VisitBegin* and *VisitEnd* refine the abstract event *Activity* and model a visiting procedure.

In our abstract specification we have assumed that the doctor agents can leave the hospital at any time. However, formal modelling has uncovered that safety cannot be guaranteed under this assumption. Hence we must impose certain restrictions on the situations when the doctors can actually leave the hospital. Before a doctor agent can leave the hospital, we should reassign his/her patients to another doctor. This is another illustration of the *Property 3*:

$$\square(\text{a doctor leaves the hospital} \rightsquigarrow \text{all his/her patients are reassigned}).$$

We split the abstract event *Deactivate* into two corresponding events: *AgentLeaving* and *ReassignDoctor*. The event *AgentLeaving* models leaving the location by a doctor. Here we check that a doctor does not have any assigned patients and is not currently involved in examining a patient:

$AgentLeaving \mathrel{\widehat{=}}$ **Refines** $Deactivate$
any ma
when
$ma \in med\_agents \land ma \notin ran(assigned\_doctor) \land$
$ma \notin last\_visit[visited]$
then
$med\_agents := med\_agents \setminus \{ma\}$
end

The event *ReassignDoctor* models leaving the location by a doctor who has assigned patients. We must be sure that all his/her patients will be reassigned to the new doctor.

C. Introducing fault tolerance by refinement

A doctor agent might leave the location because the doctor's shift is over or because the doctor agent has irrecoverably failed and should be permanently disconnected. At the second refinement step we introduce a distinction between the normal agent leaving and its disconnection due to a failure. Both cases have affect the safety requirements.

In a MAS, the agents often lose connection only for a short period of time. After the connection is restored, the agent should be able to continue its operations. Therefore, after detecting a loss of connection, the location should not immediately disengage the disconnected agent but rather set a deadline before which the agent should reconnect. If the disconnected agent restores its connection before the deadline then it can continue its normal activities. However, if the agent fails to do so, the location should permanently disengage the agent.

We introduce new variables and events that model the behaviour described above: *DisconnectAgent*, *ReconnectionSuccessful*, *ReconnectionFailed* events. The introduction of an agent disconnection allows us to make a distinction between two reasons behind leaving the location by a doctor – because of the end of shift or due to the disconnection timeout. To model these two cases, we split the event *AgentLeaving* into two events *NormalAgentLeaving* and *DetectFailedFreeAgent* respectively.

While modelling failure of a doctor agent, we again encounter the *Property 3*. In both cases we need to reassign all the patients of the disconnected doctor to another doctor. The event *ReassignDoctor* is decomposed into two events *NormalReassignDoctor* and *DetectFailedAgent*:

$DetectFailedAgent \mathrel{\widehat{=}}$ **Refines** $ReassignDoctor$
any ma, ma_new
when
$ma \in ran(assigned\_doctor) \land ma \notin last\_visited[visited] \land$
$ma\_new \in med\_agents \land ma\_new \neq ma \land$
$ma \in disconnected \land timer(ma) = timeout \land$
$(ma\_new \notin disconnected \lor$
$(ma\_new \in disconnected \land timer(ma\_new) = active))$
then
$med\_agents := med\_agents \setminus \{ma\} \parallel$
$assigned\_doctor := assigned\_doctor \Leftarrow$
$(dom(assigned\_doctor \rhd \{ma\}) \times \{ma\_new\}) \parallel$
$disconnected := disconnected \setminus \{ma\} \parallel timer := \{ma\} \lhd timer$
end

The second refinement step has resulted in a specification ensuring that no patients are left unattended neither because of the doctors shift change nor because of a doctor agent failure.

D. Ensuring Safety of Cooperative Agent Actions

Our next refinement step introduces abstract modelling of emergency calls, which are generated by patient monitoring equipment. We must ensure that each call will be properly handled by a corresponding doctor. Hence our next refinement step should transform the specification to ensure the following:

$$\square(\text{an emergency call} \rightsquigarrow \text{a medical visit happens}).$$

In this refinement step, we introduce the variable $emergency\_calls$ that associates the emergency calls with the patients:

$$emergency\_calls \in ALARMS \nrightarrow patients.$$

Moreover, we define the variable $accepted\_calls$ that establishes the correspondence between the emergency calls and the doctors that answer them:

$$accepted\_calls \in ALARMS \nrightarrow med\_agents.$$

We add new events *EmergencyCall* and *HandlingEmergencyCall* to abstractly model the occurrence of an emergency and finding a responsible doctor to handle it:

$HandlingEmergencyCall \mathrel{\widehat{=}}$ **Status** $convergent$
any ec, ma
when
$ec \in dom(emergency\_calls) \land ec \notin dom(accepted\_calls) \land$
$ma \in med\_agent \land ma \notin disconnected$
then
$accepted\_calls := accepted\_calls \cup \{ec \mapsto ma\}$
end

To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events converge. To prove it, we need to define a model variant – natural number expressions on the model variables [2]. A developer should prove that the variant expression is decreased after an event execution. In our case, a specific system variant ensures that the newly introduced events *EmergencyCall* and *HandlingEmergencyCall* do not take the control forever. Those events have status *convergent*. We define the variant as follows:

$$card(ALARMS \setminus dom(emergency\_calls)) + card(ALARMS \setminus dom(accepted\_calls)),$$

and prove that it is decreased by new events. Variants play an important role in ensuring, e.g., that error recovery terminates, service requests are eventually served, etc. In our case the variant allows us to guarantee that eventually some doctor is chosen to handle an emergency.

Moreover, at this refinement step we distinguish two types of a patient visit – a regular visit and a visit for handling an emergency call. To model this, we decompose the event *VisitBegin* into events *RegularVisitBegin* and *EmergencyVisitBegin*, see [8] for details.

The goal of our next refinement step is to introduce a detailed procedure of selecting a doctor in the case of an emergency call. The proposed procedure can be described as follows. We start by selecting an emergency call to answer. Then we model a loop of finding a suitable candidate and sending a request to him/her. If the doctor rejects it then we choose the next candidate. The procedure is repeated until we get an acceptance of the request.

To model the described procedure, we introduce a number of events to specify the corresponding steps of the selection procedure: *CallFeed*, *AcceptCall*, *ForwardCall*, *RejectCall*, etc. Note that a doctor agent can refuse to accept a call. For instance, by checking the doctor schedule it might discover that he/she is currently performing a scheduled surgery.

While verifying correctness of this refinement step, we encounter a problem in the system requirements – we cannot guarantee safety unless we assume that during the search for a doctor no disconnection of agents can occur. Hence, our system should ensure (e.g., by implementing a certain protocol) that finding a doctor takes a very short period of time. Moreover, we define an additional system variant – $card(med\_agents \setminus occupied)$ – to ensure that the event *RejectCall* is *convergent*, which means that eventually we should get an acceptance from a doctor to answer the call.

E. Data integrity

To ensure that a patient gets a correct treatment, we should guarantee that the medical personnel always access the most recent patient record. As we discussed in Section 3, we allow a doctor to access and modify the patient's data only when he/she is in a close proximity to the patient. We implement this requirement via the scoping mechanism [9], [10]. A scope provide a shared data space for a doctor and a patient. We assume that each patient agent has the scope associated with it. As soon as a doctor agent appears at a close vicinity of the patient agent, it automatically joins the scope. While in the scope, the doctor can modify the patient record (e.g., prescribe a new medicine, log the information about the delivered medicine, etc.).

To model this behaviour, we refine the abstract events *RegularVisitBegin*, *EmergencyVisitBegin*, *VisitEnd* by events *RegularEnterScope*, *EmergencyEnterScope*, *LeaveScope* and add a new event *ModifyRecord*. We introduce the variable $record$ that represents the medical history for every patient. The variable ma_data stores the data that appear on the doctor's PDA screen. When the doctor agent is in a close vicinity of a patient, its ma_data becomes equal to the value of the patient data. Finally, we define the variable $scopes$ – a partial function associating the active scopes with the doctors participating in them:

$$record \in patients \to \mathbb{P}(DATA)$$
$$ma\_data \in med\_agents \nrightarrow \mathbb{P}(DATA)$$
$$scopes \in ScopeName \rightarrowtail med\_agents$$
$$\forall ma \cdot ma \in disconnected \Rightarrow ma \notin ran(scopes).$$

The event *ModifyRecord* models an update of the patient record by a doctor, when he/she is in the scope of a patient:

$ModifyRecord \mathrel{\widehat{=}}$
any ma, sn, pa, da_new
when
$(sn \mapsto ma) \in scopes \wedge pa \in dom(last\_visit) \wedge pa \in visited \wedge last\_visit(pa) = ma \wedge da\_new \in \mathbb{P}(DATA) \wedge da\_new \neq \varnothing$
then
$ma\_data(ma) := da\_new \parallel record(pa) := da\_new$
end

The corresponding safety property stating that the medical personnel always access the most recent record is formulated as the invariant:

$$\forall ma, pa.\ (pa \mapsto ma) \in (visited \lhd last\_visit) \Rightarrow ma\_data(ma) = record(pa).$$

F. Decomposition

As a result of the previous refinement steps, we have arrived at a centralised model of the Hospital MAS. In the final refinement step we aim at deriving a distributed architecture. In reality, agents and the location communicate with each other via message passing. This refinement step is focused on defining this communication explicitly. To achieve this, we rely on the modularisation extension of Event-B.

The abstract model is refined by a model representing the *Location* part – the middleware, and two separate modules – *Doctor* and *Patient*. Each modules contain callable operations and both internal and external data. The *Location* part accesses the modules via the provided generic interfaces.

The *Location* model imports two module interfaces – *Doctor* and *Patient*. The majority of the events of the abstract machine are now refined by the location events

(*ActivateAgent*, *PatientArrival*, etc.). The remaining events, for instance, *AcceptCall*, *RejectCall*, *EmergencyCall*, etc., become a part of the autonomous processes of the doctor or patient interfaces. The abstract event *EmergencyCall* is refined by the patient's interface event *Emergency* implementing the occurrence of emergency calls. Similarly, the abstract events *AcceptCall*, *RejectCall*, *ModifyRecord* are refined by the corresponding doctors's interface events *Acceptance*, *Rejection* and *ModifyData*.

V. Conclusion

In this paper we have presented a formal development of a hospital MAS by refinement in Event-B. We formalised the main properties of MAS and demonstrated how refinement process can facilitate their preservation.

This paper focuses on modelling and verification of safety for two central critical activities – handling emergencies and consistent updates of patient data. Ensuring correctness of these activities was especially challenging due to highly dynamic nature of a hospital, volatile error-prone communication environment and autonomous agent behaviour.

The work presented in this paper is inspired by our previous work on modelling context-aware mobile agent systems [9], [10] in the CAMA framework [11], [12]. In a similar way, we rely on a timeout mechanism to tolerate agent disconnections and employ a scoping mechanism to provide shared data space for patient and doctor agents. However, in this paper we have focused on modelling and verification of safety properties of complex agent interactions rather than on reasoning about general mechanisms for agent interaction with middleware.

Formal modelling of MAS has been undertaken by [13], [14]. The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. In our approach we also have studied the problem of ensuring access to the fresh context. However, in [13] it is solved at the level of the matching agent attributes while in our approach we rely on the scoping mechanism to achieve this.

A formal modelling of MAS for the health care in Z has be undertaken by Gruer at al. [15]. The work has focused on specifying a multi-agent system for a medical help system. The authors aimed at studying how to formally represent agent interactions, e.g., during negotiations. In our approach we not only model the agent interactions but also formally prove their properties. Hence, our approach is especially suitable for developing critical MAS.

Our approach is different from numerous process-algebraic approaches used for modelling MAS. Firstly, we have relied on proof-based verification that does not impose restrictions on the size of the model, number of agents etc. Secondly, we have adopted a system's approach, i.e., we modelled the entire system and extracted specifications of its individual components by decomposition. Such an approach allows us to express and formally verify safety of the overall system, i.e., we indeed achieve verification of safety as a system level property. Finally, the adopted top-down development paradigm has allowed us to efficiently cope not only with complexity of requirements but also with complexity of verification. We have build a large formal model of a complex system by a number of rather small increments. As a result, verification efforts have been manageable because we merely needed to prove refinement between each two adjacent levels of abstraction. Hence, we conclude that refinement in Event-B constitutes a suitable technique for formal modelling and verification of critical multi-agent systems.

References

[1] OMG Mobile Agents Facility (MASIF), www.omg.org.

[2] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010.

[3] Rodin. Event-B Platform, http://www.event-b.org/.

[4] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[5] EU-project DEPLOY, http://www.deploy-project.eu/.

[6] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Supporting Reuse in Event B Development: Modularisation Approach," in *ABZ 2010, Abstract State Machines, Alloy, B, and Z*, 2010, pp. 174–188.

[7] Rodin Modularisation Plug-in, documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.

[8] I. Pereverzeva, E.Troubitsyna, and L. Laibinis, "Rigorous Development of a Safe Multi-Agent System," Turku Centre for Computer Science, Tech. Rep. 1004, 2011.

[9] L. Laibinis, E. Troubitsyna, A. Iliasov, and A. Romanovsky, "Rigorous Development of Fault-Tolerant Agent Systems," in *Rigorous Development of Complex Fault-Tolerant Systems*, 2006, pp. 241–260.

[10] L. Laibinis, E. Troubitsyna, A. Iliasov, and A. Romanovsky, "Fault Tolerant Middleware for Agent Systems: A Refinement Approach," in *EWDC 2009, European Workshop on Dependable Computing*, 2009.

[11] A. Iliasov and A. Romanovsky, "CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents," in *ECOOP 2005, Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions*, 2005.

[12] A. Iliasov, V. Khomenko, M. Koutny, and A. Romanovsky, "On Specification and Verification of Location-based Fault Tolerant Mobile Systems," in *Rigorous Development of Complex Fault-Tolerant Systems*, 2006

[13] G.-C. Roman, C. Julien, and J. Payton, "Modeling adaptive behaviors in Context UNITY," in *Theoretical Compuiture Science*, vol. 376, 2007, pp. 185–204.

[14] G.-C. Roman, P. McCann, and J. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," in *ACM Transactions of Software Engineering and Methodology*, 1997.

[15] P. Gruer, V. Hilaire, A. Koukam, and K. Cetnarowicz, "A formal framework for multi-agent systems analysis and design," in *Expert Systems with Applications*, 2002, pp. 349–355.

Paper IV

A Refinement-Based Approach to Developing Critical Multi-Agent Systems

Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis

Originally published in: *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 1, 69–91, 2013

A refinement-based approach to developing critical multi-agent systems

Inna Pereverzeva*

Turku Centre for Computer Science,
Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
E-mail: inna.pereverzeva@abo.fi
*Corresponding author

Elena Troubitsyna and Linas Laibinis

Åbo Akademi University,
Joukahaisenkatu 3-5A, 20520 Turku, Finland
E-mail: elena.troubitsyna@abo.fi
E-mail: linas.laibinis@abo.fi

Abstract: Multi-agent systems are increasingly used in critical applications. To ensure dependability of multi-agent systems, we need powerful development techniques that would allow us to master complexity inherent to such kind of systems and formally verify correctness and safety of collaborative agent activities. In this paper, we present a rigorous approach to the development and verification of critical multi-agent system in Event-B. We demonstrate how to formally specify complex agent interactions and verify their correctness and safety. We argue that the refinement approach facilitates structuring complex requirements and formal reasoning about system-level properties. We illustrate our approach by a case study: formal development of a hospital multi-agent system.

Keywords: Event-B; refinement; formal modelling; formal verification; safety; multi-agent system; MAS.

Reference to this paper should be made as follows: Pereverzeva, I., Troubitsyna, E. and Laibinis, L. (2013) 'A refinement-based approach to developing critical multi-agent systems', *Int. J. Critical Computer-Based Systems*, Vol. 4, No. 1, pp.69–91.

Biographical notes: Inna Pereverzeva received her MSc in Computer Scince from Petrozavodsk State University, Russia in 2009. She is a PhD student at the Department of Information Technologies of Åbo Akademi University, Finland. Her research interests include formal development and verification of safety-critical and fault-tolerant systems.

Elena Troubitsyna is an Academy Research Fellow at the Academy of Finland and Adj. Professor at the Department of IT at Åbo Akademi University. She received her PhD in Computer Science in 2000 on design methods for dependable systems. Her research interests include application of formal and structured methods to development of dependable complex systems. She is a leader of several national projects in the area of dependability and formal methods.

Linas Laibinis is a Senior Researcher at the Department of Information Technologies of Åbo Akademi University, Finland. He received his PhD in Computer Science in 2000 on mechanised formal reasoning about computer programmes. His research interests include interactive environments for proof and programme construction, as well as application of formal methods to modelling and development of fault tolerant and distributed software systems.

This paper is a revised and expanded version of a paper entitled 'Formal development of critical multi-agent systems: a refinement approach' presented at Dependable Computing Conference (EDCC), 2012 Ninth European, Sibiu, Romania, 8–11 May 2012.

1 Introduction

The field of mobile *multi-agent systems* (MASs) has been actively studied in the last decade. MASs are complex decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programmes acting autonomously on behalf of a person or an organisation, while coordinating their activities by communication (OMG Mobile Agents Facility, 1997). MASs are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas, etc. However, widespread use of MASs is currently hindered by the lack of methods for ensuring their dependability.

In this paper we focus on studying complex agent interactions. In a critical MAS, incorrect execution of these activities may have devastating consequences. However, ensuring correctness of complex interactions is a challenging issue due to faults caused by agent disconnections, dynamic role allocation and autonomy of the agent behaviour. To address these challenges, we need the system-level modelling approaches that would support formal verification of correctness and facilitate discovery of restrictions that should be imposed on the system to guarantee its safety.

In this paper we propose a refinement-based approach to developing critical MAS. Our formal modelling and verification framework is Event-B (Abrial, 2010). The main development technique of Event-B is *refinement* – a top-down approach to a formal development of systems that are correct by construction. A development starts from an abstract system specification that non-deterministically models the most essential system behaviour. In a sequence of refinement steps, we gradually reduce non-determinism and introduce detailed design decisions. The refinement technique allows us to ensure that a resulting specification preserves the globally observable behaviour and properties of the abstract specification it refines. Verification of each refinement step is done by proofs. These proofs establish system safety (via preservation of safety invariant properties expressed at different levels of abstraction) and liveness (via the provable absence of undesirable system deadlocks). Transitivity of the refinement relation allows us to guarantee that the system implementation adheres to the abstract specifications. (Rodin Platform, 2006) integrated development environment for Event-B – provides effective tool support for system modelling and verification.

The main novelty of our approach is in demonstrating how to gradually *derive* a system implementation that satisfies the desired safety properties. It is different from traditional approaches to verification of MAS that extract a model from a system implementation and verify the desired properties by state-exploration. Our approach is not only free of the state explosion problem but also allows the designers to discover restrictions that should be imposed on the system environment to guarantee system safety. The top-down development approach facilitates structuring of complex requirements and improves comprehensibility of formal models. We argue that the formal development in Event-B offers a useful technique for development and verification of complex critical MAS.

The paper is structured as follows. In Section 2 we describe our formal modelling framework – Event-B. In Section 3 we define the main principles of formal reasoning about MASs and their properties. In Section 4 we present our case study – a hospital MAS. We show here how to abstractly model a MAS, introduce complex collaborative agent interactions by refinement, as well as verify safety properties. Moreover, we describe the last refinement step that models system decomposition, thus achieving derivation of a distributed implementation from a centralised specification. Finally, in Section 5 we overview the related work, discuss the achieved results and outline our future work.

2 Formal modelling and refinement in Event-B

In this section we present our formal development framework – Event-B. The Event-B formalism – an evolution of the B method (Abrial, 2005) – is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems.

2.1 Modelling in Event-B

In Event-B, a system model is specified using the notion of an *abstract state machine* (Abrial, 2010). An abstract state machine encapsulates the model state represented as a collection of model variables, and defines operations on the state, i.e., it describes the dynamic behaviour of the modelled system. A machine may also have the accompanying component, called *context*. A context might include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. In Event-B, the model variables are strongly typed by the constraining predicates called *invariants*. Moreover, the invariants specify important properties that should be preserved during the system execution. A general form of Event-B models is given in Figure 1.

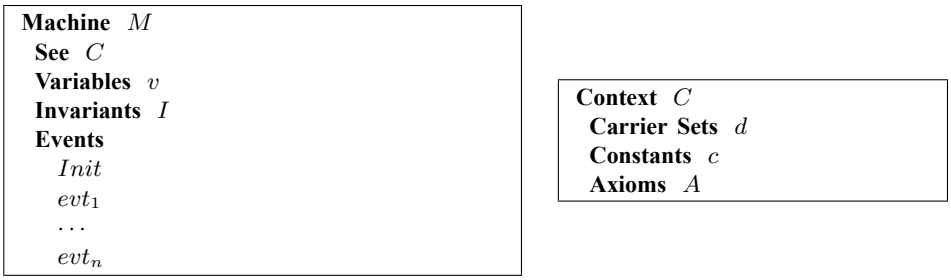
The machine is uniquely identified by its name M. The state variables, v, are declared in the *Variables* clause and initialised in the $Init$ event. The variables are strongly typed by the constraining predicates I given in the *Invariants* clause. The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the *Events* clause. Generally, an event can be defined as follows:

evt $\widehat{=}$ any a where G_e then R_e end

where a is a list of new local variables (parameters), G_e is the event guard, R_e is the event action.

Figure 1 Event-B machine and context



The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A non-deterministic assignment is denoted either as $x :\in S$, where S is a set of values, or $x :| P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to S or according to P.

2.2 Event-B semantics

The semantics of a Event-B model is completed by formulating a number of conditions – *proof obligations*, expressed in the form of logical sequences. Below we describe only several of the most important proof obligations that should be proved for the initial and refined models. The full list of proof obligations can be found in Abrial (2010).

The semantics of Event-B actions is defined using so called *before-after* (BA) predicates (Abrial, 2010). A BA predicate describes a relationship between the system states before and after execution of an event, as shown in Table 1. Here x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after-state.

Table 1 BA predicates

Action (G_e)	$BA(G_e)$
$x := E(x, y)$	$x' = E(x, y) \ \wedge \ y' = y$
$x :\in S$	$\exists z \cdot (z \in S \wedge x' = z) \ \wedge \ y' = y$
$x :\mid P(x, y, x')$	$\exists z \cdot (P(x, z, y) \wedge x' = z) \ \wedge \ y' = y$

The initial Event-B model should satisfy the event *feasibility* and *invariant preservation* properties. For each event of the model, evt_i, its feasibility means that, whenever the event is enabled, its BA predicate is well-defined, i.e., there exists some reachable after-state:

$$A(d,c),\ I(d,c,v),\ g_i(d,c,x,v)\ \vdash\ \exists v' \cdot BA_i(d,c,x,v,v') \qquad \text{(FIS)}$$

where A are model axioms, I are the model invariants, g_i is the event guard, d are model sets, c are model constants, x are the event local variables and v, v' are the variable values before and after the event execution.

Each event evt_i of the initial Event-B model should also preserve the given model invariant I_j:

$$A(d,c),\ I_j(d,c,v),\ g_i(d,c,x,v),\ BA_i(d,c,x,v,v')\ \vdash\ I_j(d,c,v') \qquad \text{(INV)}$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. These new events correspond to stuttering steps that are not visible at the abstract level. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables.

To verify correctness of a refinement step, we need to prove a number of proof obligations for a refined model. For brevity, here we show only a few essential ones.

Let us first introduce a shorthand $H(d,c,v,w)$ to stand for the hypotheses $A(d,c),\ I(d,c,v),\ I'(d,c,v,w)$, where $I,\ I'$ are respectively the abstract and refined invariants, and $v,\ w$ are respectively the abstract and concrete variables. Then the *feasibility refinement* property for an event evt_i of a refined model can be presented as follows:

$$H(d,c,v,w),\ g_i'(d,c,x,w)\ \vdash\ \exists w' \cdot\ BA_i'(d,c,x,w,w') \qquad \text{(REF\_FIS)}$$

where g_i' is the refined guard and BA_i' is a BA predicate of the refined event, and v, w are respectively the abstract and concrete variables.

The event guards in a refined model can be only strengthened in a refinement step:

$$H(d,c,v,w),\ g_i'(d,c,x,w)\ \vdash\ g_i(d,c,x,v) \qquad \text{(REF\_GRD)}$$

where g_i, g_i' are respectively the abstract and concrete guards of the event evt_i.

Finally, the *simulation refinement* property requires to show that the 'execution' of a refined event is not contradictory with its abstract version:

$$H(d,c,v,w),\ g_i'(d,c,x,w),\ BA_i'(d,c,x,w,w')$$

$$\vdash \exists v' \cdot BA_i(d,c,x,v,v')\ \wedge\ I'(d,c,v',w') \qquad \text{(REF\_SIM)}$$

where BA_i, BA_i' are respectively the abstract and concrete BA predicates of the same event evt_i.

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by Rodin Platform (2006). Proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of such complex distributed systems as MASs.

2.3 Modularisation extension

Recently the Event-B language and the tool support have been extended with a possibility to define *modules*. Modules are components containing groups of callable atomic operations (Iliasov et al., 2010; Rodin Modularisation Plug-in, 2010). Modules can have their own (external and internal) state and invariant properties. An important characteristic of modules is that they can be developed separately and, when needed, composed with the main system.

A module description consists of two parts – *module interface* and *module body*. Let M be a module. A module interface MI is a separate Event-B component. A module interface consists of the external module variables v, the external module invariants MI_I, and a collection of module operations, characterised by their pre- and postconditions. The general form of the interface is given in Figure 2.

Figure 2 Event-B interface component

Interface MI
Sees MI_C
Variables v
Invariants MI_I
Initialisation $\cdots$
Process
$P_1 =$ **any** x **where** $G_p(d, c, x, v)$ **then** $R_p(d, c, x, v, v')$ **end**
$\cdots$
Operations
$O_1 =$ **any** b **pre** $Pre(d, c, b, v)$ **post** $Post(c, s, b, v, v)$ **end**
$\cdots$

In addition, a module interface may contain a group of standard Event-B events defined in the *Process* clause. These events describe how the module external variables may change between operation calls.

A formal module development starts with the design of an interface. Once an interface is defined, it cannot be altered in any manner. This ensures that a module body may be constructed independently from a model relying on the module interface. A *module body* is an Event-B machine. It implements the interface by providing a concrete behaviour for each of the interface operations. A set of additional proof obligations are generated to guarantee that each interface operation has a suitable implementation. When the module M is imported into another Event-B machine, the importing machine may invoke the operations of M and read the external variables of M.

A general strategy of a distributed system development in Event-B is to start from an abstract centralised specification and incrementally augment it with design-specific

details. When a suitable level of details is achieved, certain events of the specification are replaced by the calls of interface operations and variables are distributed across modules. As a result, a monolithic specification is decomposed into separate modules. Since decomposition is a special kind of refinement, such a model transformation is also correctness-preserving. Therefore, refinement allows us to efficiently cope with complexity of distributed systems verification and gradually derive an implementation with the desired properties and behaviour.

In the next section, we outline main principles of formal reasoning about MASs and their properties.

3 Formal reasoning about MASs

Let us start by defining a MAS formally.

Definition 1: A MAS $\mathcal{MAS}$ is a tuple $(\mathcal{A}, \mu, \Sigma, \mathcal{E}, \mathcal{R})$, where $\mathcal{A}$ is a collection of different classes of agents, μ is the system middleware, Σ is the system state space, $\mathcal{E}$ is a collection of system events and $\mathcal{R}$ is a set of dynamic relationships between agents in a MAS.

Each agent belongs to a particular class or type of agents $A_i,\ i \in 1..n$ such that $A_i \in \mathcal{A}$. This class is dynamic, i.e., new class agents can spontaneously appear or the existing agents may disappear (both normally or abnormally). A particular agent $a_{ij} \in A_i$ is characterised by its local state consisting from agent variables and static agent attributes.

The system middleware μ can be considered as a special agent that is always present in the system. The responsibility of the middleware is to ensure communication between different agents, to detect appearance of new agents or disappearance (both normal and abnormal) of the existing agents, recover from the situations when the required connections between the agents are lost, etc.

The system state space Σ contains all the possible states of agents and the middleware. The system events $\mathcal{E}$ include all internal and external system reactions. An execution of an event may change the state of the middleware or agents. In other words, each event is associated with the corresponding relation on Σ, i.e., is of the type $\Sigma \leftrightarrow \Sigma$. Each collaborative activity between different agents (or an agent and the middleware) may be composed of a set of events. Moreover, system events may model appearance or disappearance of mobile agents, sending request from one agent to another, recovery of lost agent connections, etc. The collaborative activity should preserve the following property:

Property 1: Let $\mathcal{A}_{act}$ and $\mathcal{A}_{ina}$ be sets of active and inactive agents correspondingly, where $\mathcal{A} = \mathcal{A}_{act} \cup \mathcal{A}_{ina}$ and $\mathcal{A}_{act} \cap \mathcal{A}_{ina} = \varnothing$. Let $\mathcal{EAA}$ and $\mathcal{EA}\mu$ be all the collaborative activities (sets of events) between agents and agents and between agents and middleware respectively. Moreover, for each agent $A \in \mathcal{A}$, let $\mathcal{E}A$ be a set of events in which the agent A is involved. Then

$$\forall A \cdot\ A \in \mathcal{A}_{act} \Rightarrow \mathcal{E}A \in \mathcal{EAA} \cup \mathcal{EA}\mu \quad \text{and} \quad \forall A \cdot\ A \in \mathcal{A}_{ina} \Rightarrow \mathcal{E}A \in \mathcal{EA}\mu.$$

For instance, this property implies that while an agent is recovering from a failure it cannot be involved into cooperative activities with other agents.

A set $\mathcal{R}$ consists of dynamic relationships or connections between active agents of the same or different classes. An agent relationship is modelled as a mathematical relation

$$R(a_1, a_2, ..., a_m) \subseteq C_1^* \times C_2^* ... \times C_m^*,$$

where $a_1, a_2, ..., a_m$ are agents, $C_j^* = C_j \cup \{?\}$, where an agent class $C_j \subset \mathcal{A}_{act}$, $j \in 1..m$. A relationship can be *pending*, i.e., incomplete. This is indicated by question marks in the corresponding places of R, e.g., $R(a_1, a_2, ?, a_4, ?)$. Pending relationships are usually caused by appearance of new agents or disappearance of the existing ones. In addition, an existing agent may initiate a new relationship.

Property 2: Let $\mathcal{A}_{act}$ be a set of active agents. Let $\mathcal{EAA}$ be all the collaborative activities in which these active agents are involved. Moreover, for each agent $A \in \mathcal{A}_{act}$, let $\mathcal{R}_A$ be all the relationships it is involved in. Finally, for each collaborative activity $CA \in \mathcal{EAA}$, let $\mathcal{A}_{CA}$ be a set of all involved agents. Then, for each $CA \in \mathcal{EAA}$ and $A_1, A_2 \in \mathcal{A}_{CA}$,

$$\mathcal{R}_{A_1} \cap \mathcal{R}_{A_2} \neq \varnothing.$$

This property restricts the interactions between the agents – only the agents that are linked by relationships (some of which may be pending) can be involved into cooperative activities.

The system middleware μ keeps track of pending relationships and tries to resolve them by enquiring suitable agents to confirm their willingness to enter into a particular relationships. Additional data structure $Pref_R$ associated with a relationship $R \in \mathcal{R}$ can be used to express a specific preference of one agents over the others. The middleware then enforces this preference by enquiring the preferred agents first. Formally, $Pref_R$ is an ordering relation over the involved agent classes. Thus, for $R \subseteq C_1^* \times ... \times C_m^*$,

$$Pref_R \in C_1 \times ... \times C_m \leftrightarrow C_1 \times ... \times C_m.$$

A responsibility of the middleware is to detect situations when some of the established or to be established relationships become pending and guarantee 'fairness'. Essentially, this means that no pending request is ignored forever and middleware tries to enforce the given preferences, if possible.

While developing a critical MAS, we should ensure that certain cooperative activities, once initiated, are successfully completed. These are the activities that implement safety requirements. To ensure safety we have to verify the following property:

Property 3: Let $\mathcal{EAA}_{crit}$, where $\mathcal{EAA}_{crit} \subseteq \mathcal{EAA}$, be a subset containing critical collaborative activities. Moreover, let $\mathcal{R}_{pen}$ and $\mathcal{R}_{res}$, where $\mathcal{R}_{pen} \subseteq \mathcal{R}$ and $\mathcal{R}_{res} \subseteq \mathcal{R}$, be subsets of pending and resolved relationships defined for these activities. Finally, let $\mathcal{R}_{CA}$, where $CA \in \mathcal{EAA}$ and $\mathcal{R}_{CA} \subseteq \mathcal{R}$, be all the relationships the activity CA can affect. Then, for each activity $CA \in \mathcal{EAA}_{crit}$ and relationship $R \in \mathcal{R}_{CA}$,

$$\Box\ ((R \in \mathcal{R}_{pen}) \rightsquigarrow (R \in \mathcal{R}_{res})),$$

where $\Box$ designates 'always' and $\rightsquigarrow$ denotes 'leads to'.

This property postulates that eventually all pending relationships should be resolved for each critical cooperative activity.

In the next section we will illustrate modelling of MAS in Event-B by a case study – development of a hospital MAS. In particular, we will show how relying on the proposed principles help us to develop a correct system.

4 Modelling of a hospital MAS

4.1 A case study description

The hospital MAS consists of two types of agents – patients and medical personnel (called doctors for simplicity). The condition of each patient is monitored by the corresponding medical equipment – an agent representing a patient. The doctor agents are running on pocket PC-based devices – personal digital assistants (PDA). The hospital provides the wireless connectivity to the doctor agents. Each doctor is associated with one agent. From now on, we will use the terms 'patient' and 'doctor' to designate both agents and people that they represent.

The medical equipment continuously updates the patient's medical record consisting of different medical measurements as well as detects emergencies – dangerous changes of critical parameters (e.g., blood pressure, pulse rate, etc.). In case of an emergency, the patient agent generates an emergency call that is communicated to the doctor(s) treating the patient. An important safety requirement imposed on the system is that *all emergencies should be promptly handled by the responsible doctors*. In spite of its seeming simplicity, this requirement is hard to ensure. Indeed, a MAS operates in a volatile communication environment, i.e., agents might experience temporal disconnections. Hence the design of our system should incorporate certain fault tolerance mechanisms that would guarantee that each emergency call is eventually handled by some doctor. Moreover, different doctors can be associated with the same patient during different shifts. Therefore, we have to ensure that at the end of a doctor's shift all his/her patients are handed over to another doctor.

Another important safety requirement associated with the system is to guarantee that *a doctor always accesses the most recent patient record and the patient's data are always kept in a consistent state*. We assume that the patient record is stored at the equipment associated with her. To ensure that these requirements are satisfied, we should regulate the access to the patient's data. A specific delivery of a medicine, prescription of a treatment and so on are introduced into the patient's log by the medical personnel via their PDAs. To ensure that the data are updated consistently, we only allow the doctor to modify the patient's data when she/he is in a close proximity to the patient. Specifically, the patient data become available at the doctor's PDA only when she/he arrives to the patient location. All the modifications are synchronised with the data stored by the patient's equipment. When the doctor finishes examining the patient or delivering a medicine and leaves, the connection to the patient's data is closed. Such a restriction allows us to ensure that only a doctor who is in a close proximity to a patient is allowed to modify the patient's record. Moreover, it also ensures that the doctor has the access to the freshest information about the patient. This precludes, e.g., a possibility of delivering the medicine twice (modelling the security requirements ensuring patient's data integrity is outside of the scope of this paper).

The safety-critical requirements imposed on the system should be fulfilled in the course of complex agent interactions. Hence, the properties defined in the previous section are put in the context of doctor-patients interactions. Indeed, each patient should be assigned to a certain doctor. While the doctor is inactive (because the shift is over or a failure of the corresponding PDA), doctor-patient interactions cannot occur. Finally, each emergency call should be eventually served.

Next we demonstrate how refinement process in Event-B can facilitate modelling of intertangled agent interactions and verification of their properties.

4.2 Modelling agent interdependencies

The main focus of our development is a specification of collaborative behaviour of patients and doctors in a hospital MAS. We start with an abstract specification. It models the behaviour of the entire hospital MAS in a highly abstract way. We define the variable med_agents to model the active doctor agents:

$$med\_agents \subseteq MEDSTAFF,$$

where $MEDSTAFF$ is a set of all doctor agents.

The events *Activate* and *Deactivate* model joining and leaving hospital location by the doctor agents. While an agent is active, it can perform certain activities, which is abstractly modelled by the event *Activity*. Let us remind, that a middleware is concidered a special agent that is always present in the system.

machine $Hospital$
$Activate \mathrel{\widehat{=}}$
  any ma
  when $ma \in MEDSTAFF \wedge ma \notin med\_agents$
  then $med\_agents := med\_agents \cup \{ma\}$
  end
$Deactivate \mathrel{\widehat{=}}$
  any ma
  when $ma \in med\_agents$
  then $med\_agents := med\_agents \setminus \{ma\}$
  end

In our abstract specification we have merely introduced one type of agents – medical personnel agents. Since these agents perform only engagement and disengagement with the location, they interact with the middleware only.

In our first refinement step we augment our model with a representation of patients and introduce the events that abstractly model interactions between the introduced patients and the doctors. The variable $patients$ defines a set of patients admitted to the hospital:

$$patients \subseteq PATIENTS,$$

where $PATIENTS$ denotes a set of possible patients.

Each patient arriving at the hospital is associated with a doctor who has a primary responsibility for treating the patient. To model this relationship, we introduce the

variable *assigned_doctor*, which is defined as a total function associating hospital patients with the active doctor agents:

$$assigned\_doctor \in patients \rightarrow med\_agents.$$

We omit showing complete specifications of the subsequent refinement steps and merely describe the events and data structures relevant to critical collaborative actions. The complete specifications of the entire development can be found at Pereverzeva et al. (2011).

At the first refinement step, resulting in the model $Hospital1$, we add the new event *PatientArrival* to model a patient arrival:

machine $Hospital1 \mathrel{\widehat{=}}$ **refines** $Hospital$
$PatientArrival \mathrel{\widehat{=}}$
any ma, pa
when $pa \in PATIENTS \wedge pa \notin patients \wedge ma \in med\_agents$
then $patients := patients \cup \{pa\} \parallel assigned\_doctor(pa) := ma$
end

The guard $ma \in med\_agents$ ensures preservation of a specific instance of the *Property 1*: only the active medical agents are assigned to the patient agents. We use $\parallel$ to denote a parallel composition of actions.

The dual event *PatientDischarge* models a patient discharge from the hospital location in a similar way. Let us observe that the event *PatientArrival* ensures the following property, which can be considered as an instance of the *Property 3*:

$\Box$ (new patient $\leadsto$ assigned doctor),

while, similarly, the event *PatientDischarge* ensures that

$\Box$ (a patient leaves the hospital $\leadsto$ all his/her relationships are removed).

In the refined specification we also elaborate on the event *Activity*. Essentially, the medical personnel should examine the patients and deliver the prescribed medicine. We generalise these actions under the general term 'visiting a patient'. In our refined model, we define the variable $visited$ representing a subset of patients that are currently being examined. The new variable $last\_visit$ stores for every patient the id of the last doctor agent that has visited her. The interdependencies between the doctor and patient variables are defined by the following invariants:

$$last\_visit \in patients \nrightarrow MEDSTAFF,$$
$$visited \subseteq patients,$$
$$last\_visit[visited] \subseteq med\_agents,$$
$$visited \subseteq dom(last\_visit).$$

The new events *VisitBegin* and *VisitEnd* refine the abstract event *Activity* and model a visiting procedure. The event *VisitBegin* is specified as follows:

machine $Hospital1 \widehat{=}$ **refines** $Hospital$
$VisitBegin \widehat{=}$ **refines** $Activity$
any ma, pa
when $ma \in med\_agents \wedge pa \in patients \wedge$
$pa \notin visited \wedge ma \notin last\_visit[visited]$
then $last\_visit(pa) := ma \parallel visited := visited \cup \{pa\}$
end

Let us note that the event guard $ma \in med\_agents$ ensures preservation of another instance of the *Property 1*: only active medical agents are eligible to interact with the patient agents.

In our abstract specification we have assumed that the doctor agents can leave the hospital location at any time. However, formal modelling has uncovered that safety cannot be guaranteed under this assumption. Hence we must impose certain restrictions on the situations when the doctors can actually leave the hospital. Before a doctor agent can leave the hospital location, we should reassign his/her patients to another active doctor. This is another illustration of the *Property 3*:

$\Box$ (a doctor leaves the hospital $\leadsto$ all his/her patients are reassigned).

We split the abstract event *Deactivate* into two corresponding events: *AgentLeaving* and *ReassignDoctor*. The event *AgentLeaving* models leaving the hospital location by a doctor, who does not have any assigned patients and is not currently involved in examining a patient.

The event *ReassignDoctor* models leaving the hospital location by a doctor who has assigned patients. We must be sure that all his/her patients will be reassigned to the new active doctor. Here we again illustrate a specific case of the *Property 1*: only active medical agents are assigned to the patient agents. It ensures by the guard $ma\_new \in med\_agents$. The event is modelled as follows:

machine $Hospital1 \widehat{=}$ **refines** $Hospital$
$ReassignDoctor \widehat{=}$ **refines** $Deactivate$
any ma, ma_new
when $ma \in ran(assigned\_doctor) \wedge ma \notin last\_visited[visited] \wedge$
$ma\_new \in med\_agents \wedge ma\_new \neq ma$
then $med\_agents := med\_agents \setminus \{ma\}$
$assigned\_doctor :=$
$assigned\_doctor \Leftarrow (dom(assigned\_doctor \rhd \{ma\}) \times \{ma\_new\})$
end

4.3 Introducing fault tolerance by refinement

In the specification $Hospital1$, while defining the events *AgentLeaving* and *ReassignDoctor*, we have abstracted away from the reasons behind doctor leaving and patient reassignment. Essentially, a doctor agent might leave the location because the doctor's shift is over or because the doctor agent has irrecoverably failed and should be permanently disconnected. At the second refinement step, resulting in the machine $Hospital2$, we introduce a distinction between the normal agent leaving and its disconnection due to a failure. Both cases have a direct impact on the safety

requirements. Next we demonstrate how formal reasoning allows us specify handling of these situations in a correct way.

In a MAS, the agents often lose connection only for a short period of time. After the connection is restored, the agent should be able to continue its activities. Therefore, after detecting a loss of connection, the location should not immediately disengage the disconnected agent but rather set a deadline before which the agent should reconnect. If the disconnected agent restores its connection before the deadline then it can continue its normal activities. However, if the agent fails to do so, the location should permanently disengage the agent. Here we deal with a particular instance of the *Property 3*:

$$\square \text{ (an agent's disconnection happens } \leadsto \text{ agent reconnects activity or disengages from the location).}$$

In the refined specification, we define the variable $disconnected$ representing the subset of active agents that are detected by the location as disconnected:

$$disconnected \subseteq med\_agents.$$

Moreover, to model a timeout mechanism, we define a new variable $timer$ of the enumerated type $\{inactive, active, timeout\}$. Initially, for every active agent, the $timer$ value is set to $inactive$. As soon as an active agent loses connection with the location, its id is added to the set $disconnected$ and its timer value becomes $active$. This behaviour is specified in the new event *DisconnectAgent* as follows:

machine $Hospital2 \mathrel{\widehat{=}}$ **refines** $Hospital1$
$DisconnectAgent \mathrel{\widehat{=}}$
  any ma
  when $ma \in med\_agents \wedge ma \notin disconnected$
  then $disconnected := disconnected \cup \{ma\} \parallel timer(ma) := active$
  end

A temporarily disconnected agent can succeed or fail to reconnect, as modelled by the events *ReconnectionSuccessful* and *ReconnectionFailed* respectively. If the agent reconnects before the value of timer becomes $timeout$, the timer value is changed to $inactive$ and the agent continues its activities virtually uninterrupted. Otherwise, the agent is removed from the set of active agents. The event *ReconnectionSuccessful* is modelled as follows:

machine $Hospital2 \mathrel{\widehat{=}}$ **refines** $Hospital1$
$ReconnectionSuccessful \mathrel{\widehat{=}}$
  any ma
  when $ma \in disconnected \wedge timer(ma) = active$
  then $timer(ma) := inactive \parallel disconnected := disconnected \setminus \{ma\}$
  end

The following invariant ensures that any disconnected agent is considered to be temporarily inactive:

$$\forall ma \cdot (ma \in med\_agents \wedge timer(ma) \neq inactive \Leftrightarrow ma \in disconnected).$$

The introduction of an agent disconnection allows us to make a distinction between two reasons behind leaving the location by a doctor – because of the end of shift or due to the disconnection timeout. To model these two cases, we split the event *AgentLeaving* into two events *NormalAgentLeaving* and *DetectFailedFreeAgent* respectively.

While modelling failure of a doctor agent, we should again deal with the *Property 3*:

□ (a doctor abnormally disconnects $\leadsto$ all his/her patients are reassigned).

In both cases we need to reassign all the patients of the disconnected doctor to another active doctor. In a similar way as above, the event *ReassignDoctor* is decomposed into two events *NormalReassignDoctor* and *DetectFailedAgent*. The event *DetectFailedAgent* is specified as follows:

machine $Hospital2 \mathrel{\widehat{=}}$ **refines** $Hospital1$
$DetectFailedAgent \mathrel{\widehat{=}}$ **refines** $ReassignDoctor$
any ma, ma_new
when $ma \in ran(assigned\_doctor) \land ma \notin last\_visited[visited] \land$
$ma\_new \in med\_agents \land$
$ma\_new \neq ma \land ma \in disconnected \land timer(ma) = timeout \land$
$(ma\_new \notin disconnected \lor (ma\_new \in disconnected \land timer(ma\_new)$
$= active))$
then $med\_agents := med\_agents \setminus \{ma\}$
$assigned\_doctor :=$
$assigned\_doctor \Leftarrow (dom(assigned\_doctor \rhd \{ma\}) \times \{ma\_new\})$
$disconnected := disconnected \setminus \{ma\} \parallel timer := \{ma\} \lhd timer$
end

The second refinement step has resulted in a specification ensuring that no patients are left unattended neither because of the doctors shift change nor because of a doctor agent failure.

4.4 Ensuring safety of cooperative agent actions

Our next refinement step introduces abstract modelling of emergency calls, which are generated by a patient monitoring equipment. We must guarantee that each call will be properly handled by a corresponding doctor. Hence our next refinement step should transform the specification to ensure the following property:

□ (an emergency call by a patients $\leadsto$ a medical visit happens).

In this refinement step, we introduce the variable $emergency\_calls$, which is defined as a partial function associating the emergency calls with the patients:

$$emergency\_calls \in ALARMS \nrightarrow patients.$$

Moreover, we define the variable $accepted\_calls$ that establishes the correspondence between the emergency calls and the doctors that answer them:

$$accepted\_calls \in ALARMS \nrightarrow med\_agents.$$

At this refinement step we abstract away from an actual implementation of how a doctor handling an emergency call is chosen. A detailed model of this will be introduced at the next refinement step. Here we add the new event *EmergencyCall* to abstractly model the occurrence of an emergency call ec:

machine $Hospital3 \mathrel{\widehat{=}}$ **refines** $Hospital2$
$EmergencyCall \mathrel{\widehat{=}}$ **status** $convergent$
  any pa, ec
  when $pa \in patients \wedge ec \in ALARMS \wedge$
    $ec \notin dom(emergency\_calls) \wedge pa \notin ran(emergency\_calls)$
  then $emergency\_calls := emergency\_calls \cup \{ec \mapsto pa\}$
  end

In its turn, the new event HandlingEmergencyCall abstractly models finding a responsible doctor to handle the call ec:

$HandlingEmergencyCall \mathrel{\widehat{=}}$ **status** $convergent$
  any ec, ma
  when $ec \in dom(emergency\_calls) \wedge ec \notin dom(accepted\_calls) \wedge$
    $ma \in med\_agent \wedge ma \notin disconnected$
  then $accepted\_calls := accepted\_calls \cup \{ec \mapsto ma\}$
  end

To guarantee that the refined specification preserves the global behaviour of the abstract machine, the developer should demonstrate that the newly introduced events *converge*. To prove it, the developer should define a variant – a natural number expression on the model variables – and show that the execution of any of these events decreases it.

In our case, we define a specific system variant to ensure that the newly introduced events *EmergencyCall* and *HandlingEmergencyCall* do not take the control forever. Those events have status *convergent*. We define the variant as follows:

$$card(ALARMS \setminus dom(emergency\_calls)) + card(ALARMS \setminus dom(accepted\_calls)),$$

and prove that it is decreased by the new events. Indeed, when a call arrives, $dom(emergency\_calls)$ increases. This in turn decreases the value of $card(ALARMS \setminus dom(emergency\_calls))$ and, consequently, the whole variant expression. On the other hand, when a certain call has been accepted, the value of $card(ALARMS \setminus dom(accepted\_calls))$ decreases and, consequently, the whole variant expression decreases as well.

Variants play an important role in guaranteeing, e.g., that error recovery terminates, service requests are eventually served, etc. In our case, the variant allows us to ensure that eventually some doctor is chosen to handle an emergency. Obviously, this has important safety implications.

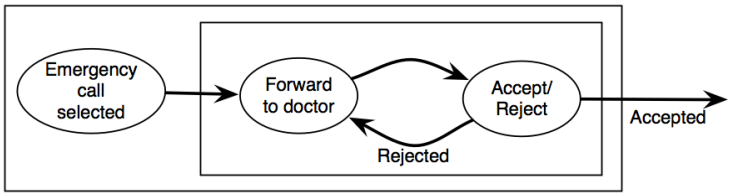
Moreover, at this refinement step, we also distinguish two types of a patient visit – a regular visit (a scheduled examination or a delivery of a medicine) and a visit for handling an emergency call. To model this, we decompose the event *VisitBegin* into the events *RegularVisitBegin* and *EmergencyVisitBegin*. The event *EmergencyVisitBegin* is specified as follows:

machine $Hospital3 \widehat{=}$ **refines** $Hospital2$
$EmergencyVisitBegin \widehat{=}$ **refines** $VisitBegin$
any ec
when $ec \in dom(accepted\_calls) \wedge emergency\_calls(ec) \notin visited \wedge$
$accepted\_calls(ec) \notin last\_visit[visited]$
with $ma = accepted\_calls(ec)$
$pa = emergency\_calls(ec)$
then $last\_visit(emergency\_calls(ec)) := accepted\_calls(ec)$
$visited := visited \cup \{emergency\_calls(ec)\}$
$emergency\_calls := emergency\_calls \setminus \{ec \mapsto emergency\_calls(ec)\}$
$accepted\_calls := accepted\_calls \setminus \{ec \mapsto accepted\_calls(ec)\}$
end

The proof of the refinement is often supported by supplying witnesses – the concrete values for the replaced abstract variables and parameters. Witnesses are specified in the event clause *with*. In our case, in the $EmergencyVisitBegin$ event, the abstract parameters ma and pa are replaced by the concrete values $accepted\_calls(ec)$ and $emergency\_calls(ec)$ respectively.

In the current refinement step we have introduced modelling of emergency calls and non-deterministic assignment of responsible doctors to handle them. The goal of our next refinement step is to introduce a detailed procedure of selecting a doctor in the case of an emergency call. It follows the steps graphically depicted in Figure 3.

Figure 3 Procedure of choosing a doctor for a certain call



The proposed procedure can be described as follows. We start by selecting an emergency call to handle. Then we model a loop of finding a suitable candidate and sending a request to him/her. If the doctor rejects it then we choose the next candidate. The procedure is repeated until we get an acceptance of the request.

To model the described procedure, we introduce a number of new variables and events to specify the corresponding steps of the selection procedure. The event *ChooseCurrentCall* models the beginning of handling of a particular emergency call. The event *CallFeed* directs the call to the assigned doctor:

machine $Hospital4 \widehat{=}$ **refines** $Hospital3$
$CallFeed \widehat{=}$
when $ec\_handling = TRUE \wedge candidate\_found = FALSE \wedge$
$assigned\_doctor(emergency\_calls(current\_call)) \notin disconnected \wedge$
$assigned\_doctor(emergency\_calls(current\_call)) \notin occupied$
then $directed(current\_call) := assigned\_doctor(emergency\_calls(current\_call))$
$candidate\_found := TRUE$
end

The event *ForwardCall* forwards the call to next suitable candidate:

$ForwardCall \mathrel{\widehat{=}}$
  any ma_new
  when $ec\_handling = TRUE \wedge candidate\_found = FALSE \wedge$
    $(assigned\_doctor(emergency\_calls(current\_call)) \in disconnected \vee$
    $assigned\_doctor(emergency\_calls(current\_call)) \in occupied) \wedge$
    $ma\_new \in med\_agents \wedge ma\_new \notin disconnected \wedge ma\_new \notin occupied$
  then $directed(current\_call) := ma\_new \parallel candidate\_found := TRUE$
  end

Let us note that a doctor agent can refuse to accept a call. For instance, by checking the doctor schedule it might discover that he/she is currently performing a scheduled surgery. The events *AcceptCall* and *RejectCall* model acceptance and rejection of the call respectively. The variable $occupied$ is used to accumulate the id's of doctors that have already refused the call. As soon as we find a suitable candidate to serve the call, we refresh the variable $occupied$ for the next call, i.e., $occupied := \varnothing$. The event *AcceptCall* is specified as follows:

machine $Hospital4 \mathrel{\widehat{=}}$ **refines** $Hospital3$
$AcceptCall \mathrel{\widehat{=}}$ **refines** $HandlingEmergencyCall$
  when $ec\_handling = TRUE \wedge candidate\_found = TRUE \wedge$
    $current\_call \in dom(emergency\_calls) \wedge current\_call \notin dom(accepted\_calls) \wedge$
    $directed(current\_call) \notin disconnected$
  with $ec = current\_call$
    $ma = directed(current\_call)$
  then $accepted\_calls(current\_call) := directed(current\_call) \parallel$
    $ec\_handling := FALSE$
    $candidate\_found := FALSE \parallel occupied := \varnothing$
  end

We define a special event $ForcedAcceptCall$ to 'force' the last available doctor to accept the call. Feasibility of such a restriction to have only one available doctor agent in the hospital location to serve the call can be checked probabilistically.

Moreover, we assume here that the whole procedure of finding a doctor in respond to a certain emergency call takes a short period of time and during this period no disconnection of agents can occur. As a result, we strengthen the guards in the event $DisconnectAgent$ to disallow any disconnection while an emergency call is handled.

While verifying correctness of this refinement step, we encounter a problem with the system requirements – we cannot guarantee safety unless we assume that during the search for a doctor no disconnection of agents can occur. Hence, our system should ensure (e.g., by implementing a certain protocol) that finding a doctor takes a very short period of time. Moreover, we define an additional system variant – $card(med\_agents \setminus occupied)$ – to ensure that the event *RejectCall* is convergent, which means that eventually we should get an acceptance from a doctor to serve the call.

4.5 Data integrity

To ensure that a patient gets a correct treatment, we should guarantee that the medical personnel always access the most recent patient record. As we discussed in Section 4.1, we allow a doctor to access and modify the patient's data only when she/he is in a close proximity to the patient. We implement this requirement via the scoping mechanism

(Laibinis et al., 2006, 2009). A *scope* provides a shared data space for a doctor and a patient. We assume that each patient agent has the scope associated with it. As soon as a doctor agent appears at a close vicinity of the patient agent, it automatically joins the scope. While in the scope, the doctor can modify the patient record (e.g., prescribe a new medicine, log the information about the delivered medicine, prescribe a new procedure, etc.).

To model this behaviour, we refine the abstract events *RegularVisitBegin*, *EmergencyVisitBegin*, *VisitEnd* by the new events *RegularEnterScope*, *EmergencyEnterScope*, *LeaveScope* and add a new event *ModifyRecord*. Moreover, we introduce the variable $scopes$, which is defined as a partial function associating the active scopes with the doctors participating in them:

$$scopes \in ScopeName \rightarrowtail med\_agents.$$

Furthermore, we define the model invariant to illustrate preservation of the *Property 1*: only active medical agents are eligible to enter the scope:

$$\forall ma \cdot ma \in disconnected \Rightarrow ma \notin ran(scopes).$$

Also we introduce the variable $record$ that represents the medical history for every patient:

$$record \in patients \rightarrow \mathbb{P}(DATA).$$

The variable ma_data stores the data that appear on the doctor's PDA screen:

$$ma\_data \in med\_agents \nrightarrow \mathbb{P}(DATA).$$

When the doctor agent is in a close vicinity of a patient, its ma_data becomes equal to the value of the patient data. The event *ModifyRecord* models an update of the patient record by a doctor, when she/he is in the scope of a patient:

machine $Hospital5 \mathrel{\widehat{=}}$ **refines** $Hospital4$
$ModifyRecord \mathrel{\widehat{=}}$
  any ma, sn, pa, da_new
  when $(sn \mapsto ma) \in scopes \wedge pa \in dom(last\_visit) \wedge pa \in visited \wedge$
    $last\_visit(pa) = ma \wedge da\_new \in \mathbb{P}(DATA) \wedge da\_new \neq \varnothing$
  then $ma\_data(ma) := da\_new \parallel record(pa) := da\_new$
  end

The corresponding safety property stating that the medical personnel always access the most recent record is formulated as the model invariant:

$$\forall ma, pa \cdot (pa \mapsto ma) \in (visited \lhd last\_visit) \Rightarrow ma\_data(ma) = record(pa).$$

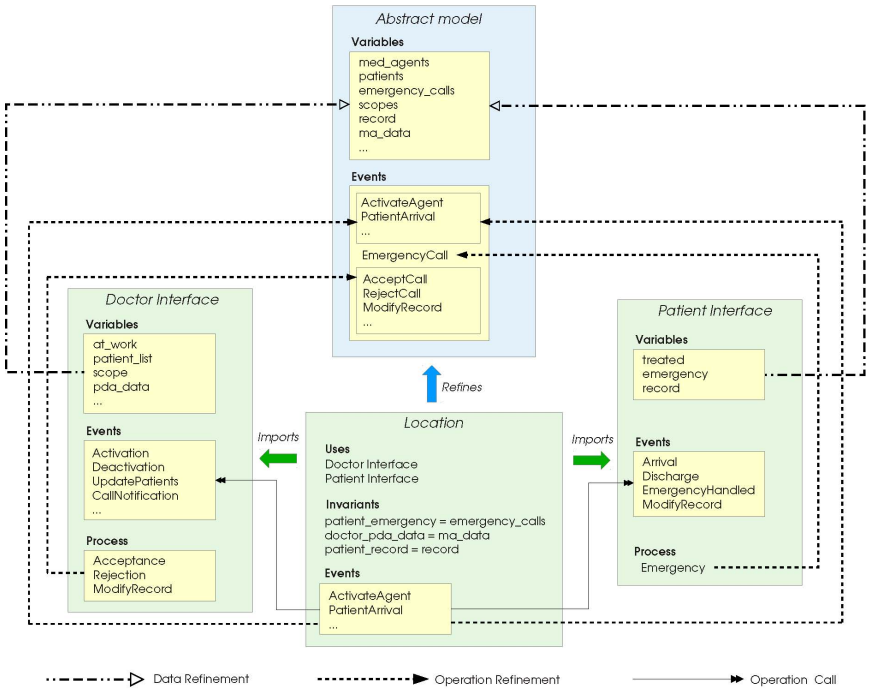
4.6 Decomposition

As a result of our previous refinement steps, we have arrived at a centralised model – $Hospital5$ of the Hospital MAS. In the final refinement step we aim at deriving a distributed architecture. In reality, the agents and the location communicate with each other via message passing. This refinement step is focused on defining this

communication explicitly. To achieve this, we rely on the modularisation extension of Event-B. Our goal is to decompose $Hospital5$ model into separate modules representing the middleware, the doctor and patient agents and explicitly model communication between them.

A graphical representation of the system after decomposition refinement is shown in Figure 4. The abstract model is refined by a model representing the *Location* part – the middleware, and two separate modules – *Doctor* and *Patient*. Each modules contain callable operations and both internal and external data. The *Location* part accesses the modules via the provided generic interfaces. All the variables and most of operations of the interface corresponding to the doctor and patient agents appear with the prefixes $doctor$ and $patient$ respectively. This is a feature of the modularisation extension that helps to avoid name clashes when importing several interfaces.

Figure 4 Illustration of the decomposition refinement (see online version for colours)



The *Location* model imports two module interfaces – *Doctor* and *Patient*. The majority of the events of the machine *Hospital5* are now refined by the location events. The remaining events, including *AcceptCall*, *RejectCall*, *EmergencyCall* become a part of the autonomous processes of the doctor or patient modules. The abstract event *EmergencyCall* is refined by the patient's interface event *Emergency* implementing the occurrence of emergency calls. Similarly, the abstract events *AcceptCall*, *RejectCall*, *ModifyRecord* are refined by the corresponding doctors's interface events *Acceptance*, *Rejection* and *ModifyRecord*.

The variables of the centralised model *Hospital5* are now refined by the variables of the *Location*, *Doctor* and *Patient* modules. For instance, the abstract variable $emergency\_calls$ is replaced by a new variable from the patient interface $patient\_emergency$. Similarly, the abstract variable ma_data is refined by a new

variable from the doctor interface $doctor\_pda\_data$. An extract from the *PatientInterface* is shown below.

Interface *PatientInterface*
Variables $treated, emergency, record$
Invariants $inv_1 : treated \in PATIENTS \to BOOL$
$inv_2 : emergency \in ALARMS \nrightarrow PATIENTS$
$inv_3 : record \in PATIENTS \nrightarrow \mathbb{P}(DATA)$
Process
Emergency
any pa, ec
when $pa \in dom(treated) \land treated(pa) = TRUE \land ec \in ALARMS \land$
$ec \notin dom(emergency) \land pa \notin ran(emergency)$
then $emergency := emergency \cup \{ec \mapsto pa\}$ **end**
Operations
Arrival
any pa, da
pre $pa \in PATIENTS \land treated(pa) = FALSE \land da \subseteq DATA$
return $void$
post
$void' \in VOID \land treated' = treated \lhd\!\!+ \{pa \mapsto TRUE\} \land$
$record' = record \lhd\!\!+ \{pa \mapsto da\}$
end

Let us describe communication between the agents by an example of a patient arrival. At our previous refinement step this activity has been modelled by the event *PatientArrival*. Essentially, a patient arrival corresponds to registering a patient by the location (middleware). By performing certain internal computations, the location assigns a doctor to a patient and confirms a successful registration by sending the corresponding messages to the patient and the doctor agents. This behaviour is specified in the refined event *PatientArrival*:

$PatientArrival \mathrel{\widehat{=}}$ **Refines** $PatientArrival$
any ma, pa, da
when $pa \in PATIENTS \land pa \notin patients \land ma \in med\_agents \land da \subseteq DATA$
then
$patients := patients \cup \{pa\} \parallel assigned\_doctor(pa) := ma$
$void := patient\_Arrival(pa \mapsto da) \parallel void2 := doctor\_UpdatePatients(ma \mapsto \{pa\})$
end

The calls to the corresponding operations from the patient and doctor interfaces, $patient\_Arrival(pa \mapsto da)$ and $doctor\_UpdatePatients(ma \mapsto \{pa\})$, model the communication between the location and agents respectively. The other events are decomposed in a similar way.

As a result of the decomposition we arrive at a specification of distributed hospital MAS. Each type of agents – doctors and patients – are represented by the corresponding module. The location serves as a communication infrastructure.

To verify correctness of the models, we discharged more than 500 proof obligations. Around 85% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment.

5 Conclusions and related work

5.1 Conclusions

In this paper we have presented a formal approach to developing MAS by refinement in Event-B. We formalised the main properties of MAS and demonstrated how refinement process can facilitate their preservation. Finally, we illustrated the proposed approach by a large case study – development of a hospital MAS. In our case study we focused on modelling and verification of safety of two central critical activities – handling emergencies and consistent updates of patient data. Ensuring correctness of these activities was especially challenging due to highly dynamic nature of a hospital, volatile error-prone communication environment and autonomous agent behaviour.

In our development we have explicitly modelled the fault tolerance mechanism that guarantee correct system functioning in the presence of agent disconnections. We have verified by proofs the correctness and safety of these two activities. Formal verification process has not only allowed us to systematically capture complex requirements but also facilitated derivation of the constraints that should be imposed on the system to guarantee its safety. Indeed, for instance, while proving convergence of the emergency handling procedure, we had to explicitly state the assumptions that the system must fulfil. These assumptions can be seen as a contract that should be checked during system deployment to guarantee its safety. In our development we have also demonstrated that the scoping mechanism provides a useful abstraction for ensuring consistent updates of the patient data.

5.2 Related work

The work presented in this paper is inspired by our previous work on modelling context-aware mobile agent systems (Laibinis et al., 2006, 2009) in the CAMA framework (Iliasov and Romanovsky, 2005; Iliasov et al., 2006). In a similar way, we rely on a timeout mechanism to tolerate agent disconnections and employ a scoping mechanism to provide shared data space for patient and doctor agents. However, in this paper we have focused on modelling and verification of safety properties of complex agent interactions rather than on reasoning about general mechanisms for agent interaction with middleware.

A large number of techniques for modelling distributed systems has been proposed in the last decade. For instance, in Schaible and Gotzhein (2003) and Fliege et al. (2005) the authors have proposed an approach to developing distributed systems based on SDL. SDL is a standardised language used for the description of architecture, behaviour, data and static interfaces (ETSI: World Class Standards, 2011). The main advantage of our approach over SDL is that it employs proof-based verification techniques.

Another technique for modelling distributed systems together with its tool support (DisCo) have been presented in Katara and Mikkonen (2001) and Aaltonen et al. (2001). The proposed approach based on temporal logic incorporates a specification language as well as a methodology and a graphic tool support for developing system specifications. DisCo also supports composition and refinement techniques. The weakness of the DisCo toolset is that it does not support verification, proposing to use instead a number of more general purpose verification tools.

Formal modelling of MAS has been undertaken by Roman et al. (2007, 2004, 1997). The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. In our approach we also have studied the problem of ensuring access to the fresh context. However, in Roman et al. (2007) it is solved at the level of the matching agent attributes while in our approach we rely on the scoping mechanism to achieve this.

A formal modelling of MAS for the healthcare in Z has be undertaken by Gruer et al. (2002). The work has focused on specifying a MAS for a medical help system. The authors aimed at studying how to formally represent agent interactions, e.g., during negotiations. In our approach we not only model the agent interactions but also formally prove their properties. Hence, our approach is especially suitable for developing critical MAS.

A problem of modelling fault tolerance MAS has been addressed by Ball and Butler (2009). They illustrated how certain typical fault tolerance mechanisms can be incorporated into a MAS specification. In our approach we consider fault tolerance as a part of ensuring safety of MAS.

Our approach is different from numerous process-algebraic approaches used for modelling MAS. Firstly, we have relied on proof-based verification that does not impose restrictions on the size of the model, number of agents, etc. Secondly, we have adopted a system's approach, i.e., we modelled the entire system and extracted specifications of its individual components by decomposition. Such an approach allows us to express and formally verify safety of the overall system, i.e., we achieve verification of safety as a *system level* property. Finally, the adopted top-down development paradigm has allowed us to efficiently cope not only with complexity of requirements but also with complexity of verification. We have build a large formal model of a complex system by a number of rather small increments. As a result, verification efforts have been manageable because we merely needed to prove refinement between each two adjacent levels of abstraction. Hence, we conclude that refinement in Event-B constitutes a suitable technique for formal modelling and verification of critical MASs.

As a future work we are planning to generalise the proposed approach and extract modelling patterns that can be used to automate formal development. Moreover, we will investigate how to model adaptive agent behaviour that depends on the surrounding context and various reconfiguration mechanisms.

References

Aaltonen, T., Katara, M. and Pitkänen, R. (2001) 'DisCo toolset – the new generation', *Journal of Universal Computer Science*, Vol. 7, No. 1, pp.3–18.

Abrial, J-R. (2005) *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, UK.

Abrial, J-R. (2010) *Modeling in Event-B*, Cambridge University Press, UK.

Ball, E. and Butler, M. (2009) 'Event-B patterns for specifying fault-tolerance in multi-agent interaction', in Butler, M., Jones, C. and Romanovsky, A. (Eds.): *Methods, Models and Tools for Fault Tolerance*, Vol. 5454 of *LNCS*, pp.104–129, Springer.

ETSI: World Class Standards (2011) [online] http://www.etsi.org/WebSite/technolo-gies/sdl.aspx (accessed 30 September 2012).

Fliege, I., Geraldy, A., Gotzhein, R., Kuhn, T. and Webel, C. (2005) 'Developing safety-critical real-time systems with SDL design patterns and components', *Computer Networks*, Vol. 49, No. 5, pp.689–706.

Gruer, P., Hilaire, V., Koukam, A. and Cetnarowicz, K. (2002) 'A formal framework for multi-agent systems analysis and design', *Expert Systems with Applications*, Vol. 23, No. 4, pp.349–355.

Iliasov, A. and Romanovsky, A. (2005) 'CAMA: structured coordination space and exception propagation mechanism for mobile agents', in *Proceedings of ECOOP 2005, Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions*.

Iliasov, A., Khomenko, V., Koutny, M. and Romanovsky, A. (2006) 'On specification and verification of location-based fault tolerant mobile systems', in Butler, M., Jones, C., Romanovsky, A. and Troubitsyna, E. (Eds.): *Rigorous Development of Complex Fault-Tolerant Systems*, Vol. 4157 of *LNCS*, pp.168–188, Springer.

Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi K., Ilic, D. and Latvala, T. (2010) 'Supporting reuse in Event B development: modularisation approach', in *Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ 2010)*, Springer, Vol. 5977 of *LNCS*, pp.174–188.

Katara, M. and Mikkonen, T. (2001) 'Aspect-oriented specification architectures for distributed real-time systems', in *Proceedings of the ICECCS 2001*, IEEE Computer Society, pp.180–190.

Laibinis, L., Troubitsyna, E., Iliasov A. and Romanovsky, A. (2006) 'Rigorous development of fault-tolerant agent systems', in Butler, M., Jones, C., Romanovsky, A. and Troubitsyna, E. (Eds.): *Rigorous Development of Complex Fault-Tolerant Systems*, Vol. 4157 of *LNCS*, pp.241–260, Springer.

Laibinis, L., Troubitsyna, E., Iliasov, A. and Romanovsky, A. (2009) 'Fault tolerant middleware for agent systems: a refinement approach', in *Proceedings of EWDC 2009, European Workshop on Dependable Computing*.

OMG Mobile Agents Facility (1997) [online] http://www.omg.org (accessed 11 November 2011).

Pereverzeva, I., Troubitsyna, E. and Laibinis, L. (2011) 'Rigorous development of a safe multi-agent system', Technical Report 1004, Turku Centre for Computer Science.

Rodin Modularisation Plug-in (2010) [online] http://wiki.event-b.org/index.php/Modularisation_Plug-in (accessed 11 November 2011).

Rodin Platform (2006) [online] http://www.event-b.org/ (accessed 11 November 2011).

Roman, G-C., McCann, P. and Plun, J. (1997) 'Mobile UNITY: reasoning and specification in mobile computing', *ACM Transactions of Software Engineering and Methodology*, Vol. 6, No. 3, pp.250–282.

Roman, G-C., Julien, Ch. and Payton, J. (2004) 'A formal treatment of context-awareness', in *FASE 2004*, Springer, Vol. 2984 of *LNCS*, pp.12–36.

Roman, G-C., Julien, Ch. and Payton, J. (2007) 'Modeling adaptive behaviors in context UNITY', *Theoretical Computer Science*, Vol. 376, No. 3, pp.185–204.

Schaible, P. and Gotzhein, R. (2003) 'Development of distributed systems with SDL by means of formalized APIs', in *SDL Forum*, Springer, Vol. 2708 of *LNCS*, pp.317–334.

Paper V

Formal Goal-Oriented Development of Resilient MAS in Event-B

Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis

Originally published in: Mats Brorsson, Luis Miguel Pinho (Eds.), *Proceedings of 17th International Conference on Reliable Software Technologies (Ada-Europe 2012)*, LNCS 7308, 147–161, Springer-Verlag Berlin Heidelberg, 2012.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-30598-6_11

Formal Goal-Oriented Development of Resilient MAS in Event-B

Inna Pereverzeva[1,2], Elena Troubitsyna[2], and Linas Laibinis[2]

[1] Turku Centre for Computer Science
[2] Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku, Finland
{inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi

Abstract. Goal-Oriented Development facilitates structuring complex requirements. To ensure resilience the designers should guarantee that the system achieves its goals despite changes, e.g., caused by failures of system components. In this paper we propose a formal goal-oriented approach to development of resilient MAS. We formalize the notion of goal and goal achievement in Event-B and propose the specification and refinement patterns that allow us to guarantee that the targeted goals are reached despite agent failures. We illustrate our approach by a case study – development of an autonomous multi-robotic system.

Keywords: Event-B, formal modelling, refinement, goal-oriented development, multi-agent system.

1 Introduction

Goal-Oriented Development [5] has been recognised as an useful framework for structuring and specifying complex system requirements. In goal-oriented development, the system requirements are defined in terms of goals – the functional and non-functional objectives that a system should achieve. Often changes in system operational environment, e.g., caused by failures of agents – independent system components of various types – might hinder achieving the desired goals. Hence, to ensure system resilience [7], i.e., guarantee its dependability in spite of the changes, we need formally verify reachability of the targeted goals. Traditionally, such a verification is undertaken by abstracting implementation up to requirements level and model-checking satisfiability of goals. However, such an approach suffers from a state explosion that is especially prohibitive for such applications as multi-robotic systems [15].

In this paper we propose a formal development approach that ensures goal reachability "by construction". Our approach is based on refinement in Event-B. Event-B [2] is a formal top-down development approach to correct-by-construction system development. The main development technique – refinement – allows us to ensure that a concrete specification preserves globally observable behaviour and properties of abstract specification. Verification of each refinement step is done by proofs. The Rodin platform [11] automates modelling and verification

in Event-B. Currently Event-B is actively used within EU project DEPLOY [4] to model dependable systems from various domains.

We formalise goal-oriented development by defining a set of specification and refinement patterns. Our formalisation reflects the main concepts of the goal-oriented engineering. In particular, we demonstrate how to define system goals at different levels of abstraction and guarantee goal reachability while specifying collaborative agent behaviour. Moreover, we propose refinement patterns that allow the system to dynamically reallocate goals from failed agents to healthy ones and per se, guarantee resilience. A development of an autonomous multi-robotic system illustrates application of the proposed patterns. We believe that our approach offers a scalable technique for development and formal verification of complex resilient multi-agent systems (MAS).

The paper has the following structure. In Section 2 we briefly present our modelling framework – Event-B. In Section 3 we present the set of specification and refinement patterns that facilitate goal-oriented development in Event-B. In Section 4 we present a case study – development of an autonomous multi-robotic system by refinement. In Section 5 we overview the related work, discuss the presented approach and outline the directions for the future research.

2 Formal Modelling and Refinement in Event B

In this section we present our formal development framework – Event-B. The Event-B formalism is an extension of the B Method [1]. It is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems.

2.1 Modelling in Event-B

In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the system state represented as a collection of model variables, and defines operations on this state, i.e., it describes the dynamic *behaviour* of the modelled system. A machine may also have the accompanying component, called *context*. A context might include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. In Event-B, the variables are strongly typed by the constraining predicates called **invariants**. Moreover, the invariants specify important properties that should be preserved during the system execution.

The dynamic behaviour of the system is defined by the set of atomic **events**. Generally, an event can be defined as follows:

$$\textbf{evt} \mathrel{\widehat{=}} \textbf{any } vl \textbf{ where } g \textbf{ then } S \textbf{ end}$$

where vl is a list of new local variables (parameters), g is the event **guard**, and S is the event **action**. The guard is a state predicate that defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments.

2.2 Event-B Refinement

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that non-deterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce non-determinism and introduce detailed design decisions. In particular, we can replace abstract variables by their concrete counterparts, i.e., perform data refinement. In this case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables. Via such a *gluing* invariant we establish a correspondence between the state spaces of the refined and the abstract machines.

Often a refinement step introduces new events and variables into the abstract specification. The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit *skip*. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events *converge*. To prove it, we need to define a *variant* – an expression over a finite subset of natural numbers – and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. Then the event obtains the status *anticipated*. This obliges the designer to prove at some later refinement step, that the event indeed converges.

Each refinement step requires to verify a number of proof obligations that ensure that the refined specification adheres to its abstract counterpart [2]. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the Rodin platform [11].

Refinement and proof-based verification of Event-B offers the designers a scalable support for the development of such complex systems as multi-agent systems (MAS). MAS are decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programs acting autonomously on behalf of a person or organisation, while coordinating their activities by communication [9]. MAS are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas, etc. In the next section we show how refinement process can facilitate modelling MAS and reasoning about goal reachability.

3 A Formal View of Goal-Oriented Multi-Agent System

3.1 Patterns for Goal-Oriented Development

The goal-oriented engineering facilitates structuring complex system requirements in terms of *goals* – objectives that the system should meet [5]. In this paper we focus on modelling functional goals, i.e., the goals defining objectives of the services that the system should deliver. We propose a number of *specification and refinement patterns* that interpret essential activities of goal-oriented engineering in terms of Event-B refinement.

A pattern in Event-B is an abstract machine that defines a generic modelling solution that can be reused in similar developments via instantiation. Usually, an Event-B pattern contains abstract types, constants and variables. The context of

such a model constraints the instantiation by defining the properties that should be satisfied by concrete instantiations of abstract data structures. The invariant properties of a pattern, once proven, remain valid for all instantiations.

The aim of defining a pattern is to capture experience gained in modelling a certain problem. To illustrate how patterns are defined, let us now present a pattern that allows the designers to explicitly define goals while modelling a system in Event-B. We call it *Abstract Goal Modelling Pattern*.

3.2 Abstract Goal Modelling Pattern

Let $GSTATE$ be an abstract type defining the system state space[3]. Moreover, let *Goal* be a non-empty proper subset of $GSTATE$ that abstractly defines the given system goals. We say that the system has achieved the desired goals if its current state belongs to *Goal*. Both $GSTATE$ and *Goal* are the abstract types. Together with their properties they are defined in the model context as follows:

$$Goal \neq \varnothing \quad \text{and} \quad Goal \subset GSTATE.$$

Let us note that *GSTATE* and *Goal* are generic parameters of the initial pattern. During a system development, we should supply their concrete instantiations that satisfy the properties shown above.

While modelling a system in Event-B, we should ensure that the system under development achieves the desired goal. We can formally express this by requiring that the system terminates in a state belong to *Goal*. The machine M_AGM is defined according to the *Abstract Goal Modelling Pattern*:

```
Machine M_AGM
Variables gstate
Invariants gstate ∈ GSTATE
Events
  Initialisation ≙
    begin
      gstate :∈ GSTATE \ Goal
    end
  Reaching_Goal ≙
    status anticipated
    when
      gstate ∈ GSTATE \ Goal
    then
      gstate :∈ GSTATE
    end
end
```

The dynamic behaviour of the system is abstractly modelled by the event Reaching_Goal. The system terminates when Reaching_Goal becomes disable, i.e., when a state satisfying *Goal* is reached.

The event Reaching_Goal has the status *anticipated*. Hence, in the machine M_AGM goal reachability is postulated rather than proved. However, it also obliges us to prove (at some refinement step) that the event or its refinements converge. Therefore, while refining a concrete specification defined according to *Abstract Goal Modelling Pattern*, we will be forced to prove goal reachability.

Let us assume that we have a collection of Event-B patterns: P_1, P_2, ..., P_n that refine each other in the following way:

$$P_1 \textbf{ is refined by } P_2 \textbf{ ... is refined by } P_n.$$

Such a refinement chain expresses a generic development by refinement. Abstract data structures of all the involved patterns become generic parameters of the development. Each pattern abstractly defines a solution for specifying a certain modelling aspect. Therefore, each refinement step has a rationale behind it – its meta-level description. We use it to formulate modelling aspects

[3] In fact, it is sufficient to consider the states that our goal depends on.

that the refinement transformation aims at defining. The result of refinement transformation is called a refinement pattern.

Next we propose several refinement patterns that allow us to implement the ideas of goal-oriented engineering in Event-B refinement. We start from defining *Goal Decomposition Pattern*.

3.3 Goal Decomposition Pattern

The main idea of goal-oriented development is to decompose the high-level system goals into a set of subgoals. This is an iterative process that aims at building the hierarchy of system goals. Essentially, subgoals define intermediate stages of the process of achieving the main goal.

The purpose of *Goal Decomposition Pattern* is to explicitly model subgoals in the system specification. While defining this pattern, we should ensure that high-level goals remain achievable. Hence our refinement pattern should reflect the relation between the high-level goals and subgoals. Moreover, it should ensure that high-level goal reachability is preserved and can be defined via reachability of lower-layer subgoals.

In this paper we assume that subgoals are independent of each other. This means that reachability of any subgoal does not affect reachability of another one. Moreover, while a certain subgoal is reached, it remains reached, i.e., the system always progresses towards achieving its goals. Formally, it can be expressed as a stability property with respect to some state predicate P:

$$Stable(P) \;\Leftrightarrow\; \textit{“once } P \textit{ becomes true it remains true”}.$$

In Event-B, stability properties can be easily expressed by introducing auxiliary variables for storing the previous value of the state and then formulating stability properties as the invariant properties of the form:

$$P(prev\_state) = TRUE \;\Rightarrow\; P(state) = TRUE.$$

To express a goal decomposition in terms of Event-B, let us define a corresponding refinement pattern. We present it by the machine M_GD. The new pattern allows us to introduce a number of subgoals into our system model and express their reachability. Moreover, the refinement relation between patterns allows us to express reachability of the main goal via reachability of its subgoals.

Let us assume for simplicity, that system goal *Goal* is achieved by reaching three subgoals. The subgoals are defined as corresponding variables of the M_GD machine: $Subgoal_1$, $Subgoal_2$, and $Subgoal_3$. The goal independence assumption allows us to partition high-level goal state space $GSTATE$ into three non-empty subsets: $SG\_STATE1, SG\_STATE2, SG\_STATE3$. We define the subgoals as follows:

$$Subgoal_i \neq \varnothing \;\text{ and }\; Subgoal_i \subset SG\_STATEi,\; i \in 1..3.$$

To establish a relationship between the new state spaces $SG\_STATEi,\; i \in 1..3$, of the M_GD machine and the abstract state space of M_AGM machine we define the following function:

$$State\_map \in SG\_STATE1 \times SG\_STATE2 \times SG\_STATE3 \rightarrowtail GSTATE,$$

where $\rightarrowtail$ designates a bijection function. Essentially it partitions the original goal state space into three independent parts.

To postulate that the main goal is reached if and only if all three subgoals are reached, we add an axiom into the context of the M_GD machine:

$$\forall sg1, sg2, sg3 \cdot\ sg1 \in Subgoal_1 \wedge sg2 \in Subgoal_2 \wedge sg3 \in Subgoal_3 \\ \Leftrightarrow\ State\_map(sg1 \mapsto sg2 \mapsto sg3) \in Goal.$$

Refinement performed according to the *Goal Decomposition Pattern* is an example of the Event-B data refinement. We replace the abstract variable $gstate$ with the new variables $gstate_i \in SG\_STATEi$, $i \in 1..3$. The new variables model the state of the corresponding subgoals. The following gluing invariant allows us to prove data refinement:

$$gstate = State\_map(gstate1 \mapsto gstate2 \mapsto gstate3).$$

Essentially the M_GD machine decomposes the Reaching_Goal event of the M_AGM machine into three similar events $\text{Reaching\_SubGoal}_\text{i}, i \in 1..3$:

Machine M_GD
Reaching_SubGoal$_\text{i}$ $\widehat{=}$ refines Reaching_Goal
status *anticipated*
when
$gstate_i \in SG\_STATEi \setminus Subgoal_i$
then
$gstate_i :\in SG\_STATEi$
end
...

Let us observe that we can easily verify that the following stability property holds for the pattern M_GD:

$$Stable(gstate_1 \in Subgoal_1) \wedge Stable(gstate_2 \in Subgoal_2) \wedge Stable(gstate_3 \in Subgoal_3).$$

The proposed *Goal Decomposition Pattern* can be repeatedly used to refine subgoals into the subgoals of finer granularity until the desired level of details is reached.

3.4 Agent Modelling Pattern

Our elaborated *Abstract Goal Modelling* and *Goal Decomposition* patterns allow us to specify the system goal(s) at different levels of abstraction. In multi-agent systems, (sub)goals are usually achieved by system agents. Agents are independent entities that are capable of performing certain tasks. In general, the system might have several types of agents that are distinguished by the type of tasks that they are capable of performing. Our next refinement pattern – *Agent Modelling Pattern* – allows us to model agents and associate them with goals.

We introduce the set *AGENTS* that abstractly defines the set of system agents. In this refinement pattern we also introduce a concept of agent *eligibility*. An agent is *eligible* if it is capable of achieving a certain task (subgoal). We define the non-empty sets *EL_AG1*, *EL_AG2*, and *EL_AG3* of the agents eligible to achieve each particular subgoal.

Agent might fail while trying to achieve a certain subgoal. Then it is removed from the dynamic set of the eligible agents represented by the variable $elig_i$: $elig_i \subseteq EL\_AGi,\ i \in 1..3$.

A goal is achieved if there is at least one eligible agent associated with it. This is formulated as the corresponding invariant property in our pattern:

$$elig_1 \neq \varnothing \ \text{ and } \ elig_2 \neq \varnothing \ \text{ and } \ elig_3 \neq \varnothing.$$

The dynamic part of the *Agent Modelling Pattern* is defined in the machine M_AM. Since we assumed that the agents can fail, the goal assigned to the failed agent cannot be reached. To reflect this assumption in our model, we refine the abstract event Reaching_SubGoal$_i$ by two events Successful_Reaching_SubGoal$_i$ and Failed_Reaching_SubGoal$_i$, $i \in 1..3$, which respectively model successful and unsuccessful reaching of the subgoal by some eligible agent:

Machine M_AM
Successful_Reaching_SubGoal$_i$ $\widehat{=}$ refines Reaching_SubGoal$_i$
status $convergent$
any ag
when
$gstate_i \in SG\_STATEi \setminus Subgoal_i \wedge ag \in elig_i$
then
$gstate_i :\in Subgoal_i$
end
Failed_Reaching_SubGoal$_i$ $\widehat{=}$ refines Reaching_SubGoal$_i$
status $convergent$
any ag
when
$gstate_i \in SG\_STATEi \setminus Subgoal_i \wedge ag \in elig_i \wedge card(elig_i) > 1$
then
$gstate_i :\in SG\_STATEi \setminus Subgoal_i \parallel elig_i := elig_i \setminus \{ag\}$
end

In the guard of the event Failed_Reaching_SubGoal$_i$ we restrict possible agent failures by postulating that at least one agent associated with the subgoal remains operational: $card(elig_i) > 1$, $i \in 1..3$. This assumption allows us to change the event status from anticipated to convergent. In other words, we are now able to prove that, for each subgoal, the process of reaching it eventually terminates. To prove the convergence we define the following variant expression:

$$card(elig_1) \; + \; card(elig_2) \; + \; card(elig_3).$$

When an agent fails, it is removed from a corresponding set of eligible agents $elig_i$. This in turn decreases the value of $card(elig_i)$ and consequently the whole variant expression. On the other hand, when an agent succeeds in reaching the goal, all the events become disabled, thus ensuring system termination as well.

In practice, the constraint to have at least one operational agent associated with our model can be validated by probabilistic modelling of goal reachability, which is planned as a future work. Let us also note that for multi-robotic systems with many homogeneous agents this constraint is usually satisfied.

3.5 Agent Refinement Pattern

Above we have defined the notion of agent eligibility quite abstractly. We establish the relationship between subgoals (tasks) and agents that are capable of achieving them. Our last refinement pattern, *Agent Refinement Pattern*, aims at unfolding the notion of agent eligibility. Here we define the agent eligibility by introducing agent attributes – *agent types* and *agent statuses*. An eligible agent will be an operational agent that belongs to a particular agent type.

We define an enumerated set of agent types $AG\_TYPE = \{TYPE1, TYPE2, TYPE3\}$ and establish the correspondence between abstract sets of eligible

agents and the corresponding agent types by the following axioms:

$$\forall ag \cdot ag \in EL\_AGi \Leftrightarrow atype(ag) = TYPEi,\ i \in 1..3.$$

An agent is eligible to perform a certain subgoal if it has the type associated with this subgoal.

An agent might be operational or failed. To model the notion of agent status we define an enumerated set $AG\_STATUS = \{OK, KO\}$, where constants OK and KO designate operational and failed agents correspondingly.

Below we present an excerpt from the dynamic part of the *Agent Refinement Pattern* – the machine M_AR. We add a new variable *astatus* to store the dynamic status of each agent:

$$astatus \in AGENTS \rightarrow AG\_STATUS.$$

Moreover, we data refine the variables $elig_i$. The following gluing invariants relate them with the concrete sets:

$$elig_i = \{a \mid a \in AGENTS \wedge atype(a) = TYPEi \wedge astatus(a) = OK\},\ i \in 1..3.$$

In our case, the dynamic set of agents eligible to perform a certain subgoal becomes a set of active agents of the particular type. The event Failed_Reaching_SubGoal$_i$ is now refined to take into account the concrete definition of agent eligibility. The event also updates the status of the failed agent.

Machine M_AR
Successful_Reaching_SubGoal$_i$ $\widehat{=}$ refines Successful_Reaching_SubGoal$_i$
  any ag
  when
    $gstate_i \in SG\_STATEi \setminus Subgoal_i \wedge astatus(ag) = OK \wedge atype(ag) = TYPE_i$
  then
    $gstate_i :\in Subgoal_i$
  end
Failed_Reaching_SubGoal$_i$ $\widehat{=}$ refines Failed_Reaching_SubGoal$_i$
  any ag
  when
    $gstate_i \in SG\_STATEi \setminus Subgoal_i \wedge astatus(ag) = OK \wedge atype(ag) = TYPE_i \wedge$
    $card(\{a | a \in AGENTS \wedge atype(a) = TYPE_i \wedge astatus(a) = OK\}) > 1$
  then
    $gstate_i :\in SG\_STATEi \setminus Subgoal_i \parallel astatus(ag) := KO$
  end

Further refinement patterns can be defined to model various fault tolerance mechanism. However, in this paper instead of building further the collection of patterns, we will demonstrate how to instantiate and use the described patterns in a concrete development.

4 Case Study: a Multi-Robotic System

4.1 A Case Study Description

As a case study we consider a multi-robotic system. The goal of the system is to coordinate identical robots to get a certain area cleaned. The area is divided into several zones, which can be further divided into a number of sectors. Each zone has a base station – a static computing and communicating device – that coordinates the cleaning of the zone. In its turn, each base station supervises a number of robots by assigning cleaning tasks to them.

A robot is an autonomous electro-mechanical device – a special kind of a rover that can move and clean. The base station may assign a robot a sector

– a certain area in the zone – to clean. As soon as the robot receives a new cleaning task, it autonomously travels to this area and starts to clean it. After successfully completing its mission, it returns back to the base station to receive a new order. The base station keeps track of the cleaned sectors. A robot may fail to clean the assigned sector. In that case, the base station assigns another robot to perform this task. To ensure that the whole area is eventually cleaned, each base station in its turn should ensure that its zone is eventually cleaned.

The system should function autonomously, i.e., without human intervention. Such kind of systems are often deployed in hazardous areas (nuclear power plants, disaster areas, mine fields, etc.). Hence guaranteeing system resilience is an important requirement. Therefore, we should formally demonstrate that the system goal is achievable despite possible robot failures.

Next, we will show how to develop a multi-robotic system by refinement in Event-B and demonstrate how to rely on the patterns proposed in Section 3 to formally specify the system behaviour to ensure reachability of the overall system goal.

4.2 Pattern-Driven Refinement of a Multi-Robotic System

In this section we will describe our formal development of a multi-robotic system in Event-B. The development is concluded via instantiation of the proposed patterns, with the goal decomposition pattern being applied twice in a row.

Abstract model. The initial model defined by the machine MRS_Abs specifies the behaviour of a multi-robotic system according to the *Abstract Goal Modelling Pattern*. We apply this pattern by instantiating abstract variables with the concrete values and specifying events that model system behaviour.

The state space of the initial model is defined by the type $BOOL$. The value TRUE corresponds to the situation when the desired goal is achieved (i.e., the whole territory is cleaned), while FALSE represents the opposite situation.

Similarly to the pattern machine M_AGM, the machine MRS_Abs contains an event, CleaningTerritory, that models system behaviour. It abstractly represents the process of cleaning the territory, where a variable $completed \in BOOL$ models the current state of the system goal. This event is constructed according to the pattern event Reaching_Goal by taking all the instantiations into account, as shown below:

Machine AbsMRS
Variables $completed$
Invariants $completed \in BOOL$
Events
...
CleaningTerritory $\widehat{=}$
status $anticipated$
when
$completed = FALSE$
then
$completed :\in BOOL$
end

The system continues its execution until the whole territory is cleaned, i.e., as long as *completed* stays FALSE. At this level of abstraction, the event CleaningTerritory has the *anticipated* status. In other words, similarly to the

abstract pattern, we delay the proof that the event eventually converges to subsequent refinements. It is easy to see that the machine AbsMRS is an instantiation of the pattern machine M_AGM, where the abstract type *GSTATE* its replaced with *BOOL*, the constant *Goal* is instantiated with a singleton set {TRUE}, and the variable *gstate* is renamed into *completed*.

First refinement. Our initial model specifies system behaviour in a highly abstract way. It models the process of cleaning the whole territory. The goal of the first refinement is to model the cleaning of the territory zones. Refinement is performed according to the *Goal Decomposition Pattern*.

In the first refinement step resulting in the machine MRS_Ref1, we augment our model with representation of subgoals. The whole territory is divided into n zones, $n \in \mathbb{N}$ and $n \geq 1$. We associate the notion of a *subgoal* with the process of *cleaning a particular zone*. Thus a subgoal is achieved when the corresponding zone is cleaned. A new variable *zone_completed* represents the current subgoal status for every zone. The value TRUE corresponds to the situation when the certain zone is cleaned:

$$zone\_completed \in 1..n \to BOOL.$$

The refined model MRS_Ref1 is built as an instantiation of the *Goal Decomposition Pattern* machine M_GD, where the subgoal states are defined as elements of the variable *zone_completed*, i.e.,

$$gstate_i \ = zone\_completed(i), \text{ for } i \in 1..n.$$

This observation suggests the following gluing invariant between the initial and the refined models:

$$completed = TRUE \Leftrightarrow zone\_completed[1..n] = \{TRUE\}.$$

The invariant can be understood as follows: the territory is considered to be cleaned if and only if its every zone is cleaned.

The pattern events $\text{Reaching\_SubGoal}_i$ correspond to a single event CleaningZone:

```
Machine MRS_Ref1
CleaningZone ≙ refines CleaningTerritory
  status anticipated
  any zone, zone_result
  when
     zone ∈ 1..n ∧ zone_completed(zone) = FALSE∧zone_result ∈ BOOL
  then
     zone_completed(zone) := zone_result
  end
```

Second refinement. In our development of a multi-robotic system we should apply the goal decomposition pattern twice, until we reach the level of "primitive" goals, i.e., the goals for which we define the classes of agents eligible for execution of these goals.

Every zone in our system is divided into k sectors, $k \in \mathbb{N}$ and $k \geq 1$. A robot is responsible for cleaning a certain sector. We associate the notion of a *subsubgoal* (or simply *task*) with the process of *cleaning a particular sector*. The task is completed when the sector is cleaned. A new variable *sector_completed* represents the current task status for every sector:

$$sector\_completed \in 1..n \to (1..k \to BOOL).$$

The refined model is again built as an instantiation of the *Goal Decomposition Pattern*, where the subsubgoal states are defined as the elements of the variable *sector_completed*, i.e.,

$$gstate_{ij} = sector\_completed(i)(j), \text{ for } i \in 1..n,\ j \in 1..k.$$

A gluing invariant expresses the relationship between subgoals and tasks:

$$\forall zone \cdot zone \in 1\,..\,n \Rightarrow (zone\_completed(zone) = TRUE \Leftrightarrow sector\_completed(zone)[1\,..\,k] = \{TRUE\}).$$

The invariant postulates that any zone is cleaned if and only if its every sector is cleaned. The abstract event CleaningZone is refined by the event CleaningSector. The subsubgoal will be achieved if this section is eventually cleaned:

Machine MRS_Ref2
CleaningSector $\widehat{=}$ refines CleaningZone
status $anticipated$
any $zone, sector, sector\_result$
when
$zone \in 1..n \ \wedge sector \in 1\,..\,k \ \wedge sector\_completed(zone)(sector) = FALSE \ \wedge$
$sector\_result \in BOOL$
then
$sector\_completed(zone) := sector\_completed(zone) \lhd\!\!+ \{sector \mapsto sector\_result\}$
end

Now we have reached the desire level of granularity of our subgoals. In the next refinement step (the machine MRS_Ref3) we are going to augment our model with an abstract representation of agents.

Third refinement. The next refined model of our development is constructed according to the refinement *Agent Modelling Pattern*. As a result, we introduce the abstract set $AGENTS$, and its subset $ELIG$ containing the eligible agents for executing the tasks. A new variable *elig* represents the dynamic set of (currently available) eligible agents. Following the proposed pattern, we should also guarantee that there will be at least one eligible agent for cleaning the sector. This property is formulated as an additional invariant: $elig \neq \varnothing$.

Moreover, according to the pattern, we need abstractly introduce agent failures. This is achieved by refining the abstract event CleaningSector by two events SuccessfulCleaningSector and FailedCleaningSector, which respectively model successful and unsuccessful execution of the task by some eligible agent:

Machine MRS_Ref3
SuccessfulCleaningSector $\widehat{=}$ refines CleaningSector
status $convergent$
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1\,..\,k \ \wedge$
$sector\_completed(zone)(sector) = FALSE \ \wedge ag \in elig$
then
$sector\_completed(zone) := sector\_completed(zone) \lhd\!\!+ \{sector \mapsto TRUE\}$
end
FailedCleaningSector $\widehat{=}$ refines CleaningSector
status $convergent$
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1\,..\,k \ \wedge sector\_completed(zone)(sector) = FALSE \ \wedge$
$ag \in elig \ \wedge card(elig) > 1$
then
$sector\_completed(zone) := sector\_completed(zone) \lhd\!\!+ \{sector \mapsto FALSE\}$
$elig := elig \setminus \{ag\}$
end

Following the proposed pattern, we add in the event FailedCleaningSector the guard $card(elig) > 1$ to restrict possible agent failure in task performance. Let us also note that for multi-robotic systems with many homogeneous agents this constraint is not unreasonable. This assumption allows us to prove the convergence of the goal-reaching events, i.e., to prove that the process of cleaning the territory eventually terminates.

Fourth refinement. Finally, the *Agent Refinement Pattern* for introducing agent types and their statuses is applied to produce the last refined model of our multi-robotic system. In this refinement step we explicitly define the agent types – robots and base stations. We partition our abstract set $AGENTS$ by disjointed non-empty subsets RB and BS, that represent robots and base stations respectively. In this case study robots perform the cleaning task. Hence our abstract set of eligible agents is completely represented by robots: $ELIG = RB$. Robots might be active or failed. We introduce the enumerated set $STATUS$, which in our case has two elements $\{active, failed\}$.

At previous refinement step we have modelled agent faults while performing their tasks in a very abstract way. Now we will specify them more concretely. We assume that only robots may fail in our multi-robotic system. Their dynamic status is stored in the variable rb_status:

$$rb\_status \in RB \to STATUS.$$

The abstract variable $elig$ is now data refined by the concrete set:

$$elig \;=\; \{a | a \in AGENTS \wedge atype(a) = RB \wedge rb\_status(a) = active\}.$$

The concrete events are also built according to the proposed pattern. For instance, the event FailedCleaningSector can now be specified as follows:

Machine MRS_Ref4
FailedCleaningSector $\widehat{=}$ refines FailedCleaningSector
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1..k \;\wedge sector\_completed(zone)(sector) = FALSE \wedge$
$ag \in RB \;\wedge card(\{a | a \in RB \wedge rb\_status(a) = active\}) > 1$
$rb\_status(ag) = active$
then
$sector\_completed(zone) := sector\_completed(zone) \Leftarrow \{sector \mapsto FALSE\}$
$rb\_status(ag) := failed$
end

An overview of the development of an autonomous multi-robotic system according to the proposed specification and refinement patterns is shown in the Fig. 1.

5 Conclusions

5.1 Discussion

In this paper we have proposed a formal goal-oriented approach to development of resilient MAS. We have demonstrated how to rigorously define goals in Event-B and ensure goal reachability by refinement. We have defined a set of modelling and refinement patterns that describe generic solutions common to formal modelling of MAS. Rigorous modelling of the impact of agent failures on goal achieving allowed us to propose a dynamic goal reallocation mechanism that

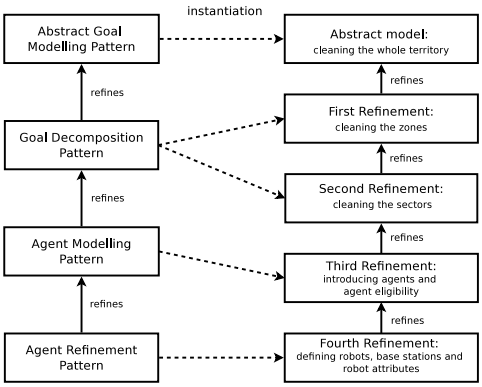


Fig. 1. Overview of the development

guarantees system resilience in presence of agent failures. We have illustrated our approach by a case study – development of an autonomic multi-robotic system.

While modelling the behaviour of a multi-robotic system, we have shown that refinement process allows us also to discover restrictions that we have to impose on system behaviour to guarantee its resilience. In our case, the goal was achievable only if at least one robot remains healthy. Feasibility of such a restriction can be checked probabilistically based on the failure rates of robots. In our future work we are planning to integrate stochastic reasoning in our formal development. Moreover, it would be also interesting to experiment with different schemes for goal decomposition and dynamic goal reallocation.

5.2 Related Work

Our approach is different from numerous process-algebraic approaches used for modeling MAS. Firstly, we relied on proof-based verification that does not impose restrictions on the size of the model, number of agents, etc. Secondly, we adopted a system's approach, i.e., we modeled the entire system and extracted the specifications of its individual components by decomposition. Such an approach allows us to ensure resilience by enabling goal reallocation at different architectural levels. Furthermore, by incrementally increasing complexity of our models, we have successfully managed to cope both with complexity of requirements and verification.

Formal modelling of MAS has been undertaken by [13, 12, 14]. The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. Our modelling pursued a different goal – we aimed at formally guaranteeing that the specified agent behaviour achieves the defined goals. Formal modelling of fault tolerant MAS in Event-B has been undertaken by Ball and Butler [3]. They have proposed a number of informally described patterns that allow the designers to add well-known fault tolerance mechanisms to the specifications. In our approach, we implemented

goal reallocation to guarantee goal reachability that can be also considered as a goal-specific fault tolerance.

The foundational work on goal-oriented development has been done by van Lamsweerde [5]. The original motivation behind the goal-oriented development was to structure the requirements and derive properties in the form of temporal logic formulas that the system design should satisfy. Over the last decade the goal-oriented approach has received several extensions that allow the designers to link git with formal modelling [6, 8, 10]. These works aimed at expressing temporal logic properties in Event-B. In our work, we have relied on goals to facilitate structuring of system behaviour but derived system specification that satisfies the desired properties by refinement.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
3. Ball, E., Butler, M.: Event-b patterns for specifying fault-tolerance in multi-agent interaction. In: Methods, Models and Tools for Fault Tolerance, pp. 104–129 (2009)
4. EU-project DEPLOY: http://www.deploy-project.eu/
5. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Requirements Engineering. pp. 249–263 (2001)
6. Landtsheer, R.D., Letier, E., van Lamsweerde, A.: Deriving tabular event-based specifications from goal-oriented requirements models. In: Requirements Engineering. p. 200 (2003)
7. Laprie, J.: From dependability to resilience. In: 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks. pp. G8–G9 (2008)
8. Matoussi, A., Gervais, F., Laleau, R.: A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification. In: 16th International Conference on Engineering of Complex Computer Systems. pp. 139–148. IEEE (2011)
9. OMG Mobile Agents Facility (MASIF): http://www.omg.org
10. Ponsard, C., Dallons, G., Philippe, M.: From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In: ERCIM News (75). pp. 22–23 (2008)
11. Rodin. Event-B Platform: http://www.event-b.org/
12. Roman, G.C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In: In Proc. of FASE04. LNCS, vol. 2984, pp. 12–36. Springer (2004)
13. Roman, G.C., Julien, C., Payton, J.: Modeling Adaptive Behaviors in Context UNITY. In: Theoretical Compuuture Science. vol. 376, pp. 185–204 (2007)
14. Roman, G.C., P.McCann, Plun, J.: Mobile UNITY: Reasoning and Specification in Mobile Computing. In: ACM Transactions of Software Engineering and Methodology. pp. 250–282 (1997)
15. Vain, J., Tammet, T., Kuusik, A., Juurik, S.: Towards scalable proofs of robot swarm dependability. In: BEC 2008. pp. 199 – 202 (2008)

Paper VI

A Case Study in Formal Development of a Fault Tolerant Multi-Robotic System

Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis

Originally published in: Paris Avgeriou (Ed.), *Proceedings of 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, LNCS 7527, 16–31, Springer-Verlag Berlin Heidelberg, 2012.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-33176-3_2

A Case Study in Formal Development of a Fault Tolerant Multi-Robotic System

Inna Pereverzeva[1,2], Elena Troubitsyna[2], and Linas Laibinis[2]

[1] Turku Centre for Computer Science
[2] Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku, Finland
{inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi

Abstract. Multi-robotic systems are typical examples of complex multi-agent systems. The robots – autonomic agents – cooperate with each other in order to achieve the system goals. While designing multi-robotic systems, we should ensure that these goals remain achievable despite robot failures, i.e., guarantee system fault tolerance. However, designing the fault tolerance mechanisms for multi-agent systems is a notoriously difficult task. In this paper we describe a case study in formal development of a complex fault tolerant multi-robotic system. The system design relies on cooperative error recovery and dynamic reconfiguration. We demonstrate how to specify and verify essential properties of a fault tolerant multi-robotic system in Event-B and derive a detailed formal system specification by refinement. The main objective of the presented case study is to investigate suitability of a refinement approach for specifying a complex multi-agent system with co-operative error recovery.

Keywords: Event-B, formal modelling, refinement, fault tolerance, multi-robotic system

1 Introduction

Over the last decade, the field of autonomous multi-robotic systems has grown dramatically. There are several research directions that are continuously receiving significant attention: autonomous navigation and control, self-organising behaviour, architectures for multi-robot co-operation, to name a few. The robot co-operation is studied from a variety of perspectives: delegation of authority and control, heterogeneous versus homogeneous architectures, communication structure etc. In this paper we focus on studying the fault tolerance aspects of multi-robotic co-operation. Namely, we show by example how to formally derive a specification of a multi-robotic system that relies on dynamic reconfiguration and co-operative error recovery to achieve fault tolerance.

Our paper presents a case study in formal development of a cleaning multi-robotic system. That kind of systems are typically employed in hazardous areas. The system has a heterogenous architecture consisting of several stationary devices, base stations, that coordinate the work of respective groups of robots. A

base station assigns a robot to clean a certain segment. Since both base stations and robots can fail, the main objective of our formal development is to formally specify co-operative error recovery and verify that the proposed design ensures goal reachability, i.e., guarantees that the whole territory will be eventually cleaned. The proposed development approach ensures goal reachability "by construction". It is based on refinement in Event-B [2] – a formal top-down approach to correct-by-construction system development. The main development technique – refinement – allows us to ensure that a resulting specification preserves the globally observable behaviour and properties of the specifications it refines. The Rodin platform [8] automates modelling and verification in Event-B.

In this paper we demonstrate how to formally define a system goal and, in a stepwise manner, *derive* a detailed specification of the system architecture. While refining the system specification, we gradually introduce a representation of the main elements of the architecture – base stations and robots – as well as failures and the fault tolerance mechanisms. Moreover, we identify the main properties of a fault tolerant multi-robotic system and demonstrate how to formally specify and verify them as a part of the refinement process. In particular, we show how to derive a mechanism for cooperative error recovery in a systematic way.

Traditionally, the behaviour of multi-robotic systems is verified by simulation and model checking. These approaches allow the designers to investigate only a limited number of scenarios and require a significant reduction of the state space. In our paper, we discuss advantages and limitations of a refinement approach to achieve full-scale verification of a multi-robotic system.

The paper is structured as follows. In Section 2 we briefly overview the Event-B formalism. Section 3 describes the requirements for our case study – a multi-robotic cleaning system – and outlines the development strategy. Section 4 presents a formal development of the cleaning system and demonstrates how to express and verify its properties in the refinement process. Finally, in Section 5 we conclude by assessing our contributions and reviewing the related work.

2 Modelling and Refinement in Event-B

The Event-B formalism – a variation of the B Method [1] – is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the model state represented as a collection of variables and defines operations on the state, i.e., it describes the *behaviour* of the modelled system. Usually, a machine has an accompanying component, called *context*, which may include user-defined carrier sets, constants and their properties given as a list of model axioms. In Event-B, the model variables are strongly typed by the constraining predicates. These predicates and the other important properties that must be preserved by the model constitute model *invariants*.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of the event is defined by the next-state relation R_e. In Event-B, R_e is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. The guard defines the conditions under which the assignment can be performed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If an event does not have local variables, it can be described simply as:

$$e \mathrel{\widehat{=}} \textbf{when } G_e \textbf{ then } R_e \textbf{ end}.$$

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract system specification that non-deterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce non-determinism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we should define so called *gluing invariant* as a part of the invariant of the refined machine. The gluing invariant defines the relationship between the abstract and concrete variables.

Often a refinement step introduces new events and variables into the abstract specification. The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit *skip*. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events *converge*. To prove it, we need to define a *variant* – an expression over a finite subset of natural numbers – and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of abstraction. Then the event obtains the status *anticipated*. This obliges the designer to prove, at some later refinement step, that the event indeed converges.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is formally demonstrated by discharging the relevant proof obligations generated by the Rodin platform [8]. Rodin also provides an automated tool support for proving.

3 Multi-Robotic Systems

Our paper focuses on formal modelling and development of multi-robotic systems that should function autonomously , i.e., without human intervention. Such kind of systems are often deployed in hazardous areas, e.g., nuclear power plants, disaster areas, minefields, etc.

Typically, the main task or *goal* that a multi-robotic system should accomplish is split between the deployed robots. The robot activities are coordinated by a number of stationary units – base stations. Since both robots and base stations may fail, to ensure success of the overall goal we should incorporate the fault tolerance mechanisms into the system design. These mechanisms rely on co-operative error recovery that allows the system dynamically reallocate functions from the failed agents to the healthy ones.

Designing co-operative error recovery for multi-agent systems is a notoriously complex task. The complexity is caused by several factors: asynchronous communication, a highly decentralised system architecture and the lack of the "universally known" global system state. Yet, the designers should guarantee that the system goals are achievable despite failures. A variety of failure modes and scenarios makes verification of goal reachability of the co-operative error recovery difficult and time-consuming. Therefore, there is a clear need for rigorous approaches that support scalable design and verification in a systematic manner.

Next we present the requirements of our case study – a multi-robotic system for cleaning a territory. Then we will demonstrate how we can formally develop the system in Event-B and prove its essential properties.

3.1 A Case Study: Cleaning a Territory

The goal of the system is to get a certain territory cleaned by robots. The whole territory is divided into several *zones*, which in turn are further divided into a number of *sectors*. Each zone has a *base station* that coordinates the cleaning activities within the zone. In general, one base station might coordinate several zones. In its turn, each base station supervises a number of robots attached to it by assigning cleaning tasks to them.

A robot is an autonomous electro-mechanical device that can move and clean. A base station may assign a robot a specific sector to clean. Upon receiving the assignment, the robot autonomously moves to this sector and performs cleaning. After successfully completing its mission, the robot returns back to the base station to receive a new assignment. The base station keeps track of the cleaned and non-cleaned sectors. Moreover, the base stations periodically exchange the information about their cleaned sectors.

While performing the given task, a robot might fail which subsequently leads to a failure to clean the assigned sector. We assume that a base station is able to detect all the failed robots attached to it. In case of a robot failure, the base station may assign another active robot to perform the failed task.

A base station might fail as well. We assume that a failure of a base station can be detected by the others base stations. In that case, the healthy base stations redistribute control over the robots coordinated by the failed base station.

Let us now to formulate the main requirements and properties associated with the multi-robotic system informally described above.

(PR1) *The main system goal: the whole territory has to be cleaned.*
(PR2) *To clean the territory, every its zone has to be cleaned.*
(PR3) *To clean a zone, every its sector has to be cleaned.*
(PR4) *Every cleaned sector or zone remains cleaned during functioning of the system.*
(PR5) *No two robots should clean the same sector.* In other words, a robot gets only non-assigned and non-cleaned sectors to clean.
(PR6) *The information about the cleaned sectors stored in any base station has to be consistent with the current state of the territory.* More specifically, if a base station sees a particular sector in some zone as cleaned, then this sector

is marked as cleaned in the memory of the base station responsible for it. Also, if a sector is marked as non-cleaned in the memory of the base station responsible for it, then any base station sees it as non-cleaned.

(PR7) *Base station cooperation: if a base station has been detected as failed then some base station will take the responsibility for all the zones and robots of the failed base station.*

(PR8) *Base station cooperation: if a base station has no more active robots, a group of robot is sent to this base station from another base station.*

(PR9) *Base station cooperation: if a base station has cleaned all its zones, its active robots may be reallocated under control of another base station.*

The last three requirements essentially describe the co-operative recovery mechanisms that we assume to be present in the described multi-robot system.

3.2 Formal Development Strategy

In the next section we will present a formal Event-B development of the described multi-system robotic system. We demonstrate how to specify and verify the given properties (PR1)–(PR9). Let us now give a short overview of this development and highlight formal techniques used to ensure the proposed properties.

We start with a very abstract model, essentially representing the system behaviour as a process iteratively trying to achieve the main goal (PR1). The next couple of data refinement steps decompose the main goal into a set of subgoals, i.e., reformulate it in the terms of zones and sectors. We will define and prove the relevant gluing invariants establishing a formal relationship between goals and the corresponding subgoals.

While the specification remains highly abstract, we postulate goal reachability property by defining *anticipate* status for the involved events. Once, as a result of the refinement process, the model becomes sufficiently detailed, we change the event status into *convergent* and prove their termination by supplying the appropriate variant expression.

Next we introduce different types of agents (i.e., base stations and robots). The base stations coordinate execution of the tasks required to achieve the corresponding subgoal, while the robots execute the tasks allocated on them. We formally define the relationships between different types of agents, as well as agents and respective subgoals. These relationships are specified and proved as invariant properties of the model.

The consequent refinement steps explicitly introduce agent failures, the information exchange as well as co-operation activities between the agents. The integrity between the local and the global information stored within base stations is again formulated and proved as model invariant properties.

We assume that communication between the base stations as well as the robots and the base stations is reliable. In other words, messages are always transmitted correctly without any loss or errors. The main focus of our development is on specifying and verifying the co-operative recovery mechanisms.

4 Development of a Multi-Robotic System in Event-B

4.1 Abstract Model

We start our development by abstractly modelling the described multi-robotic system. We aim to ensure the property (PR1). The main system goal is to clean the whole territory. The process of achieving this goal is modelled by the simple event Body presented below. A variable $goal \in STATE$ models the current state of the system goal. It obtains values from the enumerated set $STATE = \{incompl, compl\}$, where the value $compl$ corresponds to the situation when the goal is achieved, otherwise it is equal to $incompl$. The system continues its execution until the whole territory is not cleaned, i.e., while $goal$ stays $incompl$.

Body $\widehat{=}$
status *anticipated*
when
$goal \neq compl$
then
$goal :\in STATE$
end

The event Body has the status *anticipated*. This means that goal reachability is postulated rather than proved. However, at some refinement step it also obliges us to prove that the event or its refinements converge, i.e., to prove that the process of achieving goal eventually terminates.

4.2 First Refinement: Zone Cleaning

Our initial model represents the system behaviour at a high level of abstraction. The objective of our first refinement step is to elaborate on the process of cleaning the territory. Specifically, we assume that the whole territory is divided into n zones, where $n \in \mathbb{N}$ and $n \geq 1$, and aim at ensuring the property (PR2).

We augment our model with a representation of subgoals. We also associate the notion of a *subgoal* with the process of *cleaning a particular zone*. A subgoal is achieved only when the corresponding zone is cleaned. A new variable $zones$ represents the current subgoal status for every zone:

$$zones \in 1..n \rightarrow STATE.$$

In this refinement step we perform a data refinement: we replace the abstract variable $goal$ with a new variable $zones$. To establish the relationship between those variables, we formulate the following gluing invariant:

$$goal = compl \Leftrightarrow zones[1..n] = \{compl\}.$$

The invariant can be understood as follows: the territory is considered to be cleaned if and only if its every zone is cleaned. Hence, hereby we have formalised the property (PR2). The refined event Body is presented below:

Body $\widehat{=}$ refines Body
status *anticipated*
any z, res
when
$z \in 1..n \ \wedge \ zones(z) \neq compl \wedge res \in STATE$
then
$zones(z) := res$
end

Moreover, while a certain subgoal is reached, it stays such, i.e., the system always progresses towards achieving its goals. Thereby we ensure the property (PR4).

4.3 Second Refinement: Sector Cleaning

In the next refinement step we further decompose system subgoals into a set of subsubgoals. Specifically, we assume that each zone in our system is divided into k sectors, where $k \in \mathbb{N}$ and $k \geq 1$, and aim at formalising the property (PR3). We establish the relationship between the notion of a subsubgoal (or simply *a task*) and the process of *cleaning a particular sector*. A task is completed when the corresponding sector is cleaned. A new variable *territory* represents the current status of each sector:

$$territory \in 1 \mathinner{..} n \to (1 \mathinner{..} k \to STATE).$$

The refinement step is again an example of a data refinement. Indeed, we replace the abstract variable *zones* with a new variable *territory*. The following gluing invariant expresses the relationship between subgoals and subsubgoals (tasks) and correspondingly ensures the property (PR3):

$$\forall j \cdot j \in 1 \mathinner{..} n \Rightarrow (zones(j) = compl \Leftrightarrow territory(j)[1 \mathinner{..} k] = \{compl\}).$$

The invariant postulates that a zone is cleaned if and only if each of its sectors is cleaned.

The abstract event Body is further refined. It is now models cleaning of a previously non-cleaned sector s in a zone z. The task is achieved when this sector is eventually cleaned, i.e., *result* becomes *compl*.

Body $\widehat{=}$ refines Body
status *anticipated*
any $z, s, result$
when
$zone \in 1..n \ \wedge s \in 1 \mathinner{..} k \ \wedge territory(z)(s) \neq compl \ \wedge result \in STATE$
then
$territory(z) := territory(z) \Leftarrow \{s \mapsto result\}$
end

Let us observe that the event Body also preserve the property (PR4).

At this refinement step we have achieved a sufficient level of detail to introduce an explicit representation of the agents – base stations and robots. This constitutes the main objective of our next refinement step.

4.4 Third Refinement: Introducing Agents

We start by defining, in the model context, the abstract finite set $AGENTS$ and its disjointed non-empty subsets RB and BS that represent the robots and the base stations respectively. To define a relationship between a zone and its supervising base station, we introduce the variable *responsible*, which is defined as the following total function:

$$responsible \in 1 \mathinner{..} n \to BS.$$

Each robot is supervised by a certain base station. During system execution robots might become inactive (failed). We model the relationship between robots and their supervised station by a variable *attached*, defined as partial function:

$$attached \in RB \nrightarrow BS.$$

The new function variables $asgn\_z$ and $asgn\_s$ model respectively the zone and the sector assigned to a robot to clean. When a robot is idle, i.e., it does not have a task assigned to it, the corresponding function value is 0:

$$asgn\_z \in RB \nrightarrow 0 \mathinner{..} n,\ asgn\_s \in RB \nrightarrow 0 \mathinner{..} k.$$

We require that only the robots that have a supervisory base station might receive a cleaning task:

$$dom(attached) = dom(asgn\_z),\ dom(asgn\_z) = dom(asgn\_s).$$

Now we can formulate the property (PR5) – *no two robots can clean the certain sector at the same time* – as a model invariant:

$$\forall rb1, rb2 \cdot rb1 \in dom(attached) \wedge rb2 \in dom(attached) \wedge asgn\_z(rb1) = asgn\_z(rb2) \wedge$$
$$asgn\_s(rb1) \neq 0 \wedge asgn\_s(rb2) \neq 0 \wedge asgn\_s(rb1) = asgn\_s(rb2) \Rightarrow rb1 = rb2.$$

To coordinate the cleaning process, a base station stores the information about its own cleaned sectors and periodically updates information about the status of the other cleaned sectors. Therefore, we assume that each base station has a "map" – a knowledge about all sectors of the whole territory. To model this, we introduce a new variable, $local\_map$:

$$local\_map \in BS \rightarrow (1 \mathinner{..} n \nrightarrow (1 \mathinner{..} k \rightarrow STATE)).$$

The "maps" are defined only for the base stations that have any zone cleaning to coordinate, i.e., $bs \in ran(responsible)$:

$$\forall bs \cdot bs \in ran(responsible) \Rightarrow local\_map(bs) \in 1 \mathinner{..} n \rightarrow (1 \mathinner{..} k \rightarrow STATE),$$
$$\forall bs \cdot bs \in BS \wedge bs \notin ran(responsible) \Rightarrow local\_map(bs) = \varnothing.$$

The abstract variable *territory* represents the global knowledge on the whole territory. For any sector and zone, this global knowledge has to be consistent with the information stored by the base stations. Namely, if in the local knowledge of any base station bs a sector s is marked as cleaned, i.e., $local\_map(bs)(z)(s) = compl$, then it should be cleaned according to the global knowledge as well, i.e., $territory(z)(s) = compl$; and vice versa: if a sector s is marked as non-cleaned in the global knowledge, i.e., $territory(z)(s) = incompl$, then it remains non-cleaned according the local knowledge of any base station bs, i.e., $local\_map(bs)(z)(s) = incompl$. To establish those relationships, we formulate and prove the following invariants:

$$\forall bs, z, s \cdot bs \in ran(responsible) \wedge z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \Rightarrow$$
$$(local\_map(bs)(z)(s) = compl \Rightarrow territory(z)(s) = compl),$$

$$\forall bs, z, s \cdot bs \in ran(responsible) \wedge z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \Rightarrow$$
$$(territory(z)(s) = incompl \Rightarrow local\_map(bs)(z)(s) = incompl).$$

For each base station, the local information about its zones and sectors always coincides with the global knowledge about these zones and sectors:

$$\forall bs, z, s \cdot bs \in ran(responsible) \wedge z \in 1 \mathinner{..} n \wedge responsible(z) = bs \wedge s \in 1 \mathinner{..} k \Rightarrow$$
$$(territory(z)(s) = incompl \Leftrightarrow local\_map(bs)(z)(s) = incompl).$$

All together, these three invariants formalise the property (PR6).

A base station assigns a cleaning task to its attached robots. This behaviour is modelled by a new event NewTask. In the event guard, we check that the assigned sector s is not cleaned yet, i.e., $local\_map(bs)(z)(s) = incompl$, and no other robot is currently cleaning it. The last condition can be formally expressed as $s \notin ran((dom(asgn\_z \rhd \{z\})) \lhd asgn\_s)$, i.e., the sector s is not assigned to any robot that performs cleaning in the zone z:

NewTask $\widehat{=}$
any bs, rb, z, s
when
$bs \in BS \land rb \in dom(attached) \land attached(rb) = bs \ \land z \in 1 \,..\, n \ \land$
$responsible(z) = bs \ \land asgn\_z(rb) = 0 \ \land s \in 1 \,..\, k \ \land asgn\_s(rb) = 0 \ \land$
$local\_map(bs)(z)(s) = incompl \ \land s \notin ran((dom(asgn\_z \rhd \{z\})) \lhd asgn\_s)$
then
$asgn\_s(rb) := s$
$asgn\_z(rb) := z$
end

The robot failures have impact on execution of the cleaning process. The cleaning task cannot be performed if a robot assigned for this task has failed. To reflect this behaviour in our model, we refine the abstract event Body by two events TaskSuccess and TaskFailure, which respectively model successful and unsuccessful execution of the task. If the task has been successfully performed by the assigned robot rb, its supervising base station bs changes the status of the sector s to cleaned, i.e., we override the previous value of $local\_map(bs)(z)(s)$ by the value $compl$.

TaskSuccess $\widehat{=}$ refines Body
status $convergent$
any bs, rb, z, s
when
$bs \in BS \land rb \in dom(attached) \land attached(rb) = bs \land$
$z \in 1 \,..\, n \ \land responsible(z) = bs \ \land asgn\_z(rb) = z \land$
$s \in 1 \,..\, k \ \land asgn\_s(rb) = s \land local\_map(bs)(z)(s) = incompl$
then
$territory(z) := territory(z) \Leftarrow \{s \mapsto compl\}$
$local\_map(bs) := local\_map(bs) \Leftarrow \{z \mapsto local\_map(bs)(z) \Leftarrow \{s \mapsto compl\}\}$
$asgn\_s(rb) := 0$
$asgn\_z(rb) := 0$
$counter := counter - 1$
end

The dual event TaskFailure abstractly models the opposite situation caused by a robot failure. As a result, all the relationships concerning the failed robot rb are removed:

TaskFailure $\widehat{=}$ refines Body
status $convergent$
any bs, rb, z, s
when
$bs \in BS \land rb \in dom(attached) \land attached(rb) = bs \land$
$z \in 1 \,..\, n \ \land responsible(z) = bs \ \land asgn\_z(rb) = z \land$
$s \in 1 \,..\, k \ \land asgn\_s(rb) = s \land local\_map(bs)(z)(s) = incompl$
then
$territory(z) := territory(z) \Leftarrow \{s \mapsto incompl\}$
$asgn\_s := \{rb\} \lhd asgn\_s$
$asgn\_z := \{rb\} \lhd asgn\_z$
$attached := \{rb\} \lhd attached$
end

At this refinement step, we are ready to demonstrate that the events TaskSuccess and TaskFailure converge. To prove it, we define the following variant expression over system variables:

$$counter \ + \ card(dom(attached)),$$

where *counter* is an auxiliary variable that stores the number of all non-cleaned sectors of the whole territory. The initial value of *counter* is equal to $n * k$. When a robot fails to perform a task, it is removed from the corresponding set of the attached robots $dom(attached)$. This in turn decreases the value of $card(dom(attached))$ and consequently the whole variant expression. On the other hand, when a robot succeeds in cleaning a sector, the variable *counter* decreases and consequently the whole variant expression decreases as well. If there are no sectors to clean, the events become disabled and the system terminates.

A base station keeps track of the cleaned and non-cleaned sectors and repeatedly receives the information from the other base stations about their cleaned sectors. This knowledge is inaccurate for the period when the information is sent but not yet received. In this refinement step we abstractly model receiving the information by a base station. In the next refinement step, we are going to define this process of information broadcasting more precisely.

The new event UpdateMap models updating the local map of a base station bs. Here we have to ensure that the obtained information is always consistent with the global one. Specifically, the base station updates a sector s as cleaned only if it has this status according to the global knowledge, i.e., $territory(z)(s) = compl$.

UpdateMap $\widehat{=}$
any bs, z, s
when
$bs \in BS \wedge z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \wedge$
$responsible(z) \neq bs \wedge bs \in ran(responsible) \wedge$
$territory(z)(s) = compl$
then
$local\_map(bs) := local\_map(bs) \Leftarrow \{z \mapsto local\_map(bs)(z) \Leftarrow \{s \mapsto compl\}\}$
end

In this refinement step we also introduce an abstract representation of the base station co-operation defined by the property (PR7). Namely, we allow to reassign a group of robots from one base station to another. This behaviour is defined by the event ReassignRB. In the next refinement steps we will elaborate on this event and define the conditions under which this behaviour takes place.

Additionally, we model a possible redistribution between the base stations their pre-assigned responsibility for zones and robots. This behaviour is defined in the new event GetAdditionalResponsibility presented below. The event guard defines the conditions when such a change is allowed. A base station bs_j can take the responsibility for a set of new zones zss if it has the accurate knowledge about these zones, i.e., the information about their cleaned and non-cleaned sectors. Specifically, in the guard we check that the global status of each sector s from the zone z, i.e., $territory(z)(s)$, coincides with the local information that the base station bs_j has about this sector. In that case, we reassign responsibility for the zone(s) zss and the robots rbs to the base station bs_j:

GetAdditionalResponsibility $\widehat{=}$
any bs_i, bs_j, rbs, zs
when
$bs\_i \in BS \wedge bs\_j \in BS \wedge zs \subset 1 \mathinner{..} n \wedge$
$zs = dom(responsible \rhd \{bs\_i\}) \ \wedge rbs \subset dom(attached) \wedge$
$rbs = dom(attached \rhd \{bs\_i\}) \ \wedge bs\_i \neq bs\_j \ \wedge bs\_j \in ran(responsible) \wedge$
$(\forall z, s \cdot z \in zs \wedge s \in 1 \mathinner{..} k \Rightarrow territory(z)(s) = local\_map(bs\_j)(z)(s))$
then
$responsible := responsible \Leftarrow (zs \times \{bs\_j\})$
$attached := attached \Leftarrow (rbs \times \{bs\_j\})$
$asgn\_s := asgn\_s \Leftarrow (rbs \times \{0\})$
$asgn\_z := asgn\_z \Leftarrow (rbs \times \{0\})$
$local\_map(bs\_i) := \varnothing$
end

Modelling this behaviour allows us to formalise the property (PR9). Our next refinement step will elaborate on our chosen communication model that is needed to achieve such co-operative recovery.

4.5 Fourth Refinement: a Model of Broadcasting

In the fourth refinement step we aim at defining an abstract model of broadcasting. After receiving a notification from a robot about successful cleaning the assigned sector, a base station updates its local map and broadcasts the message about the cleaned sector to the other base stations. In its turn, upon receiving the message, each base station correspondingly updates its own local map. We assume that the communication between base stations is reliable: no message is lost and eventually every base station receives it. In further refinement steps, this model of the broadcasting can be further refined by a more concrete mechanism.

To model the described behaviour, we introduce a new relational variable, msg, that models the message broadcasting buffer:

$$msg \in BS \leftrightarrow (1 \mathinner{..} n \times 1 \mathinner{..} k).$$

If a message $(bs \mapsto (z \mapsto s))$ belongs to this buffer, this means that the sector s from the zone z has been cleaned, i.e., $territory(z)(s) = compl$. The first element of the message, bs, determines the base station the message is sent to. We formulate this property by the following system invariant:

$$\forall z, s \cdot z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \wedge (z \mapsto s) \in ran(msg) \Rightarrow territory(z)(s) = compl.$$

If there are no messages in the msg buffer for any particular base station then the local map of this base station is accurate, i.e., it coincides with the global knowledge about the teritory:

$$\forall bs, z, s \cdot z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \wedge bs \in ran(responsible) \wedge (bs \mapsto (z \mapsto s)) \notin msg \Rightarrow territory(z)(s) = local\_map(bs)(z)(s),$$

$$\forall bs \cdot bs \in ran(responsible) \wedge bs \notin dom(msg) \Rightarrow (\forall z, s \cdot z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \ \Rightarrow territory(z)(s) = local\_map(bs)(z)(s)).$$

After receiving a notification about successful cleaning of a sector, a base station marks this sector as cleaned in its local map and then broadcasts the message about it to other base stations. To model this, we refine the abstract event TaskSuccess. Specifically, in the event body we add a new assignment $msg := msg \cup (bss \times \{z \mapsto s\})$ to add a new message to the broadcasting buffer.

We also refine the abstract event UpdateMap. In particular, we replace the guard $territory(z)(s) = compl$ by the guard $(bs \mapsto (z \mapsto s)) \in msg$. This guard checks that there is a message for the base station bs about the cleaned sector s from the zone z. As a result of the event, the base station bs reads the message and marks the sector s in the zone z as cleaned in its local map.

UpdateMap $\widehat{=}$ refines UpdateMap
any bs, z, s
when
$bs \in BS \wedge z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \ \wedge responsible(z) \neq bs \wedge$
$bs \in ran(responsible) \ \wedge (bs \mapsto (z \mapsto s)) \in msg$
then
$local\_map(bs) := local\_map(bs) \lhd\!\!+ \{z \mapsto local\_map(bs)(z) \lhd\!\!+ \{s \mapsto compl\}\}$
$msg := msg \setminus \{bs \mapsto (z \mapsto s)\}$

4.6 Fifth Refinement: Introducing Robot Failures

Now we aim at modelling possible robot failures and elaborate on the abstract events concerning robot and zone reassigning. We start by partitioning the robots into active and failed ones. The current set of all active robots is defined by a new variable $active$ with the following invariant properties:

$$active \subseteq dom(attached),\ active \subseteq dom(asgn\_s),\ active \subseteq dom(asgn\_z).$$

Initially all robots are active, i.e., $active = RB$. A new event RobotFailure models possible robot failures that can happen at any time during system execution:

RobotFailure $\widehat{=}$
any rb
when
$rb \in active \wedge card(active) > 1$
then
$active := active \setminus \{rb\}$
end

We make an assumption that the last active robot can not fail and add the corresponding guard $card(active) > 1$ to the event RobotFailure to restrict possible robot failures. Let us note that for multi-robotic systems with many homogeneous robots this constraint is not unreasonable.

A base station monitors all its robots and detects the failed ones. The abstract event TaskFailure abstractly models such robot detection.

To formalise the property (PR8), we should model a situation when some base station bs_j does not have active robots anymore, i.e., $dom(attached \rhd \{bs\_j\}) \nsubseteq active$. In that case, some group of active robots rbs has to be sent to this base station bs_j from another base station bs_i. This behaviour is modelled by the event ReassignNewBStoRBs that refines the abstract event ReassignRB. As a result, all the robots from rbs become attached to the base station bs_j:

ReassignNewBStoRBs $\widehat{=}$ refines ReassignRB
any bs_i, bs_j, rbs
when
$bs\_i \in BS \wedge bs\_j \in BS \wedge rbs \subset active \wedge$
$ran(rbs \lhd attached) = \{bs\} \ \wedge bs\_i \in ran(responsible) \wedge$
$ran(rbs \lhd asgn\_s) = \{0\} \ \wedge rbs \neq \varnothing \ \wedge bs\_j \in ran(responsible) \wedge$
$bs\_i \neq bs\_j \ \wedge bs\_i \in ran(rbs \lhd\!\!\!- attached) \ \wedge dom(attached \rhd \{bs\_j\}) \nsubseteq active$
then
$attached := attached \lhd\!\!+ (rbs \times \{bs\_j\})$
end

This event can be further refined by a concrete procedure to choose a particular base station that will share its robots (e.g., based on load balancing).

Finally, to ensure the property (PR9) , let us consider the situation when all the sectors for which a base station is responsible are cleaned. In that case, all the active robots of the base station may be sent to some other base station that still has some unfinished cleaning to co-ordinate. This functionality is specified by the event SendRobotsToBS (a refinement of the event ReassignRB):

SendRobotsToBS $\widehat{=}$ refines ReassignRB
any bs_i, bs_j, rbs
when
$bs\_i \in operating \wedge bs\_j \in operating \wedge rbs \subset active \wedge$
$ran(rbs \lhd attached) = \{bs\_i\} \wedge bs\_i \in ran(responsible) \wedge$
$ran(rbs \lhd asgn\_s) = \{0\} \wedge rbs \neq \varnothing \wedge bs\_j \in ran(responsible) \wedge$
$bs\_i \neq bs\_j \wedge bs\_i \in ran(rbs \lhd attached) \wedge rbs = dom(attached \rhd \{bs\_i\}) \wedge$
$(\forall z \cdot z \in 1 \,..\, n \wedge responsible(z) = bs\_i \Rightarrow local\_map(bs\_i)(z)[1 \,..\, k] = \{compl\})$
then
$attached := attached \lhd\!\!+ (rbs \times \{bs\_j\})$
end

4.7 Sixth Refinement: Introducing Base Station Failures

In the final refinement step presented in the paper, we aim at specifying the base station failures. Each base station might be either operating or failed. We introduce a new variable *operating* to define the set of all operating base stations. The corresponding invariant properties are as follows.

$$operating \subseteq BS,$$
$$\forall bs \cdot bs \in BS \wedge local\_map(bs) = \varnothing \;\Rightarrow\; bs \notin operating.$$

Also, similarly to the event RobotFailure, we introduce a new event BaseStationFailure to model a possible base station failure.

In the fourth refinement step we assumed that a base station can take over the responsibility for the robots and zones of another base station. This behaviour was modelled by the event GetAdditionalResponsibility. Now we can refine this event by introducing an additional condition – only if a base station is detected as failed, another base station can take over its responsibility for the respective zones and robots:

GetAdditionalResponsibility $\widehat{=}$ refines GetAdditionalResponsibility
any bs_i, bs_j, za, rbs
when
$bs\_i \in BS \wedge bs\_j \in operating \wedge zs \subset 1 \,..\, n \wedge$
$zs = dom(responsible \rhd \{bs\_i\}) \wedge rbs \subset active \wedge rbs = dom(attached \rhd \{bs\_i\}) \wedge$
$bs\_i \neq bs\_j \wedge bs\_j \notin dom(msg) \wedge bs\_i \notin operating$
then
$responsible := responsible \lhd\!\!+ (zs \times \{bs\_j\})$
$attached := attached \lhd\!\!+ (rbs \times \{bs\_j\})$
$asgn\_s := asgn\_s \lhd\!\!+ (rbs \times \{0\})$
$asgn\_z := asgn\_z \lhd\!\!+ (rbs \times \{0\})$
$local\_map(bs\_i) := \varnothing$
end

As a result of the presented refinement chain, we arrived at a centralised model of the multi-robotic system. We can further refine the system to derive its distributed implementation, relying on the modularisation extension of Event-B to achieve this.

5 Discussion

Assessment of the development. The development of the presented multi-robotic system has been carried out with the support of the Rodin platform [8]. We have derived a complex system specification in six refinement steps. In general, the refinement approach has demonstrated a good scalability and allowed us to model intricate dependencies between the system components. We have been able to express and verify all the desired properties defined for our system. Therefore, we can make a general conclusion about suitability of the refinement technique for formal development and verification of the multi-robotic systems.

However, we have also identified a number of problems. Firstly, in spite of seeming simplicity, the relationships between the base stations, zones and sectors have been modelled using quite complex nested data structures (functions). The Rodin platform could not comfortably handle the proofs involving manipulations with the nested functions and required rather time-consuming interactive proving efforts. Secondly, the Rodin platform does not support the direct assignment to a function with nested arguments. For instance, instead of simply specifying $local\_map(bs)(z)(s) := compl$, we have to express it as the following intricate statement $local\_map(bs) \Leftarrow \{z \mapsto local\_map(bs)(z) \Leftarrow \{s \mapsto compl\}\}$, i.e., use the overriding relation twice. These two problems can be alleviated with a mathematical extension of the Rodin platform that is currently under development.

Despite certain technical difficulties, we have found the refinement approach as such to be beneficial for deriving precise requirements and the corresponding model of a multi-robotic system. In the refinement process, we have discovered a number of subtleties in the system requirements. The proving effort has helped us to localise the present problems and ambiguities and find the appropriate solutions. For instance, we had to impose extra restrictions on the situations when a base station takes a new responsibility for other zones and robots. Moreover, we had to make our assumptions about robot failures more precise.

Related work. Formal modelling of multi-agent systems has been undertaken in [10, 9, 11]. The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. Our modelling have pursued a different goal – we have aimed at formally guaranteeing that the specified agent behaviour achieves the pre-defined goals. Formal modelling of fault tolerant MAS in Event-B has been undertaken by Ball and Butler [3]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach, we have implemented a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability.

The foundational work on goal-oriented development has been done by van Lamsweerde [4]. The original motivation behind the goal-oriented development was to structure the requirements and derive properties in the form of temporal logic formulas that the system design should satisfy. Over the last decade, the goal-oriented approach has received several extensions that allow the designers to link it with formal modelling [5–7]. These works aimed at expressing temporal

logic properties in Event-B. In our work, we have relied on goals to facilitate structuring of the system behaviour and derived a detailed system model that satisfies the desired properties by refinement.

Conclusions. In this paper we have presented a formal development of a fault tolerant multi-robotic system. The development has been carried out by refinement in Event-B. As a result of the formal development process, we have achieved the desired goal – formally specified the complex system behaviour and proved the desired properties. The formal development has allowed us to uncover missing requirements and rigorously define the relationships between agents. The refinement approach has also allowed us to derive a complex mechanism for cooperative error recovery in a systematic manner.

Our approach has demonstrated a number of advantages comparing to various process-algebraic approaches used for modelling multi-agent systems. The reliance on a proof-based verification has allowed us to derive a quite complex model of the behaviour of a multi-agent robotic system. We have not needed to avoid complex data types and could comfortably express intricate relationships between the system goals and the employed agents. As a result, our approach scales well with respect to the number of system states, agents, and their complex interactions. We believe that, once the mentioned technical difficulties of handling complex nested functions are resolved in the Rodin platform, Event-B and the associated tool set will provide a suitable framework for formal modelling of complex multi-robotic systems.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
3. Ball, E., Butler, M.: Event-b patterns for specifying fault-tolerance in multi-agent interaction. In: Methods, Models and Tools for Fault Tolerance, pp. 104–129 (2009)
4. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: RE. pp. 249–263 (2001)
5. Landtsheer, R.D., Letier, E., van Lamsweerde, A.: Deriving tabular event-based specifications from goal-oriented requirements models. In: Requirements Engineering, 9(2). pp. 104–120 (2004)
6. Matoussi, A., Gervais, F., Laleau, R.: A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification. In: 16th International Conference on Engineering of Complex Computer Systems. IEEE (2011)
7. Ponsard, C., Dallons, G., Philippe, M.: From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In: ERCIM News (75). pp. 22–23 (2008)
8. Rodin: Event-B Platform, online at http://www.event-b.org/
9. Roman, G.C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In: FASE'2004. LNCS, vol. 2984. Springer (2004)
10. Roman, G.C., Julien, C., Payton, J.: Modeling adaptive behaviors in Context UNITY. In: Theoretical Computure Science. vol. 376, pp. 185–204 (2007)
11. Roman, G.C., P.McCann, Plun, J.: Mobile UNITY: Reasoning and Specification in Mobile Computing. In: ACM Transactions of Software Engineering and Methodology (1997)

Paper VII

Formal Development and Quantitative Assessment of a Resilient Multi-robotic System

Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna and Linas Laibinis

Originally published in: Anatoliy Gorbenko, Alexander Romanovsky, Vyacheslav S. Kharchenko (Eds.), *Proceedings of 5th International Workshop on Software Engineering for Resilient Systems (SERENE 2013)*, LNCS 8166, 109–124, Springer-Verlag Berlin Heidelberg, 2013.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-40894-6_9

Formal Development and Quantitative Assessment of a Resilient Multi-Robotic System

Anton Tarasyuk[2], Inna Pereverzeva[1,2],
Elena Troubitsyna[2], and Linas Laibinis[2]

[1] Turku Centre for Computer Science
[2] Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku, Finland
{anton.tarasyuk, inna.pereverzeva,
elena.troubitsyna, linas.laibinis}@abo.fi

Abstract. Ensuring resilience of multi-robotic systems is a notoriously difficult task. Decentralised architectures and asynchronous communication require powerful modelling techniques to demonstrate system resilience. In this paper, resilience of a multi-robotic system is defined as the ability to achieve goals despite robot failures. We demonstrate how to rigorously specify and verify essential properties of resilience mechanisms of multi-robotic systems by refinement in Event-B. To assess the desired resilience characteristics, we augment our formal models with statistical data and rely on probabilistic verification. The automated support provided by the PRISM model checker allows us to calculate the probability of goal reachability in the presence of robot failures and compare different reconfiguration strategies for selected architectures. We demonstrate our approach by a case study – development and assessment of a cleaning multi-robotic system.

Keywords: Formal modelling, resilience, Event-B, refinement, probabilistic model checking, multi-robotic system.

1 Introduction

Development and assessment of resilience – a property of a system to remain dependable despite changes [9] – of complex multi-robotic systems constitute a significant engineering challenge. Asynchronous communication, highly distributed architecture and a large number of components puts high scalability and expressiveness demands on the techniques for reasoning about resilience of multi-robotic systems. Typically, the behaviour of such systems is analysed using simulation. However, simulation allows us to validate the system behaviour only for selected scenarios, environments and architectural configurations. In this paper we propose an alternative approach to development and assessment of resilient multi-robotic systems. Our approach is based on combination of formal goal-oriented development by refinement in Event-B [1] and probabilistic model checking in PRISM [11].

Event-B is a formal top-down approach to correct-by-construction system development. Usually development starts from a high-level abstract specification that is transformed into a detailed model by a number of refinement steps. While developing multi-robotic systems, we start from a highly abstract model defining the main system goal. Our refinement steps unfold the system architecture and introduce the required resilience mechanisms. In our case study – a multi-robotic cleaning system this corresponds to specifying the behaviour of cleaning robots and supervising base stations both in nominal conditions and in the presence of failures. When a detailed logical architecture is derived by refinement, we augment the obtained model with the probabilistic information required to conduct probabilistic resilience assessment. We rely on the probabilistic model checker PRISM to assess the probability of achieving the goal as well as to compare several alternative system configurations. We believe that the proposed approach facilitates development and verification of complex multi-robotic systems. It complements our previous work [12] on fault tolerant multi-robotic systems by removing an artificial assumption that a perfect robot would be always available to achieve the system goal. Instead, in this paper we compute the actual probabilities of success for different architectural configurations.

The paper is structured as follows. In Section 2 we briefly overview the Event-B formalism. Section 3 describes the requirements for our case study – a multi-robotic cleaning system – and outlines the formal development strategy. Section 4 briefly presents a formal development of the cleaning system and demonstrates how to express and verify its properties in the refinement process. Section 5 describes quantitative assessment of the system goal reachability via probabilistic model checking in PRISM. Finally, in Section 6 we conclude by discussing the paper contribution and reviewing the related work.

2 Modelling and Refinement in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving [1]. In Event-B, a system model is specified using the notion of an *abstract state machine*. An abstract state machine encapsulates the model state represented as a collection of variables, and defines operations on this state, i.e., it describes the *behaviour* of the modelled system. A machine usually has the accompanying component, called *context*. A context may include user-defined carrier sets, constants and their properties (model *axioms*). In Event-B, the model variables are strongly typed by the constraining predicates called *invariants*. Moreover, the invariants specify important properties that should be preserved during the system execution.

The dynamic behaviour of the system is defined by a set of atomic *events*. An event is essentially a *guarded command* that, in the most general form, can be defined as follows:

$$\textbf{evt} \mathrel{\widehat{=}} \textbf{any}\ vl\ \textbf{where}\ g\ \textbf{then}\ S\ \textbf{end}$$

where vl is a list of new local variables, g is the *guard*, and S is the *action*. The guard is a state predicate that defines the conditions under which the action

can be executed. In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments.

Event-B employs a top-down *refinement-based* approach to system development. A development starts from an *abstract* system specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce non-determinism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we should define so called *gluing invariant* as a part of the invariant of the refined machine. The gluing invariant defines the relationship between the abstract and concrete variables.

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness. The consistency of Event-B models, i.e., invariant preservation, correctness of refinement steps, should be formally demonstrated by discharging the relevant proof obligations. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by Event-B tool support – the Rodin platform [14].

3 Case Study: a Multi-Robotic Cleaning System

We start by briefly describing our case study – a multi-robotic cleaning system – and formulating a formal development strategy for such a system.

3.1 Case Study Description

The *main goal* of the considered multi-robotic system is to get a certain territory cleaned by available *robots*. The whole territory is divided into several *zones*, which in turn are further divided into a number of *sectors*. Each zone has a *base station* that coordinates the cleaning activities within the zone. In general, the coordination activities of one base station may span over several zones. Moreover, each base station supervises a number of robots attached to it by assigning cleaning tasks to them.

A robot is an autonomous electro-mechanical device that can move and clean. A base station may assign a robot a specific sector to clean. Upon receiving an assignment, the robot autonomously moves to this sector and performs cleaning. After successfully completing its mission, the robot returns back to the base station to receive a new assignment. The base station keeps track of the cleaned and non-cleaned sectors. Moreover, the base stations periodically exchange the information about their cleaned sectors.

While performing the given task, a robot might fail. It subsequently leads to a failure to clean the assigned sector. We assume that a base station is able to detect all the failed robots attached to it. In case of a robot failure, the base station may assign another active robot to perform the failed task.

A base station might fail as well. We assume that a failure of a base station can be detected by the others stations. In that case, some healthy base stations redistribute control over the robots coordinated by the failed base station.

Below we formulate the main requirements and properties associated with the multi-robotic system informally described above.

(PR1) *The main system goal: the whole territory has to be cleaned.*
(PR2) *To clean the territory, every its zone has to be cleaned.*
(PR3) *To clean a zone, every its sector has to be cleaned.*
(PR4) *Every cleaned sector or zone remains cleaned during functioning of the system.*
(PR5) *No two robots should clean the same sector.* In other words, a robot gets only non-assigned and non-cleaned sectors to clean.
(PR6) *The information about the cleaned sectors stored in any base station has to be consistent with the current state of the territory.* More specifically, if a base station sees a particular sector in some zone as cleaned, then this sector is marked as cleaned in the memory of the base station responsible for it. Also, if a sector is marked as non-cleaned in the memory of the base station responsible for it, then any other base station considers it to be non-cleaned.
(PR7) *Base station cooperation: if a base station has been detected as failed then some base station will take the responsibility for all the zones and robots of the failed base station.*
(PR8) *Base station cooperation: if a base station has cleaned all its zones, its active robots may be reallocated under control of another base station.*

The last two requirements essentially describe the co-operative recovery mechanisms that we assume to be present in the described multi-robot system.

3.2 A Formal Development Strategy

In the next section we will present a formal Event-B development of the described multi-system robotic system. Let us now give a short overview of this development and highlight the formal techniques used to ensure the proposed properties (PR1)–(PR8).

We start with a very abstract model, essentially representing the system behaviour as a process iteratively trying to achieve the main goal (PR1). The next couple of data refinement steps decompose the main goal into a set of subgoals, i.e., reformulate it in terms of zones and sectors. We will define and prove the relevant gluing invariants establishing a formal relationship between goals and the corresponding subgoals.

Next we introduce different types of agents – base stations and robots. The base stations coordinate execution of the tasks required to achieve the corresponding subgoal, while the robots execute the tasks allocated to them. We formally define the relationships between different types of agents, as well as agents and the respective subgoals. These relationships are specified and proved as invariant properties of the model.

The consequent refinement steps explicitly introduce agent failures, the information exchange, as well as the co-operation activities between the agents. The integrity between the local and global information stored within base stations is again formulated and proved as model invariant properties.

We assume that communication between the base stations as well as the robots and the base stations is reliable. In other words, messages are always (eventually) transmitted correctly without any loss or errors.

4 Formal Development of a Multi-Robotic System

In this section, we briefly present a formal development of our case study, by only highlighting essential elements (i.e., data structures, events, proved properties) of models. The detailed description of the case study can be found in [12].

4.1 Modelling system goals and subgoals

Abstract model. The initial model abstractly represents the behaviour of the described multi-robotic system. Essentially, we aim at ensuring the property (PR1). We define a variable $goal \in STATE$ modelling the current state of the system goal, where $STATE = \{incompl, compl\}$. The value *compl* corresponds to the situation when the goal is achieved, i.e., the whole territory is cleaned.

In the process of achieving the goal, modelled by the event Body, the variable *goal* may eventually change its value from *incompl* to *compl*. The system continues its execution until the whole territory is not cleaned:

$$\mathsf{Body} \mathrel{\widehat{=}} \textbf{when } goal \neq compl \textbf{ then } goal :\in STATE \textbf{ end}$$

First and second refinement. First, we assume that the whole territory is divided into n zones, where $n \in \mathbb{N}^+$. We augment our model with a representation of subgoals and aim at ensuring the property (PR2). We associate the notion of a *subgoal* with the process of *cleaning a particular zone*. A subgoal is achieved only when the corresponding zone is cleaned. A new variable *zones* represents the current subgoal status for every zone, i.e., $zones \in 1..n \to STATE$.

To establish the relationship between goal and subgoals and formalise the property (PR2) per se, we formulate the following gluing invariant:

$$goal = compl \Leftrightarrow zones[1..n] = \{compl\}.$$

The invariant can be understood as follows: the territory is considered to be cleaned if and only if its every zone is cleaned. To model cleaning of a zone(s), we refine the abstract event Body. In the refined event we reflect on a fact that once a subgoal is reached, it stays reached. Hence we ensure the property (PR4).

In the second refinement step, we further decompose system subgoals into a set of subsubgoals. We assume that each zone in our system is divided into k sectors, where $k \in \mathbb{N}^+$, and aim at formalising the property (PR3). We establish the relationship between the notion of a subsubgoal (or simply *a task*) and the process of *cleaning a particular sector*. A task is completed when the corresponding sector is cleaned. A new variable *territory* represents the current status of each sector:

$$territory \in 1 \,..\, n \to (1 \,..\, k \to STATE).$$

The following gluing invariant expresses the relationship between subgoals and subsubgoals (tasks) and correspondingly ensures the property (PR3):

$$\forall j \cdot j \in 1 \,..\, n \Rightarrow (zones(j) = compl \Leftrightarrow territory(j)[1 \,..\, k] = \{compl\}).$$

The invariant says that a zone is cleaned if and only if its each sector is cleaned.

4.2 Introducing Agents

In the third refinement step, we augment our model with a representation of agents. The structure of the refined model is presented in Fig. 1. In the model context, we define an abstract finite set $AGENTS$ and its disjoint non-empty subsets RB and BS that represent the robots and the base stations respectively. New variables *responsible* and *attached* formalise, respectively, the relationship between a zone and its supervising base station and the relationship between a robot and the base station associated to it:

$$responsible \in 1\,..\,n \to BS, \quad attached \in RB \nrightarrow BS.$$

Here $\nrightarrow$ denotes a *partial* function, which reflects the fact that some robots may be not attached to any base station (e.g., failed).

To coordinate the cleaning process, a base station stores the information about its own cleaned sectors and updates the information about the status of the other cleaned sectors. We assume that each base station has a "map" – the knowledge about all sectors of the whole territory. To model this, we introduce a new variable *local_map*:

$$local\_map \in BS \to (1\,..\,n \nrightarrow (1\,..\,k \to STATE)).$$

The abstract variable *territory* represents the global knowledge on the whole territory. For any sector and zone this global knowledge has to be consistent with the information stored by the base stations. In particular, if in the local knowledge of any base station a sector is marked as cleaned, then it should be cleaned according to the global knowledge as well. To establish those relationships, we formulate and prove the following invariant:

$$\forall bs, z, s \cdot bs \in ran(responsible) \wedge z \in 1\,..\,n \wedge s \in 1\,..\,k \Rightarrow \\ (territory(z)(s) = incompl \Rightarrow local\_map(bs)(z)(s) = incompl).$$

Moreover, for each base station, its local information always coincides with the global knowledge about the corresponding zones and sectors:

$$\forall bs, z, s \cdot bs \in ran(responsible) \wedge z \in 1\,..\,n \wedge responsible(z) = bs \wedge s \in 1\,..\,k \Rightarrow \\ (territory(z)(s) = incompl \Leftrightarrow local\_map(bs)(z)(s) = incompl).$$

All together, these two invariants formalise the property (PR6).

A base station may only assign a cleaning task to its attached robots. Here, we have to ensure the property (PR5) – *no two robots can clean the certain sector at the same time*. We model this behaviour by a new event NewTask.

The robot failures have some impact on execution of the cleaning process. The task cannot be performed if the robot assigned to it has failed. To reflect this behaviour, we refine the event Body by two events TaskSuccess and TaskFailure, which respectively model successful and unsuccessful execution of the task.

A base station keeps track of the cleaned and non-cleaned sectors and repeatedly receives the information from the other base stations about their cleaned sectors. The knowledge is inaccurate for the time span when the information is

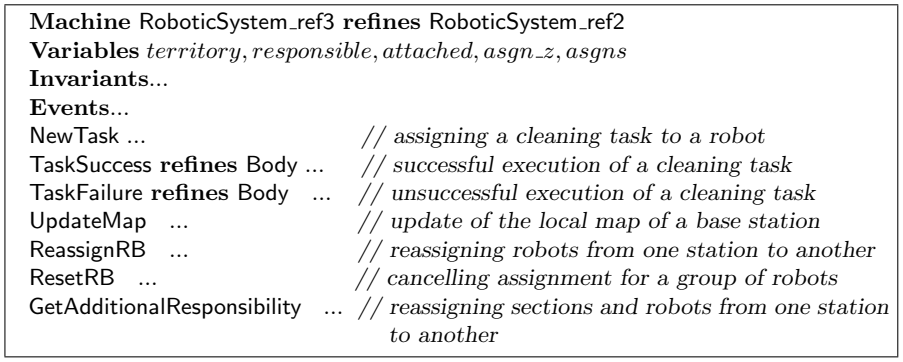


Fig. 1. Multi-Robotic System: the third refinement step

sent but not yet received. In this refinement step, we abstractly model receiving the information by a base station by introducing a new event UpdateMap.

In this refinement step, we also introduce an abstract representation of the base station cooperation defined by the property (PR8). Namely, we allow to reassign a group of robots from one base station to another. We define such a behaviour in a new event ReassignRB. In that case, all the robots of the base station may be sent to some other base station that still has some unfinished cleaning to co-ordinate. We also reserve a possibility to cancel all the current assignments for a group of robots in another new event ResetRB. This functionality will be needed later on, e.g., to describe the effect of base station failures.

Finally, we model a possible redistribution between the base stations their pre-assigned responsibility for zones and robots by a new event GetAdditionalResponsibility. Note that a base station can take the responsibility for a new zone only if it has the accurate knowledge about this zone, i.e., the information about its cleaned and non-cleaned sectors. Modelling this behaviour allows us to formalise the property (PR7).

4.3 Modelling of Broadcasting

In the next, fourth refinement step, we aim at defining an abstract model of broadcasting. After receiving a notification from a robot about successful cleaning the assigned sector, a base station updates its local map and broadcasts the message about the cleaned sector to the other base stations. In its turn, upon receiving the message, each base station correspondingly updates its own local map. A new relational variable msg models the message broadcasting buffer:

$$msg \in BS \leftrightarrow (1\,..\,n \times 1\,..\,k).$$

If a message $(bs \mapsto (z \mapsto s))$ belongs to this buffer then the sector s from the zone z has been cleaned. The first element of the message, bs, determines to which base station the message is sent. If there are no messages in the msg buffer for any particular base station then the local map of this base station is accurate,

i.e., it coincides with the global knowledge about the territory:

$$\forall bs, z, s \cdot z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \wedge bs \in ran(responsible) \wedge (bs \mapsto (z \mapsto s)) \notin msg \Rightarrow territory(z)(s) = local\_map(bs)(z)(s),$$

$$\forall bs \cdot bs \in ran(responsible) \wedge bs \notin dom(msg) \Rightarrow (\forall z, s \cdot z \in 1 \mathinner{..} n \wedge s \in 1 \mathinner{..} k \Rightarrow territory(z)(s) = local\_map(bs)(z)(s)).$$

After receiving a notification about successful cleaning of a sector, a base station marks this sector as cleaned in its local map and then broadcasts the message about it to other base stations. To model this, we refine the abstract events TaskSuccess and UpdateMap.

4.4 Introducing Robot and Base Station Failures

Fifth refinement. Now we aim at modelling possible robot failures. To achieve this, we partition the robots into active and failed ones. The current set of all active robots is defined by a new variable *active*. Initially all robots are active, i.e., $active = RB$. Moreover, new events RobotFailure and TaskFailure model respectively possible robot failures and their detection by the base stations. In our modelling, we assume that a robot may fail only during its cleaning assignments.

The events NewTask, ReassignRB and GetAdditionalResponsibility are now refined to reflect that only active robots can be given cleaning assignments and reattached to other base stations.

In our previous work [12], in order to verify goal reachability, we made an assumption that the last active robot and the last active base station can not fail. In this paper, we drop this artificial constraint and verify goal reachability by means of probabilistic model checking, as explained in Section 5.

Sixth refinement. In the final refinement step presented in the paper, we aim at specifying the base station failures. The structure of the final refined model is presented in Fig. 2.

Each base station might be either operating or failed. We introduce a new variable $operating \subseteq BS$ to define the set of all operating base stations. The event BaseStationFailure (extension of the abstract event ResetRB) models a possible base station failure. It removes the failed base station from the set *operating* and cancels all the cleaning assignments to the attached robots of the station.

Let us note that, once BaseStationFailure is executed, the events TaskSuccess and RobotFailure cannot be executed any more for this particular zone, i.e., the failure of a base station leads to interruption of all the cleaning activities performed by the robots that the station coordinates. Moreover, the event GetAdditionalResponsibility is now refined by introducing an additional condition – only if a base station is detected as failed, another base station can take over its responsibility for the respective zones and robots.

Once again, verification of reachability of the system goals despite possible failures of base stations is addressed by probabilistic model checking presented in Section 5.

Machine RoboticSystem_ref6 **refines** RoboticSystem_ref5
Variables $territory, responsible, attached, asgn\_z, asgns$
Invariants ...
Events ...
NewTask **refines** NewTask ... // *assigning a cleaning task to a robot*
TaskSuccess **refines** TaskSuccess ... // *successful execution of a cleaning task*
TaskFailure **refines** TaskFailure ... // *unsuccessful execution of a cleaning task*
UpdateMap **refines** UpdateMap ... // *update of the local map of a base station*
RobotFailure **refines** RobotFailure ... // *robot failure*
BaseStationFailure **refines** ResetRB ... // *base station failure*
ReassignRB **refines** ReassignRB ... // *reassigning robots from one station to another*
GetAdditionalResponsibility **refines** GetAdditionalResponsibility ...
// *reassigning sections and robots from one station to another*

Fig. 2. Multi-Robotic System: the sixth refinement step

4.5 Discussion

As a result of the presented refinement chain, we arrived at a centralised model of the multi-robotic system. We can further refine the system to derive its distributed implementation, relying on the modularisation extension of Event-B to achieve this.

The development of the presented multi-robotic system has been carried out with the support of the Rodin platform. To verify correctness of the models, we have discharged more than 230 proof obligations. Around 80% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. As s result, we have derived a complex system specification in six refinement steps. In general, the refinement approach has demonstrated a good scalability and allowed us to model intricate dependencies between the system components. We have been able to express and verify all the desired properties defined for our system.

In the refinement process, we have discovered a number of subtleties in the system requirements. The proving effort has helped us to localise the present problems and ambiguities and find the appropriate solutions. For instance, we had to impose extra restrictions on the situations when a base station takes a new responsibility for other zones and robots.

In contrast to our previous work, we did not postulate the goal reachability property nor make any assumptions related to robot and base station failures. To evaluate goal reachability properties in the presence of agent failures, we employ quantitative assessment – probabilistic model checking techniques, which we discuss in the next section.

5 Quantitative Assessment

In this paper, we aim at applying probabilistic model checking for the quantitative reasoning about goal reachability of a resilient multi-robotic system modelled in Event-B. To achieve this, we use the probabilistic symbolic model checker PRISM [11] – one of the leading software tools in the domain of formal modelling and verification of probabilistic systems.

5.1 Probabilistic Model of a Multi-Robotic Cleaning System

To enable probabilistic analysis of Event-B models in PRISM, we rely on the continuous-time probabilistic extension of the Event-B framework [17]. This extension allows us to annotate actions of all model events with real-valued rates and then transform a probabilistically augmented Event-B specification into a continuous-time Markov chain. It also implicitly introduces the notion of time into Event-B models: for any state, the sum of action rates of all enabled in this state events defines a parameter of the exponentially distributed time delay that takes place before some enabled action is triggered.

We assume that all the base station and robots in our multi-robotic system are identical and define five action rates as constants in its PRISM model (see Fig. 3). Specifically, we define the cleaning task assignment rate (λ) of a base station, the failure rates of a base station (δ) and a robot (γ), the robot's service (cleaning) rate (μ) and, finally, the reconfiguration rate (τ) – the rate of reassigning the robots and sectors (if any) to another station.

To apply model checking for verification of our robotic system, we also have to instantiate some constants declared in the model's context. In particular, we need to specify the total number of sectors in the territory as well as the initial number of active robots. In the PRISM specification shown in Fig. 3, these two values equal to 60 and 12 correspondingly. Furthermore, we need to evenly distribute the sectors and robots among several zones. Here we consider two different arrangements, namely, the territories consisting of two and three zones. In our PRISM models, we represent the behaviour of each separate zone as a single module. As a result, in the model with three zones there are three identical – up to variable names and synchronisation labels – modules $Station_1$, $Station_2$ and $Station_3$. In addition, the module *End* models system deadlock in one of three possible terminating states: the successfully accomplished main goal, failure of all base stations and failure of all the available robots.

Let us now consider in more detail the module $Station_1$ (Fig. 3). Its first three commands model respectively (1) failure of a base station, (2) assigning of a new task to an idle robot, and (3) task completion or failure of an assigned robot. We assume that all the system failures are permanent, i.e., there is no possibility to repair a failed base station or a robot. The next two pairs of guarded commands correspond to the event ReassignRB and describe system reconfiguration after some other base station has achieved its subgoal. Specifically, the commands (4) and (5) represent transferring robots from the second station to the first one, while the (6) and (7) represent reassigning the robots from the third station.

Similarly, the guarded commands from the (8) to (11) correspond to the event GetAdditionalResponsibility and model reassigning of both robots and non-cleaned sectors in the case of failure of the second or third base station. Let us describe how the reconfiguration procedure is resolved if there are *two* potential target stations capable of accepting new robots (and sectors): *operational* base stations that have their own *unfinished tasks* "compete" for getting new robots (and tasks), and the reassignment rate for each base station is $\tau/2$. Obviously, in such a case the time delay required to perform the reconfiguration procedure

const double $\lambda = 0.2$; *// task (sector) assignment rate*
const double $\delta = 0.0007$; *// base station failure rate*
const double $\mu = 0.035$; *// robot service (work) rate*
const double $\gamma = 0.003$; *// robot failure rate*
const double $\tau = 0.07$; *// robots (and sectors) reassignment rate*

const int $n = 3$; *// number of zones (and base stations)*
const int $k = 20$; *// initial number of sectors in each zone*
const int $m = 4$; *// initial number robots in each zone*

module $Station_1$

z_1 : **bool init true**; *// base station's status (***true***=operational,* **false***=failed)*
$s_1 : [0..k \cdot n]$ **init** k; *// number of unfinished tasks*
$r_1 : [0..m \cdot n]$ **init** m; *// number of active robots*
$a_1 : [0..m \cdot n]$ **init** 0; *// number of currently assigned robots*

(1) [] $z_1 \,\&\, s_1 + s_2 + s_3 > 0 \,\&\, r_1 + r_2 + r_3 > 0 \rightarrow \delta : (z_1' = \textbf{false}) \,\&\, (a_1' = 0)$;
(2) [] $z_1 \,\&\, a_1 < r_1 \,\&\, a_1 < s_1 \rightarrow \lambda : (a_1' = a_1 + 1)$;
(3) [] $z_1 \,\&\, a_1 > 0 \,\&\, s_1 > 0 \,\&\, r_1 > 0 \rightarrow a_1 \cdot \mu : (s_1' = s_1 - 1) \,\&\, (a_1' = a_1 - 1) +$
$a_1 \cdot \gamma : (r_1' = r_1 - 1) \,\&\, (a_1' = a_1 - 1)$;

(4) $[tr21]\ z_1 \,\&\, s_1 > 0 \,\&\, s_2 = 0 \,\&\, r_2 > 0 \,\&\, (!z_3 \mid s_3 = 0) \,\&\, r_1 + r_2 \leq m \cdot n \rightarrow \tau : (r_1' = r_1 + r_2)$;
(5) $[tr21]\ z_1 \,\&\, s_1 > 0 \,\&\, s_2 = 0 \,\&\, r_2 > 0 \,\&\, z_3 \,\&\, s_3 > 0 \,\&\, r_1 + r_2 \leq m \cdot n \rightarrow \tau/2 : (r_1' = r_1 + r_2)$;

(6) $[tr31]\ z_1 \,\&\, s_1 > 0 \,\&\, s_3 = 0 \,\&\, r_3 > 0 \,\&\, (!z_2 \mid s_2 = 0) \,\&\, r_1 + r_3 \leq m \cdot n \rightarrow \tau : (r_1' = r_1 + r_3)$;
(7) $[tr31]\ z_1 \,\&\, s_1 > 0 \,\&\, s_3 = 0 \,\&\, r_3 > 0 \,\&\, z_2 \,\&\, s_2 > 0 \,\&\, r_1 + r_3 \leq m \cdot n \rightarrow \tau/2 : (r_1' = r_1 + r_3)$;

(8) $[tr21]\ z_1 \,\&\, !z_2 \,\&\, s_2 > 0 \,\&\, (!z_3 \mid s_3 = 0) \,\&\, (r_1 + r_2 \leq m \cdot n) \,\&\, (s_1 + s_2 \leq k \cdot n) \rightarrow$
$\tau : (s_1' = s_1 + s_2) \,\&\, (r_1' = r_1 + r_2)$;
(9) $[tr21]\ z_1 \,\&\, !z_2 \,\&\, s_2 > 0 \,\&\, z_3 \,\&\, s_3 > 0 \,\&\, (r_1 + r_2 \leq m \cdot n) \,\&\, (s_1 + s_2 \leq k \cdot n) \rightarrow$
$\tau/2 : (s_1' = s_1 + s_2) \,\&\, (r_1' = r_1 + r_2)$;

(10) $[tr31]\ z_1 \,\&\, !z_3 \,\&\, s_3 > 0 \,\&\, (!z_2 \mid s_2 = 0) \,\&\, (r_1 + r_3 \leq m \cdot n) \,\&\, (s_1 + s_3 \leq k \cdot n) \rightarrow$
$\tau : (s_1' = s_1 + s_3) \,\&\, (r_1' = r_1 + r_3)$;
(11) $[tr31]\ z_1 \,\&\, !z_3 \,\&\, s_3 > 0 \,\&\, z_2 \,\&\, s_2 > 0 \,\&\, (r_1 + r_3 \leq m \cdot n) \,\&\, (s_1 + s_3 \leq k \cdot n) \rightarrow$
$\tau/2 : (s_1' = s_1 + s_3) \,\&\, (r_1' = r_1 + r_3)$;

(12) $[tr12]$ **true** $\rightarrow 1 : (r_1' = 0) \,\&\, (s_1' = 0)$;
(13) $[tr13]$ **true** $\rightarrow 1 : (r_1' = 0) \,\&\, (s_1' = 0)$;

endmodule

...

module *End*

[] $(!z_1 \,\&\, !z_2 \,\&\, !z_3) \mid r_1 + r_2 + r_3 = 0 \mid s_1 + s_2 + s_3 = 0 \rightarrow$ **true**;

endmodule

Fig. 3. PRISM model: 3 zones with 20 sectors and 4 robot in each zone initially

is exponentially distributed with parameter τ. Finally, the last two commands – (12) and (13) – of the module $Station_1$ are required to reset the number of station's active robots and unfinished tasks to zero while reassigning them to another station. To synchronise the commands modelling reassignment of robots (and sectors) from $Station_i$ to $Station_j$, we label them with actions trij.

Note that our PRISM specification has less "events" than its Event-B counterpart. To reduce the size of the model, we have suppressed, yet without loss of generality, those Event-B events that do not affect the non-functional system behaviour. Usually, such events represent certain steps of the functional system behaviour that must be individually addressed in an Event-B model, yet, from the point of view of the non-functional behaviour, they can be considered as a whole. For instance, in our PRISM model, the event TaskFailure is omitted because task failure is implicitly modelled by the event RobotFailure, which is characterised by the failure rate γ. Moreover, we abstract away from modelling the broadcasting as the communication channel is perfectly reliable and, conse-

quently, any broadcasted message will be eventually delivered. Hence we assume that, in the case of system reconfiguration, the delay required to update the local knowledge of a base station (UpdateMap) is covered by the rate τ.

5.2 Probabilistic Reasoning about Goal Reachability

Our first objective is to compute the probability that the system goal – *cleaning of the whole territory* – is *eventually* reachable. We have mentioned earlier that we consider two different system configurations – the territory partitioned into two and three zones. Each zone has a single base station. In the first configuration each zone initially has 30 sectors and 6 robots, while in the second one each zone has 20 sectors and 4 robots. In the PRISM property specification language, the eventual goal reachability for our models is defined as the following CSL (Continuous stochastic logic) formulae

$$\mathbf{P}_{=?}\,\{\mathbf{F}\; s_1 + s_2 = 0\} \quad \text{and} \quad \mathbf{P}_{=?}\,\{\mathbf{F}\; s_1 + s_2 + s_3 = 0\}.$$

After verifying these formulae in the PRISM model checker, we can state that the probability to eventually clean the whole territory, with the work and failure rates as given in Fig. 3, is 0.939 for the first system configuration and 0.979 for the second one.

In addition, it is interesting to assess the sources of system failure, i.e., to check how robots and base station failures affect goal reachability. The negative outcomes *all robots have failed* and *all base stations have failed* (for the case of three zones) can be specified in the PRISM language as

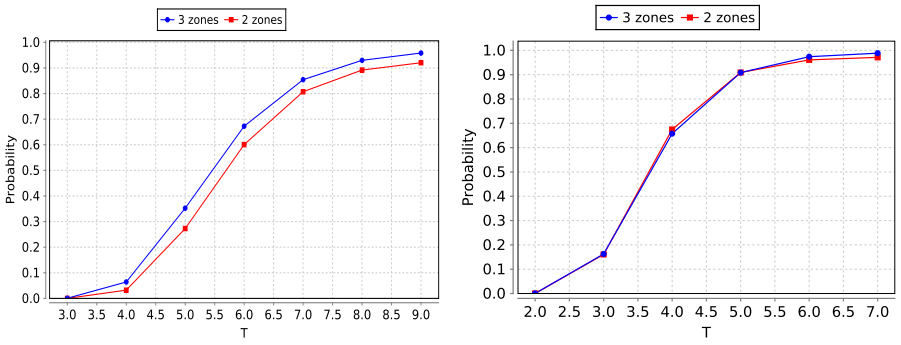
$$\mathbf{P}_{=?}\,\{\mathbf{F}\; r_1 + r_2 + r_3 = 0\} \quad \text{and} \quad \mathbf{P}_{=?}\,\{\mathbf{F}\; !z_1 \,\&\, !z_2 \,\&\, !z_3\}.$$

The verification results for the 2-zones and 3-zones configurations are 0.008, 0.053 and 0.009, 0.012 respectively. It is easy to see that, for the given size of the model and probabilistic characteristics of agents, the arrangement with three zones is slightly better, and that in both cases the goal *unreachability* is mostly induced by failures of the base stations.

Moreover, one can be interested in not only the eventual goal reachability but also in performance of the system. This can be especially important when the system must achieve its goal within a specified time interval. PRISM provides us with means for such kind of analysis. Specifically, we define two formulae

$$\mathbf{P}_{=?}\,\{\mathbf{F}^{\leq T}\; s_1 + s_2 = 0\} \quad \text{and} \quad \mathbf{P}_{=?}\,\{\mathbf{F}^{\leq T}\; s_1 + s_2 + s_3 = 0\}$$

which we will use to verify the *time-bounded* reachability of the main system goal. After specifying a desirable upper time bound T, we can analyse goal reachability progress over the time interval $[0, T]$. However, sometimes it is also useful to identify a lower time bound T_0 since often there is a period of time $[0, T_0]$ during which the probability to reach the goal is negligible. This can be done by repeatedly verifying the formulae above while gradually increasing the constant value T. After the lower bound is defined, in the same manner one can define a suitable value for the upper bound T (if the one is not predefined) and run necessary experiments in PRISM for the time interval $[T_0, T]$.



(a) $\lambda = 0.2,\ \delta = 0.0007,\ \mu = 0.035,$ $\gamma = 0.003,\ \tau = 0.07$

(b) $\lambda = 0.5\ \delta = 0.0007,\ \mu = 0.043,$ $\gamma = 0.003,\ \tau = 0.1$

Fig. 4. Case study: results of probabilistic analysis by PRISM

Fig. 4 shows the verification results for two different sets of probabilistic characteristics of agents. These results can be understood as follows. Fig. 4 (a) shows the goal reachability progress within the time interval from 3 to 9 hours of work, while time units used in the definition of rates are minutes (e.g., the expected time required for a robot to clean one sector is approximately 29 minutes). The similar interpretation is apparently valid for Fig. 4 (b).

We need to say that, even for the reduced models considered in this paper, the state space explosion problem is very prominent. For a relatively small territory and quite a small number of robots – 60 and 12 respectively – the size of the model grows from approximately 10^6 states and $6 \cdot 10^6$ transitions in the case of two zones to approximately $7 \cdot 10^7$ states and more than $5 \cdot 10^8$ transitions in the case of three zones. For the latter model, the verification process, especially concerning the time-bounded properties, is quite time-consuming. As a matter of fact, the use of probabilistic model checking to analyse of a multi-robotic system that has a larger territory and more than three base stations is getting rather problematic. It is also worth to mention that for the conducted case study the *sparse* computation engine of PRISM demonstrated the best performance in terms of model checking time.

6 Conclusions and Related Work

In this paper, we have presented an integrated approach to development and assessment of resilient multi-robotic systems. We have demonstrated how the proposed approach can be applied to develop and assess a cleaning multi-robotic system. Our approach combines the strengths of two formal techniques – refinement and probabilistic model checking – to achieve scalability and expressiveness required for reasoning about resilience of multi-robotic systems. Indeed, formal development by refinement allows us derive a complex system architecture, formally verify correctness of robot interactions as well as logical properties of the incorporated resilience mechanisms. On the other hand, probabilistic model checking allows us to reason about the probability of achieving the system goal

despite robot failures as well as compare the resilience characteristics of different configurations. An integration with probabilistic model checking has allowed us to avoid an artificial assumption about the presence of a perfect robot that was required to prove goal reachability in [12]. PRISM could cope sufficiently well with scalability challenge for reasonably sized systems, yet its application to very large multi-agent systems typical for, e.g., Internet of Things, would be limited. We argue that our approach allows us to achieve a certain degree of generality in the development and assessment – the models can be reused for a wide class of multi-robotic systems that comply to the defined architectural style.

Formal modelling of MAS has been undertaken in [16, 15]. The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. Our modelling have pursued a different goal – we have aimed at formally guaranteeing that the specified agent behaviour achieves the pre-defined goals. Formal modelling of fault tolerant MAS in Event-B has been also undertaken by Ball and Butler [2]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach, we have implemented a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability.

The foundational work on goal-oriented development has been done by van Lamsweerde [7]. The original motivation behind the goal-oriented development was to structure the system requirements and derive properties in the form of temporal logic formulae. Over the last decade, the goal-oriented approach has received several extensions that allow the designers to link it with formal modelling [8, 13]. These works aimed at expressing temporal logic properties in Event-B. In our work, we have relied on goals to facilitate structuring of the system behaviour and, as a result, derived a detailed system model that satisfies the desired properties by refinement.

The use of model checking techniques for reasoning about MAS properties has been largely studied in MAS research communities (see, e.g., [3, 4, 10]). In particular, [4] presents a framework for verification of agent programs against BDI (belief-desire-intention) agent specifications. In the proposed approach, an agent system is first programmed using the logic-based agent oriented programming language AgentSpeak(F). Then the AgentSpeak(F) programs are translated into Promela – the specification language of the SPIN model checker – to verify the resulting system. The paper [10] presents the symbolic model checker MCMAS, specifically tailored for verification of MAS. The MCMAS tool takes as inputs models written in the Interpreted Systems Programming Language, which allows for describing both agents and working environment of a multi-agent system. MCMAS also benefits from the dedicated specification language that supports, in addition to the traditionally used Computation tree logic, the epistemic logic that has proved useful in the robotics domain. The discussed approaches are illustrated by many case studies from different domains. However, unlike the PRISM model checker, none of the above techniques provide the means for probabilistic assessment of the system behaviour.

There is a number of papers that discuss the use of probabilistic model techniques for analysis of multi-agents systems. For example, in [6], the authors address model-checking of probabilistic knowledge (relative to the agent knowledge) by developing an algorithm in the MCK model checker, while in [5], the authors represent a MAS as a discrete-time Markov chain and verify such system properties as convergence and convergence rate in the PAT model checker. Yet, to the best of our knowledge, the approaches combining both theorem proving and verification via model checking are still scarce.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. Ball, E., Butler, M.: Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In: Methods, Models and Tools for Fault Tolerance, pp. 104–129. Springer (2009)
3. Bordini, R., Fisher, M., Pardavila, C., Wooldridge, M.: Model Checking AgentSpeak. In: AAMAS 2003. pp. 409–416. ACM Press (2003)
4. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-agent Programs by Model Checking. Autonomous Agents and Multi-Agent Systems 12(2), 239–256 (2006)
5. Hao, J., Song, S., Liu, Y., Sun, J., Gui, L., Dong, J.S., Leung, H.: Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In: PRIMA 2012. LNCS, vol. 7455, pp. 16–30. Springer (2012)
6. Huang, X., Luo, C., van der Meyden, R.: Symbolic Model Checking of Probabilistic Knowledge. In: TARK 2011. pp. 177–186. ACM (2011)
7. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: RE'01. pp. 249–263. IEEE Computer Society (2001)
8. Landtsheer, R.D., Letier, E., van Lamsweerde, A.: Deriving tabular event-based specifications from goal-oriented requirements models. In: Requirements Engineering, 9(2). pp. 104–120 (2004)
9. Laprie, J.: From Dependability to Resilience. In: DSN 2008. pp. G8–G9 (2008)
10. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In: CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer (2009)
11. M. Kwiatkowska, G.N., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV 2011. pp. 585–591. Springer (2011)
12. Pereverzeva, I., Troubitsyna, E., Laibinis, L.: A Case Study in Formal Development of a Fault Tolerant Multi-Robotic System. In: SERENE 2012. LNCS, vol. 7527, pp. 16–31. Springer (2012)
13. Ponsard, C., Dallons, G., Philippe, M.: From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In: ERCIM News (75). pp. 22–23 (2008)
14. Rodin: Event-B Platform, online at http://www.event-b.org/
15. Roman, G.C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In: FASE 2004. LNCS, vol. 2984. Springer (2004)
16. Roman, G.C., Julien, C., Payton, J.: Modeling adaptive behaviors in Context UNITY. In: Theoretical Computure Science. vol. 376, pp. 185–204 (2007)
17. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In: IFM 2012. pp. 237–252. Springer (2012)

Paper VIII

Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems

Linas Laibinis, Inna Pereverzeva and Elena Troubitsyna

Originally published in: TUCS Technical Report 1133, TUCS, April 2015.
(Submitted in *Formal Aspects of Computing Journal.*)

Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems

Linas Laibinis
Åbo Akademi University, Department of Computer Science
linas.laibinis@abo.fi
Inna Pereverzeva
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science
inna.pereverzeva@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
elena.troubitsyna@abo.fi

TUCS Technical Report
No 1133, April 2015

Abstract

In this paper we present our formalisation of a resilient goal-oriented multi-agent system and its essential properties. The formalisation covers the notions of system goals and agents, various formal structures (functions and relations) defining different interrelationships between these notions, as well as constraints on the system dynamics allowing a multi-agent system to become more reconfigurable and thus resilient in order to achieve the system goals. The formalisation results in establishing connections between goals at different levels of abstraction, system architecture and agent responsibilities. The proposed formal systematisation of the involved concepts can be seen as generic guidelines for formal development of reconfigurable systems. Moreover, we demonstrate how such guidelines can be interpreted within the Event-B framework.

Keywords: Formal reasoning, Multi-agent system, Goal-oriented development, Resilience, Event-B

TUCS Laboratory
Embedded Systems Laboratory

1 Introduction

Resilience is an ability of a system to remain trustworthy despite changes [15]. It is an evolution of dependability concept that puts an emphasis on the ability of a system to adapt to different operating conditions. In this paper, we view adaptability as an ability of a system to reconfigure and continue to function in the presence of faults and other changes. Our aim is to propose a comprehensive theoretical study of relevant aspects of the system architecture and dynamic behaviour to facilitate formal development of reconfigurable distributed systems.

We consider distributed systems that are composed of asynchronously communicating heterogeneous components. The components interact with each other to execute functions required from the system. Moreover, to facilitate system resilience, the system components cooperatively perform fault tolerance activities as well as exchange information about their current status. The cooperative nature of the component behaviour makes it convenient to consider them as collaborating agents and the overall distributed system as a multi-agent system correspondingly.

Often research on multi-agent systems focuses on studying the emerging behaviour, i.e., it adopts a bottom-up approach that investigates whether agent interactions give rise to the desired behaviour or properties. In our work, we take an opposite approach: we aim at deriving the architectural and behavioural constraints to guarantee system resilience, i.e., ensure that the system, besides correct execution of its functions in the nominal conditions, can also reconfigure and remain operational in the presence of faults and other changes.

We rely on the goal-oriented paradigm because it provides us with a suitable conceptual basis for our reasoning. Goals are functional and non-functional[1] objectives that the system should achieve [34, 36]. High-level goals representing the overall system objectives can be decomposed into sub-goals. Decomposition facilitates unfolding of the layered system architecture and reasoning about system properties at different levels of abstraction. It also allows us to eventually derive constraints on the agent behaviour and ensure that their collaboration guarantees achieving the desired goals. The goal-oriented framework also provides us with an especially suitable basis for reasoning about reconfigurability. In particular, it allows us to define reconfigurability as an ability of agents to redistribute their responsibilities to ensure goal reachability.

Reasoning about reconfigurability within the goal-oriented multi-agent framework spans over a large set of inter-twined concepts addressing both system architecture and its dynamic behaviour. Therefore, there is a clear need for a formal systematic study of these complex interdependencies. This

[1]The non-functional aspect is not considered in the paper.

is the task that we tackle in this paper. Namely, we propose a systematic set-theoretic formalisation of the reconfigurability concept for multi-agent goal-oriented framework. The formalisation results in establishing connections between goals at different levels of abstraction, system architecture and agent responsibilities. The proposed formal systematisation of these concepts can be also seen as generic guidelines for formal development of reconfigurable systems. In this paper, we demonstrate how such guidelines can be interpreted within the Event-B framework [1].

The paper is structured as follows. Section 2 overviews the kind of systems and their properties we are interested in studying and briefly describes an illustrative example of such systems – a multi-robotic cleaning system. In Section 3 we gradually present our formalisation of a resilient goal-oriented multi-agent system and its reconfiguration mechanisms. Section 4 discusses how the formalised notions can be represented in a concrete formal framework – Event-B. Finally, we overview the related work and give some concluding remarks in Section 5.

2 Resilient Goal-Oriented Multi-Agent Systems

Resilience is an ability of a system to remain trustworthy despite changes [15]. To react on such changes, the system needs to reconfigure. The reconfiguration might be reactive or proactive. In the former case, reconfiguration is usually triggered by a component failure and the system should reconfigure to achieve fault tolerance, i.e., perform error recovery. In the latter case, the system might attempt to execute some of its services more efficiently, e.g., by deploying the available idle components. In both cases, the system components should collaborate to ensure system resilience.

In this paper, we study reconfigurability as an essential mechanism of achieving resilience of distributed systems. Since the collaborative aspect of the component behaviour is important for our study, we represent system components as agents and the overall system as a multi-agent system correspondingly.

Agents are autonomous software components that asynchronously communicate with each other. Each agent has a certain functionality that it provides. In this paper, we consider heterogeneous multi-agent systems, i.e., agents may have different functionalities. Moreover, some agents might play a role of supervisors of another agent or a group of agents. As a result of reconfiguration, an agent might receive additional responsibilities, i.e., it should become involved into an execution of tasks that were not allocated on it initially. We assume that agents are co-operative, i.e., they always accept new responsibilities. At the same time, the agents are unreliable, i.e., they

might fail and cease performing their functions. This might trigger system reconfiguration. As a result, the responsibilities of the failed agents can be re-allocated to the healthy ones. If an agent is healthy and idle, it can be deployed to perform the functions of failed agents or it might also become engaged into an execution of some other task, e.g., to improve the system performance and/or increase the likelihood of successful task completion.

While developing a multi-agent system, we should establish a link between system requirements and the agent behaviour. It is widely recognised that the goal-oriented development framework facilitates achieving this. The key concept of the framework is the notion of a goal – a functional or non-functional objective that a system should satisfy. Goals also constitute a convenient mechanism for structuring requirements via goal decomposition. In the decomposition process, the high-level system goals are iteratively decomposed into subgoals. Moreover, the low-level subgoals can be directly linked with the behaviour of agents, i.e., they can be used to derive requirements and constraints on the agent behaviour.

The goal-oriented framework provides us with a suitable basis for reasoning about reconfigurable multi-agent systems. It enables reasoning about the system behaviour at different levels of abstraction. At the same time, goal-decomposition process facilitates incremental unfolding of the system architecture. It also helps us to build a hierarchy of agents according to their responsibilities in achieving certain kind of goals. Moreover, the goal-oriented framework allows us to formulate reconfigurability as an ability of agents to redistribute their responsibilities to ensure goal reachability.

To summarise, in the rest of the paper we aim at studying the systems that have the following characteristics:

- There is a number of main (global) goals defined for the system. The goals can be decomposed into a subset of corresponding subgoals;
- The system consists of a number of agents – autonomic software components;
- The agents are organised hierarchically, i.e., one agent may be a supervisor of one or a group of other agents;
- The agents interact with each other in order to achieve the system goals;
- In general, agent interactions can vary from simple information exchanges to requests for specific actions or services to be performed;
- The system agents are unreliable components that might fail during system execution;

- In the case of agents failures, the system should, if possible, dynamically reconfigure itself to achieve the overall system goal;

- The system can also reconfigure to achieve some of its goals more efficiently by means of, e.g., deploying idle agents.

Next we describe our running example – a *multi-robotic cleaning system* – that can be considered as an illustrative instance of the systems whose properties we described above.

A Multi-Robotic Cleaning System. The *main goal* of the multi-robotic system is to get a certain area cleaned by the means of involved system agents. There are two *types of agents* that are responsible for achieving the goal. The first type is base stations – the stationary devices that coordinate cleaning activities by assigning cleaning tasks to the second type of agents – robots. The robots are autonomous electro-mechanical devices that can move, clean, as well as communicate with the base stations. Both base stations and robots are unreliable, i.e., they can fail at any moment. In the case of these failures, the system should, if possible, *dynamically reconfigure* itself to achieve the overall system goal.

The whole territory to clean is divided into several zones, which are further divided into a number of sectors. To clean the territory, every its zone has to be cleaned. In its turn, to clean a zone, every its sector has to be cleaned. Each zone has the associated base station that *coordinates* the cleaning activities within the zone. In general, the coordination activities of one base station may span over several zones. Moreover, each base station *supervises* a number of robots attached to it by *assigning* cleaning tasks to them.

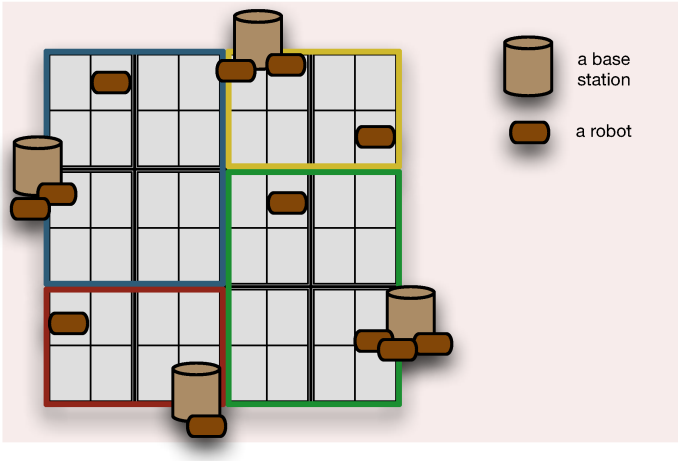


Figure 1: Multi-Robotic Cleaning System

A base station might assign a robot a specific sector to clean. Upon receiving a cleaning assignment, the robot autonomously moves to this sector and performs cleaning. After successfully completing its mission, the robot returns back to the base station to receive a new assignment. The base station keeps track of the cleaned and non-cleaned sectors. Moreover, the base stations periodically exchange the information about their cleaned sectors.

While performing a given task, a robot might fail at any moment. In that case, the base station may assign another active robot to perform the failed task. A base station might fail as well. In that case, the healthy base stations should redistribute the responsibility over the zones as well as the control over the associated robots of the failed base station.

As we can see, such a multi-robotic system exhibits the general characteristics and properties that we described above. We are going to use this system as the running example for the rest of this paper.

3 Formalisation of a Resilient Goal-Oriented MAS

In this section we present our formalisation of a resilient goal-oriented system and its essential properties. The formalisation will cover the notions of system goals and agents, various formal structures (functions and relations) defining different interrelationships between these notions, as well as constraints on the system dynamics allowing a multi-agent system to become more reconfigurable and thus resilient in order to achieve the system goals. The formalisation summarises our experience in formal modelling and verification of resilient goal-oriented multi-agent systems [22, 23, 24, 32, 13].

Notational conventions. In addition to the standard set-theoretical notation we are going rely on (e.g., $\in, \subseteq, \cap, \cup, \varnothing$, etc), the operator $\setminus$ is used to denote set subtraction. $T_1 \times T_2$ is a cartesian product of two types (sets) T_1 and T_2. The notation $\mathcal{P}(T)$ stands for the powerset (set of all subsets) type over elements of the type T, while $T_1 \leftrightarrow T_2$ denotes a relation between elements of two types (sets), i.e.,

$$R : T_1 \leftrightarrow T_2 \;\Leftrightarrow\; R : \mathcal{P}(T_1 \times T_2).$$

Moreover, **dom** and **ran** are respectively the relation domain and range operators, while $R_1; R_2$ stands for relational composition of two relations R_1 and R_2. Id represents the identity relation, while R^* denotes the reflexive transitive closure of a relation R, i.e.,

$$R^* \;=\; Id \cup R \cup R;R \cup R;R;R \cup \ldots .$$

We will also use the transitive closure of a relation R, denoted as R^+ and defined as $R^+ = R^* \backslash Id$, or

$$R^+ = R \cup R;R \cup R;R;R \cup \ldots .$$

We will treat functions as a special kind of relations and relations as a special kind of sets (i.e., sets of pairs, triples, etc). The notation $(e_1 \mapsto e_2) \in R$ will be used to check that two elements are related by the binary relation or function R. Similarly, $(e_1 \mapsto e_2 \mapsto e_3) \in R$ will be used for checking membership in a ternary relation.

3.1 A Goal-oriented State Transition System

We are going to build our formalisation of a goal-oriented multi-agent system by gradually extending the standard definition of a state transition system, typically defined as a triple $(\Sigma, \mathit{Init}, \mathit{Trans})$, where Σ represents all the system states, *Init* stands for its possible initial states, and *Trans* defines all the allowed transitions between system states. We start by introducing the notion of system goals that such a state transition system should try to achieve.

Definition 1 Goal-oriented state transition system (GSTS). *A GSTS system is a tuple $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$, where $\mathcal{G}$ is a set of all possible system goals, Σ is the system state space, Init is a set of initial system states, Trans is a next-state relation of a GSTS system, and GMap is a function mapping a system goal to a subset of system states, such that*

(1.1) $\mathit{Init} : \mathcal{P}(\Sigma)$,
(1.2) $\mathit{Trans} : \Sigma \leftrightarrow \Sigma$,
(1.3) $\mathit{GMap} : \mathcal{G} \to \mathcal{P}(\Sigma)$,
(1.4) $\mathit{Init} \neq \varnothing$,
(1.5) $\mathit{Init} \subseteq \mathbf{dom}(\mathit{Trans})$,
(1.6) $\forall g : \mathcal{G}.\ \mathit{GMap}(g) \neq \varnothing$,
(1.7) $\forall g : \mathcal{G}.\ \exists \sigma, \sigma' : \Sigma.\ \sigma \in \mathit{Init} \wedge (\sigma \mapsto \sigma') \in \mathit{Trans}^* \wedge \sigma' \in \mathit{GMap}(g)$.

The required properties (1.1), (1.2), (1.4), and (1.5) are inherited from the standard definition of a state transition system. The two new elements of a GSTS introduce the abstract notion of system goals (as the type $\mathcal{G}$) and relate these goals (via the function *GMap*) with specific system states where these goals are considered to be achieved. Essentially, the function *GMap* assigns semantics to any goal from $\mathcal{G}$ by associating it with a non-empty set of states (a predicate) of Σ, as stated in (1.3) and (1.6). Finally, the last (1.7)

property of a GSTS requires that all the system goals must be achievable after system initialisation, i.e., they should be true either initially or after a number of system transitions defined by *Trans*.

Let us recall our running example – the multi-robotic cleaning system described in Section 2. The set $\mathcal{G}$ of this system contains two kinds of goals: "The zone j must be cleaned", for any $j \in 1..NumberOfZones$, and "The sector i of the zone j must be cleaned", for any $i \in 1..NumberOfSectors$ and $j \in 1..NumberOfZones$.

What would be a possible definition of *GMap* for this system? Let us consider the goal $g =$"The sector k of the zone l must be cleaned", for some fixed $k \in 1..NumberOfSectors$ and $l \in 1..NumberOfZones$. The mapping $GMap(g)$ then may be defined as, e.g.,

$$GMap(g) \;=\; \{\sigma \mid \; (SectorCleaned(\sigma))[l,k] \;=\; TRUE\},$$

where *SectorCleaned* is a state variable (binary array) storing information about the cleaned sectors. The type of such a variable is

$$\Sigma \to (1..NumberOfZones \times 1..NumberOfSectors \to BOOL).$$

An important dynamic property of a GSTS system is *stability* with respect to its goals, i.e., the system ability to retain the goals that have been already achieved.

Definition 2 Stable GSTS. *A GSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap)$ *is called* stable *if*

$$\textit{(2.1)}\;\; \forall \sigma, \sigma' : \Sigma, g : \mathcal{G}.\; (\sigma \mapsto \sigma') \in Trans \;\wedge\; \sigma \in GMap(g) \;\Rightarrow\; \sigma' \in GMap(g)$$

The system stability is a very desirable system property to have (especially for formal verification), however, it is also quite strong constraint on the system behaviour. For our example of the cleaning system, such an assumption would mean that all the cleaning goals are achievable in short duration, i.e., none of the cleaned sectors or zones gets "dirty" again before system termination.

So far, we considered system goals as members of a given set, which can be pursued and accomplished completely independently. Often, we can talk about some structure introducing inter-relationships between the system goals, e.g., distinguishing particular goals and their subgoals. This also implies that their semantic definitions (i.e., *GMap* functions) should be interrelated too. We can define these interrelationships by introducing two new structures – *G_graph* and *SGMap*.

Definition 3 Structured GSTS. *A GSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$ *is called* structured *if exist a relation on goals G_graph and a function SGMap, such that*

$$\begin{array}{ll} (3.1) & \mathit{G\_graph} : \mathcal{G} \leftrightarrow \mathcal{G}, \\ (3.2) & \mathit{SGMap} : \mathcal{G} \to \mathcal{P}(\Sigma), \\ (3.3) & \forall g : \mathcal{G}.\ (g \mapsto g) \notin \mathit{G\_graph}^+, \\ (3.4) & \forall g : \mathcal{G}.\ \mathit{GMap}(g) \subseteq \mathit{SGMap}(g), \\ (3.5) & \forall g, g' : \mathcal{G}.\ g' \in \mathit{Subgoals}(g) \Rightarrow \mathit{SGMap}(g) \cap \mathit{GMap}(g') \neq \varnothing, \end{array}$$

where $\mathit{Subgoals}(g) = \{g' : \mathcal{G} \mid (g \mapsto g') \in \mathit{G\_graph}\}$.

Moreover, the following is true

$$(3.6) \quad \forall \sigma : \Sigma, g : \mathcal{G}.\ \sigma \in \mathit{SGMap}(g) \wedge \sigma \notin \mathit{GMap}(g) \Rightarrow \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \sigma' \in \mathit{GMap}(g).$$

Mathematically, *G_graph* stands for an acyclic graph on goals. It describes relationships between different goals, e.g., how a particular goal can be split into its subgoals and so on. The property (3.3) states that a goal cannot be a subgoal of itself, i.e., the graph does not contains loops. The properties (3.4) - (3.6) give an alternative definition of the states associated with a particular goal, given as a function *SGMap*, in connection to the corresponding states of its subgoals. Specifically, (3.4) states that achieving g according to *GMap* implies that the goal g was achieved according to *SGMap* as well, while (3.5) requires that achieving any of subgoals must contribute to that of the parent goal. Intuitively, $\mathit{SGMap}(g)$ stands for the necessary precondition for achieving g, relating it with an arbitrary expression on the subgoals of g.

Finally, the last property (3.6) requires that, once the associated expression on subgoals *SGMap* is satisfied, the system always has an opportunity (a respective state transition) to complete the parent goal g, i.e., reach a state from *GMap(g)*. We deliberately allow such a "gap" in terms of an extra transition between achieving the main goal and that of its subgoals in the system dynamics because, as we will see later, achieving different goals can be responsibility of different system agents.

Let us go back to our running example. Since any zone is considered cleaned only after all its sectors are cleaned, the relation *G_graph* can be simply defined as

$$\{\text{"The zone } j \text{ must be cleaned"} \mapsto \text{"The sector } i \text{ of the zone } j \text{ must be cleaned"} \mid i \in 1..\mathit{NumberOfSectors} \wedge j \in 1..\mathit{NumberOfZones}\}.$$

Moreover, for the goal $g =$ "The zone l must be cleaned", for some fixed $l \in 1..NumberOfZones$, and its subgoals $Subgoals(g) = \{$"The sector k of the zone l must be cleaned" $\mid k \in 1..NumberOfSectors\}$, the mapping $SGMap(g)$ can be defined as

$$SGMap(g) \;=\; \{\sigma \mid \forall k \in 1..NumberOfSectors.\, (SectorCleaned(\sigma))[l,k] = TRUE\},$$

where *SectorCleaned* is a state variable described above.

Having the goal structure defined, we can easily distinguish the top goals of a structured GSTS. These are the goals that do not participate as subgoals for any other goal.

Definition 4 Top goals. *For a structured GSTS and its relation on goals G_graph, the system top goals are defined as*

$$(4.1) \qquad TopG \;=\; \mathbf{dom}(G\_graph) \backslash \mathbf{ran}(G\_graph).$$

Since *G_graph* is acyclic, the set *TopG* is always non-empty.

In particular, the top goals are especially important when we consider terminating GSTSs. The system termination can be easily formally defined by analysing their next state relation *Trans* as follows.

Definition 5 Terminating GSTS. *A GSTS is terminating if*

$$(5.1) \qquad \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\neq\; \varnothing.$$

When such a system terminates, we usually expect a certain property to be true on its top goals. A concrete choice depends of course on the considered system. Two obvious solutions are formalised below. The first one requires that the system in its terminating states achieves all its goals:

$$\forall \sigma : \Sigma, g : \mathcal{G}.\; \sigma \in \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\wedge\; g \in TopG \;\Rightarrow\; \sigma \in GMap(g).$$

Alternatively, we can require that at least one top goal is achieved:

$$\forall \sigma : \Sigma.\; \sigma \in \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\Rightarrow\; \exists g : \mathcal{G}.\; g \in TopG \wedge \sigma \in GMap(g).$$

For our running example, the top goals are obviously those that are related with zone cleaning. In the terminating states of the system, the cleaning system is required to achieve all its goals (a property of the first kind). Alternatively, the system could have a failsafe mechanism installed, which must be activated when completion of the cleaning (for whatever reason) becomes impossible. In this case, a property of the second kind can be enforced, requiring that either all the zones are cleaned or the failsafe procedure is successfully finished.

3.2 Introducing Agents

Now we extend the definition of a goal-oriented state transition system presented in the previous section by introducing agents that can carry out tasks leading to achieving the system goals.

Definition 6 Multi-agent goal-oriented state transition system (MAGSTS). *A MAGSTS system is a tuple* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *such that* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$ *is a GSTS,* $\mathcal{A}$ *is a set of all system agents, and* Active *is a function returning a subset of active agents in a particular system state, where*

$$\mathit{Active} : \Sigma \to \mathcal{P}(\mathcal{A}). \tag{6.1}$$

The definition introduces a type (set) $\mathcal{A}$ for all possible system agents and also associates a subset of active agents in the current system state via the function *Active*. Our interpretation of "active" agents is that only active agents can carry out the tasks in order to achieve the system goals. Inactive agents are either those are not currently present in the system or those that are failed and thus incapable to carry out any tasks.

If a multi-agent systems allows the agents to become active or inactive (e.g., failed) at any moment, we call such a system open. Formally, we define it as follows.

Definition 7 Open MAGSTS. *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *is open if the following properties hold:*

$$\begin{aligned} (7.1) \quad & \forall \sigma : \Sigma,\ a : \mathcal{A}.\ \sigma \in \mathbf{dom}(\mathit{Trans}) \ \wedge\ a \in \mathit{Active}(\sigma) \ \Rightarrow \\ & \exists \sigma'.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \mathit{Active}(\sigma') = \mathit{Active}(\sigma) \backslash \{a\} \end{aligned}$$

and

$$\begin{aligned} (7.2) \quad & \forall \sigma : \Sigma,\ a : \mathcal{A}.\ \sigma \in \mathbf{dom}(\mathit{Trans}) \ \wedge\ a \notin \mathit{Active}(\sigma) \ \Rightarrow \\ & \exists \sigma'.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \mathit{Active}(\sigma') = \mathit{Active}(\sigma) \cup \{a\}. \end{aligned}$$

In our running example, the set $\mathcal{A}$ include all the system base stations and robots, active as well as inactive ones. Both base stations and robots can fail at any moment, thus becoming inactive. If we require the described cleaning system open, this would mean that some recovery mechanism must be in place, allowing any failed base station or robot to be reintroduced into the system as an active agent.

Since the set $\mathcal{A}$ contains all possible system agents, some of them may have very different functionalities (abilities). In order to associate certain classes of agents with specific types of system goals they are able to accomplish, we first introduce classifications of system agents and goals and then define relationships between the introduced classes.

Definition 8 Typed MAGSTS. *A structured MAGSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap, \mathcal{A}, Active)$ *is typed if there exist the functions atype and gtype, such that*

$$\begin{aligned}
&(8.1) \quad atype : \mathcal{A} \to AType,\\
&(8.2) \quad gtype : \mathcal{G} \to GType,\\
&(8.3) \quad \forall at : AType.\ \exists a : \mathcal{A}.\ atype(a) = at,\\
&(8.4) \quad \forall gt : GType.\ \exists g : \mathcal{G}.\ gtype(g) = gt,\\
&(8.5) \quad \forall g_1, g_2 : \mathcal{G},\ gt : GType.\ gtype(g_1) = gt \ \wedge\ gtype(g_2) = gt \ \wedge\ g_1 \neq g_2 \ \Rightarrow\\
&\qquad\qquad GMap(g_1) \cap GMap(g_2) = \varnothing,
\end{aligned}$$

where AType and GType are abstract types containing all possible agent and goal types respectively.

In (8.1) and (8.2), the functions *atype* and *gtype* associate each agent and goal with their respective type. Both agent and goal types are nonempty in the sense that they must have at least one agent or goal associated with them (properties (8.3) and (8.4)). The last property (8.5) is introduced to ensure that distinct goals of the same goal type can be achieved independently, i.e., can be assigned to different agents to accomplish them in parallel. However, before giving such assignments to agents, we have to be sure that they are able to accomplish the assigned tasks. To formalise this, we introduce a special relation to represent interrelationships between different agent and goal types.

Definition 9 Relationship between agent and goal types *We say that agent and goal types are related if there exists a relation AG_Rel, such as*

$$\begin{aligned}
&(9.1) \quad AG_Rel : AType \leftrightarrow GType,\\
&(9.2) \quad \mathbf{dom}(AG_Rel) = AType, \text{ and}\\
&(9.3) \quad \mathbf{ran}(AG_Rel) = GType.
\end{aligned}$$

For convenience, the relation *AG_Rel* can be represented as a pair of functions A_goals, $A\_goals : AType \to \mathcal{P}(GType)$, and G_agents, $G\_agents : GType \to \mathcal{P}(AType)$, such that

$$\forall at : AType, gt : GType.\ gt \in A\_goals(at) \ \Leftrightarrow\ (at \mapsto gt) \in AG\_Rel$$

and

$$\forall gt : GType, at : AType.\ at \in G\_agents(gt) \ \Leftrightarrow\ (at \mapsto gt) \in AG\_Rel.$$

From this definition, we immediately get that

$$\forall at : AType, gt : GType.\ gt \in A\_goals(at) \ \Leftrightarrow\ at \in G\_agents(gt)$$

In our running example, *AType* can be easily defined as the set $\{BaseStations, Robots\}$, while *GType* is simply $\{CleaningZones, CleaningSectors\}$. The relation *AG_Rel* then interconnects the introduced agent and goal types as follows:

$$\{BaseStations \mapsto CleaningZones, Robots \mapsto CleaningSectors\}.$$

Knowing the interrelationships between the agent and goal types allows us to check in a straightforward way whether a concrete agent is able to accomplish a specific goal.

Definition 10 Agent ability *We say that an agent $a : \mathcal{A}$ is able to accomplish a goal $g : \mathcal{G}$ if*

$$(10.1) \qquad atype(a) \in G\_agents(gtype(g))$$

or, equivalently,

$$(10.2) \qquad gtype(g) \in A\_goals(atype(a)).$$

Often, the hierarchical structure between goals and subgoals, formalised by *G_graph*, is reflected on the goal types as well.

Definition 11 Hierarchy of goal types *We say a structured MAGSTS system supports a hierarchy of goal types if there is exists a relation GT_graph, such that*

$$(11.1) \qquad GT\_graph : GType \leftrightarrow GType,$$

$$(11.2) \qquad \forall gt \in GType.\ (gt \mapsto gt) \notin GT\_graph^{+},$$

$$(11.3) \qquad \forall g_1, g_2 : \mathcal{G}.\ (gtype(g_1) \mapsto gtype(g_2)) \in GT\_graph \ \Rightarrow$$
$$gtype(g_1) \neq gtype(g_2) \ \wedge\ (g_1 \mapsto g_2) \in G\_graph^{+}.$$

For our running example, *GT_graph* is simply a singleton set $\{CleaningZones \mapsto CleaningSectors\}$.

The hierarchical structures between goals and subgoals introduced above define the existing dependencies between the goals and thus imply the manner their achievement can be coordinated among the involved agents. Moreover, the formalised connection between the agent and goal types clarifies which agents can be given the tasks related with specific system goals.

3.3 Agent Subordination and Supervision

Having agent types and hierarchy of goal types defined makes it possible to introduce a subordination structure between agent types.

Definition 12 Subordinated MAGSTS. *A MAGSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap, \mathcal{A}, Active))$ *is called* subordinated *if it is typed, supports a hierarchy of goal types, and exists a relation on agent types A_Sub, such that*

$$\begin{aligned}
&(12.1) \quad A_Sub : AType \leftrightarrow AType,\\
&(12.2) \quad \mathbf{dom}(A_Sub) \cup \mathbf{ran}(A_Sub) = AType,\\
&(12.3) \quad \forall at \in AType.\ (at \mapsto at) \notin A_Sub^{+}.
\end{aligned}$$

Moreover, for each $at_1, at_2 : AType$*, such that* $(at_1 \mapsto at_2) \in A\_Sub$*, the following property must hold*

$$\begin{aligned}
(12.4) \quad &\exists gt_1, gt_2 : GType.\\
&(gt_1 \mapsto gt_2) \in GT_graph \ \wedge \ gt_1 \in A_goals(at_1) \wedge gt_2 \in A_goals(at_2).
\end{aligned}$$

According to the definition (properties (12.1)–(12.3)), *A_Sub* is acyclic graph covering all system agent types. The last property (12.4) states the required connection between two hierarchical structures: the goal type structure *GT_graph* and the agent subordination structure *A_Sub*. Namely, for each pair of subordinated agent types, exists (at least one) pair of the related goal types such that goals of the parent goal type can be handled by agents of the "master" agent type, while goals of the subgoal type can be handled by agents of the subordinate agent type.

It is obvious that, for our running example, the only possible way to define *A_Sub* is as a singleton set $\{BaseStations \mapsto Robots\}$. The base stations are responsible for zone cleaning, which can be decomposed into cleaning of the constituent sectors by robots. Since base stations are responsible for a higher level goal (i.e., are aware of a "bigger picture"), it is natural to appoint them as supervisors with respect to robots.

If a system is centralised one, even members of the top agent type may be needed to be supervised. In that case, the agent type hierarchy can be artificially extended with the top element *SystemType*, which has a single agent *System* as its member.

Similarly as for *AG_Rel*, the relation *A_Sub* can be represented as a pair of functions *AS_goals*, $AS\_goals : AType \to \mathcal{P}(GType)$, and *GS_agents*, $GS\_agents : GType \to \mathcal{P}(AType)$, such that

$$\begin{aligned}
&\forall at : AType, gt : GType.\ gt \in AS_goals(at) \ \Leftrightarrow\\
&\quad \exists gt' : GType.\ (gt' \mapsto gt) \in GT_graph \ \wedge \ gt' \in A_Goals(at)
\end{aligned}$$

and

$$\forall gt : GType, at : AType.\ at \in GS\_agents(gt) \ \Leftrightarrow \\ \exists gt' : GType.\ (gt' \mapsto gt) \in GT\_graph \ \wedge \ gt' \in A\_Goals(at).$$

From this definition, we immediately get that

$$\forall at : AType, gt : GType.\ gt \in AS\_goals(at) \ \Leftrightarrow \ at \in GS\_agents(gt)$$

The above definitions allow us to check in a straightforward way whether a concrete agent is able to supervise accomplishing a specific goal.

Definition 13 Agent supervision *We say that an agent $a : \mathcal{A}$ is able to supervise a goal $g : \mathcal{G}$ if*

$$(13.1) \qquad atype(a) \in GS\_agents(gtype(g))$$

or, equivalently,

$$(13.2) \qquad gtype(g) \in AS\_goals(atype(a)).$$

The notions about agents introduced so far (agent types, subordination, ability to accomplish or supervise a particular goal) define required static properties of a multi-agent goal-oriented system. The only exception is a function *Active*, which returns a set of active agents in a particular system state. Since agents can change their active/inactive status during system execution, the function expresses a dynamic system characteristic.

In subordinated MAGSTSs, a part of system agents supervise activities of other agents. Moreover, they can give concrete goal assignments to subordinate agents, which, in turn, should "report" to their supervisors once the assigned goal has been accomplished. The unreached system goals can be also dynamically partitioned among the supervisor agents.

This allows us to introduce a few additional dynamic system characteristics. Namely, in a specific dynamic system state, a particular agent can be "attached" to another agent, which serves as its supervisor. A specific, yet unreached goal can be put under responsibility of a particular supervisor agent. Finally, a specific goal can be "assigned" by a supervisor to one of its subordinate agents.

Let us now to define these dynamic notions formally.

Definition 14 Agent attachment *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *supports agent attachment if there is a dynamic attribute (function) Attached, such that*

$$\textit{(14.1)} \quad \mathit{Attached} : \Sigma \to \mathcal{P}(\mathcal{A} \times \mathcal{A}),$$

$$\begin{aligned}\textit{(14.2)} \quad & \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \Rightarrow \\ & a_1 \in \mathit{Active}(\sigma)\ \wedge\ a_2 \in \mathit{Active}(\sigma)\ \wedge\ \mathit{atype}(a_1) \mapsto \mathit{atype}(a_2) \in \mathit{A\_Sub}\ \wedge \\ & \neg(\exists a_3 : \mathcal{A}.\ a_3 \neq a_1\ \wedge\ (a_3 \mapsto a_2)\ \in\ \mathit{Attached}(\sigma)).\end{aligned}$$

Moreover, the following property is true

$$\begin{aligned}\textit{(14.3)} \quad & \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ a_1 \in \mathit{Active}(\sigma)\ \wedge\ a_2 \in \mathit{Active}(\sigma)\ \wedge \\ & \mathit{atype}(a_1) \mapsto \mathit{atype}(a_2) \in \mathit{A\_Sub}\ \wedge\ \neg(\exists a_1' : \mathcal{A}.\ (a_1' \mapsto a_2)\ \in\ \mathit{Attached}(\sigma)) \\ & \Rightarrow \\ & \qquad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma').\end{aligned}$$

Therefore, for any agents a_1, a_2 and system state σ, the expression $(a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)$ implies that (i) both agents are active in σ, (ii) the agent type of a_2 is subordinate to that of a_1, and (iii) the agent a_2 is not currently attached to any other supervisor agent (property (14.2)).

Moreover, a MAGST system supports agent attachment if, at any point where the conditions for agent attachment are satisfied, the system has an opportunity (but not an obligation) to do such an action (property (14.3)).

In our running example, any active and yet unattached robot can be attached to any active base station. It can also change its supervisor base station to a different active one.

Definition 15 Goal responsibility *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *supports goal responsibility if there is a dynamic attribute (function) Responsible, such that*

$$\textit{(15.1)} \quad \mathit{Responsible} : \Sigma \to \mathcal{P}(\mathcal{G} \times \mathcal{A}),$$

$$\begin{aligned}\textit{(15.2)} \quad & \forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ (g \mapsto a) \in \mathit{Responsible}(\sigma)\ \Rightarrow \\ & \quad a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a))\ \wedge \\ & \quad \neg(\exists a' : \mathcal{A}.\ a' \neq a\ \wedge\ (g \mapsto a')\ \in\ \mathit{Responsible}(\sigma)).\end{aligned}$$

Moreover, the following property is true

$$\begin{aligned}\textit{(15.3)} \quad & \forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a))\ \wedge \\ & \neg(\exists a' : \mathcal{A}.\ (g \mapsto a')\ \in\ \mathit{Responsible}(\sigma)) \\ & \Rightarrow \\ & \qquad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (g \mapsto a) \in \mathit{Responsible}(\sigma').\end{aligned}$$

Therefore, for any goal g, agent a and system state σ, the expression $(g \mapsto a) \in \mathit{Responsible}(\sigma)$ implies that (i) the agent a is active in the state σ, (ii) the agent type allows it to supervise g, and (iii) the goal g is not currently under responsibility of any other supervisor agent (property (15.2)).

Moreover, a MAGST system supports goal responsibility if, at any point where the conditions for an agent taking responsibility for some goal are satisfied, the system has an opportunity (but not an obligation) to do this action (property (15.3)).

In our running example, any active base station can take responsibility over a zone which is not yet responsibility of any other base station. Zone responsibility can also "migrate" from one base station to another as a part of the system reconfiguration.

Definition 16 Goal assignment *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *supports goal assignment if there is a dynamic attribute (function)* $\mathit{Assigned}$*, such that*

(16.1) $\quad \mathit{Assigned} : \Sigma \to \mathcal{P}(\mathcal{G} \times \mathcal{A} \times \mathcal{A}),$

(16.2)
$$\begin{aligned}
&\forall \sigma : \Sigma,\ g : \mathcal{G},\ a_1, a_2 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma)\ \Rightarrow \\
&(a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge\ (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\
&\mathit{gtype}(g) \in \mathit{A_goals}(\mathit{atype}(a_2))\ \wedge \\
&\neg(\exists g' : \mathcal{G},\ a' : \mathcal{A}.\ g' \neq g\ \wedge\ a'_1 \neq a_1\ \wedge\ (g' \mapsto a'_1 \mapsto a_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\neg(\exists a'_1, a'_2 : \mathcal{A}.\ a'_1 \neq a_1\ \wedge\ a'_2 \neq a_2\ \wedge\ (g \mapsto a'_1 \mapsto a'_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\sigma \notin \mathit{GMap}(g)\ \wedge\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ \sigma' \in \mathit{GMap}(g).
\end{aligned}$$

Moreover, the following property is true

(16.3)
$$\begin{aligned}
&\forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge \\
&\mathit{gtype}(g) \in \mathit{A_goals}(\mathit{atype}(a_2))\ \wedge\ (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\
&\neg(\exists g' : \mathcal{G},\ a' : \mathcal{A}.\ g' \neq g\ \wedge\ a'_1 \neq a_1\ \wedge\ (g' \mapsto a'_1 \mapsto a_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\neg(\exists a'_1, a'_2 : \mathcal{A}.\ (g \mapsto a'_1 \mapsto a'_2)\ \in\ \mathit{Assigned}(\sigma)) \\
&\Rightarrow \\
&\exists (\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma'),
\end{aligned}$$

Therefore, for any agents a_1, a_2, a goal g and system state σ, the expression $(g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma)$ implies that (i) a_2 is attached to a_1 in the state σ, (ii) a_1 is responsible for achieving the goal g in the state σ, (iii) a_2 is able to accomplish any goal of the type $\mathit{gtype}(g)$, (iv) the agent a_2 is not assigned to any other goal, (v) the goal g is not assigned to any other agent, (vi) the goal g is not yet completed, and (vii) once the goal g is

assigned, it can be completed at any moment (property (16.2)). The last two properties allow us to associate the goal reachability with goal assignment, and by transitivity, goal responsibility and agent attachment mechanisms.

Moreover, a MAGST system supports goal assignment if, at any point where the conditions for goal assignment are satisfied, the system has an opportunity (but not obligation) to do this action (property (16.3)).

In our running example, any attached and active robot without the current cleaning assignment (e.g., just after finishing the previous one) can be given a new cleaning assignment by its supervisor base station.

3.4 System Reconfiguration and Goal Reachability in the Presence of Agent Failures

Even though the above definitions require the existence of system transitions for the agents and goals that are "free", i.e., have not been attached or assigned, they implicitly cover two more kinds of system transitions:

1. Since all the definitions depend on the assumptions that the involved agents are active, change of the agent status to inactive (e.g., agent failure) during system transitions would mean automatic update of *Attached, Responsible* and *Assigned* by removing all those records that refer to the failed agents;

2. In situations when the involved agents remain active during the system transitions, the above definitions do not forbid changing the actual relationships between the agents and the goals. In other words, the agents can be reattached, goal responsibility can be redistributed, and goals can be reassigned among the active agents.

Let us explicitly define multi-agent systems that support the dynamic reconfiguration described in the latter observation. Specifically, these are the systems that allow redistributing (unassigned) goals to different responsible agents or reattaching (unassigned) agents to different supervisor agents. The multi-robotic cleaning system that we have used as the running example is an instance of such systems.

Definition 17 Reconfigurable agent system *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *is reconfigurable if it is structured, open, and supports agent attachment, goal responsibility, and goal assignment. Moreover, the following properties hold*

(17.1)
$$\begin{aligned}\forall \sigma : \Sigma,\ g : \mathcal{G},\ : a_1, a_2 : \mathcal{A}.\ & (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\ & \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a\_2))\ \wedge \\ & \neg(\exists a_3 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_3)) \in \mathit{Assigned}(\sigma))\ \Rightarrow \\ & \quad \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (g \mapsto a_2) \in \mathit{Responsible}(\sigma')\end{aligned}$$

and

$$(17.2) \qquad \forall \sigma : \Sigma,\ a_1, a_2, a_3 : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge \\ (\mathit{atype}(a_3) \mapsto \mathit{atype}(a_2)) \in \mathit{A\_Sub}\ \wedge \\ \neg(\exists g : \mathcal{G}.\ (g \mapsto a_1 \mapsto a_2)) \in \mathit{Assigned}(\sigma))\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_3 \mapsto a_2) \in \mathit{Attached}(\sigma')$$

In our first definition of a goal-oriented multi-agent system, we required that any system goal is reachable from some initial system state. For a reconfigurable MAGSTS system, we can formulate and prove a stronger property: *"Any goal that is not yet reached at any (non-final) system state is reachable."*

Theorem 1 Goal reachability in a reconfigurable agent system. *For a reconfigurable MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$*, the following property is true:*

$$\forall \sigma : \Sigma,\ g : \mathcal{G}.\ \sigma \in \mathbf{dom}(\mathit{Trans})\ \wedge\ \sigma \notin \mathit{GMap}(g)\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}^{+}\ \wedge \sigma' \in \mathit{GMap}(g).$$

Proof 1 *Let us consider an arbitrary state* $\sigma : \Sigma$*, such that* $\sigma \in \mathbf{dom}(\mathit{Trans})$*, and an arbitrary goal* $g : \mathcal{G}$*, such that* $\sigma \notin \mathit{GMap}(g)$*. Moreover, as the worst case scenario, let us assume that there is no single active agent able to supervise this goal nor single active agent able to accomplish it. Formally,*

$$\neg(\exists a : \mathcal{A}.\ a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a)))$$

and

$$\neg(\exists a : \mathcal{A}.\ a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{A\_goals}(\mathit{atype}(a))).$$

According to Definition 8 of a typed MAGSTS system, there should exists an agent able to supervise the goal g as well as a one able to accomplish it. Moreover, since our system is open, the second property of an open MAGSTS system (Definition 7) states a possibility to activate any agent at an arbitrary moment.

Let $a\_super : \mathcal{A}$*, such that*

$$a\_super \notin \mathit{Active}(\sigma) \qquad \textit{and} \qquad \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a\_super))$$

be an agent able to supervise the goal g. Moreover, let $\sigma_1 : \Sigma$*, such that* $(\sigma \mapsto \sigma_1) \in \mathit{Trans}$ *and* $\mathit{Active}(\sigma_1) = \mathit{Active}(\sigma) \cup \{a\_super\}$*, be a next state where*

this agent is activated. The existence of such state is required by Definition 7.

In a similar way, we activate an agent for accomplishing the goal g, a_worker, such that

$$a\_worker \notin Active(\sigma) \qquad and \qquad gtype(g) \in A\_goals(atype(a\_worker))$$

in a next state, σ_2, such that

$$(\sigma_1 \mapsto \sigma_2) \in Trans \qquad and \qquad Active(\sigma_2) = Active(\sigma_1) \cup \{a\_worker\}.$$

Relying on the definitions of agent attachment, goal responsibility and goal assignment (Definitions 14, 15 and 16), we can construct a chain of further states σ_3, σ_4, σ_5, where a_super and $a\_workers$ become attached, a_super takes responsibility for the goal g, and a_worker gets assigned the goal g respectively. There could be also as many as necessary intermediate state transitions where the statuses of a_super and a_worker are unaffected.

Finally, the definition of agent assignment (more specifically, the last consequent of property (16.2) of Definition 16) also connects this notion with goal reachability. Namely, for any state where a particular agent is assigned a specific goal, there exists a possible subsequent state where this goal is reached. Since the state σ_5 satisfies these criteria, we can claim that there exists a state, σ', such that

$$(\sigma_5 \mapsto \sigma') \in Trans \ \wedge \ \sigma' \in GMap(g).$$

By transitivity, we proved that

$$(\sigma \mapsto \sigma') \in Trans^{+} \ \wedge \sigma' \in GMap(g),$$

which is exactly what the theorem states.

In a similar manner, we can construct proofs for less adverse cases involving agent failures (i.e., becoming inactive), which in turn lead to specific agents becoming unattached, unassigned or specific goals losing the supervisors responsible for their completion.

To complete the proof, we have also consider the system states when the system has all the active agents needed to achieve a particular unreached goal, however the specific agent attachment and goal responsibility distribution has to be adjusted first. In other words, the system has to be reconfigured before proceeding.

Let us again consider an arbitrary state $\sigma : \Sigma$, such that $\sigma \in \mathbf{dom}(Trans)$, and an arbitrary goal $g : \mathcal{G}$, such that

$$\sigma \notin GMap(g) \qquad and \qquad \neg(\exists a_1, a_2 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_2) \in Assigned(\sigma)).$$

The completion of g in the state σ is responsibility of the agent a_super, i.e., $(g \mapsto a\_super) \in Responsible(\sigma)$.

Moreover, there exist the agents a_other and a_worker such that

$$(a\_super \mapsto a\_worker) \in Attached(\sigma) \ \ and \ \ gtype(g) \in A\_goals(atype(a\_worker)).$$

In other words, a_worker is attached to a_other and is able to accomplish the goal g. This also implies that a_other is able to supervise the goal g.

Relying on the definition of a reconfigurable multi-agent system (Definition 17), we can state that exists a next state σ'', where the agent a_worker is now attached to the new supervisor a_super, i.e.,

$$(g \mapsto a\_super) \in Responsible(\sigma'') \ \ and \ \ (a\_super \mapsto a\_worker) \in Attached(\sigma'').$$

Alternatively, we can state that exists a next state σ'', where the responsibility of completing g is now moved to the new supervisor a_other, i.e.,

$$(g \mapsto a\_other) \in Responsible(\sigma'') \ \ and \ \ (a\_other \mapsto a\_worker) \in Attached(\sigma'').$$

In both cases, we can proceed by assigning the goal g to the agent a_worker and completing the goal as described above. In other words, we can show that there is the state $\sigma' : \Sigma$ such that

$$(\sigma'' \mapsto \sigma') \in Trans^+ \ \wedge \ \sigma' \in GMap(g).$$

By transitivity, we get that

$$(\sigma \mapsto \sigma') \in Trans^+ \ \wedge \sigma' \in GMap(g),$$

which is again exactly what we needed to prove.

□

The theorem requires for a multi-agent system to be open, which is not always the case in practice. In general, even without the openness assumption, we can still demonstrate goal reachability provided there always exists at least one agent which is able to accomplish this goal as well as one agent which is able to supervise it. The incorporated reconfigurability mechanisms will be then still sufficient to enable completion of the goal. Alternatively, we can quantitatively assess (based on the given agent failure and service rates) goal reachability using probabilistic model checking. In our previous work [32], we have employed such techniques for quantitative assessment of goal-oriented multi-agent systems using the PRISM probabilistic model checker. To enable probabilistic analysis of system models in PRISM, we have relied on our continuous-time probabilistic extension of the Event-B framework [33].

4 Formal Development of a Goal-Oriented MAS in Event-B

In the previous section we presented a general theory for reasoning about goal-oriented multi-agent state transition systems with incorporated reconfiguration mechanisms. There are many formalisms that support modelling and verification of state transition systems. In this section we will briefly overview one of them – Event-B – and present a number guidelines demonstrating how the notions defined above can be easily transferred and incorporated in Event-B. In a sense, by doing this we show a possible instantiation of our general theory in a concrete formalism. In turn, this should facilitate formal development of a goal-oriented MAS in Event-B.

4.1 Event-B: Background

Event-B is a state-based framework that promotes the correct-by-construction approach to system development and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [1]. An Abstract State Machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important properties of the systems are defined in the model *invariants*. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

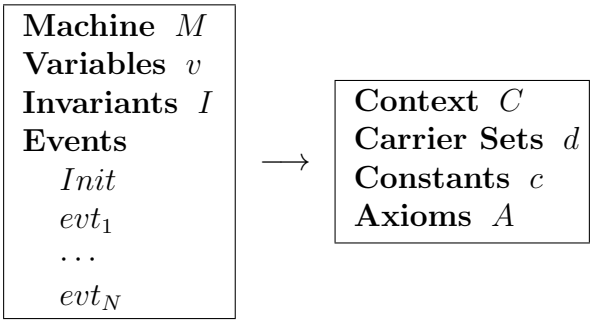


Figure 2: Event-B machine and context

A general form for Event-B models is given in Fig. 2. The machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialised in the $Init$ event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties (e.g., safety invariants) that should be preserved during system execution.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. In Event-B, an assignment represents a corresponding next-state relation R_e. The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

If an event does not have local variables, it can be described simply as:

$$e \mathrel{\widehat{=}} \textbf{when } G_e \textbf{ then } R_e \textbf{ end}.$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables.

Often a refinement step introduces new events and variables into the abstract specification. The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit skip. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events converge. To prove it, we need to define a variant an expression over a finite subset of natural numbers and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. Then the event obtains the status *anticipated*. This obliges the designer to prove at some later refinement step, that the event indeed converges.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging a number of verification conditions – proof obligations. The Rodin platform [27] provides an automated support for formal modelling and verification in Event-B. In particular, it automatically generates the required proof obligations and attempts to discharge them. The remaining unproven conditions can be dealt with by using the provided interactive provers.

Context C0
Sets $Goals$, $Agents$
Constants G_graph, $GraphClosure$, $Subgoals$
Axioms
axm1: $Goals \neq \varnothing$
axm2: $Agents \neq \varnothing$
axm3: $G\_graph \in Goals \leftrightarrow Goals$
axm4: $GraphClosure \in Goals \leftrightarrow Goals$
axm5: $G\_graph \subseteq GraphClosure$
axm6: $\forall g1, g2 \cdot (g1 \mapsto g2) \in G\_graph \Leftrightarrow$
$(g1 \mapsto g2) \in G\_graph \vee (\exists g3 \cdot (g1 \mapsto g3) \in G\_graph \wedge (g3 \mapsto g2) \in GraphClosure)$
axm7: $\forall g \cdot g \in Goals \Rightarrow (g \mapsto g) \notin GraphClosure$
axm8: $\forall g \cdot g \in Goals \Rightarrow Subgoals(g) = \{g1 \mid (g \mapsto g1) \in G\_graph\}$
...

Figure 3: Context C0

4.2 A Goal-Oriented MAS in Event-B

To formally develop a multi-agent system based on the theory presented in the previous section, we have to translate or represent the introduced notions and definitions in terms of the corresponding Event-B elements. Below we present our guidelines for such a translation.

Event-B separates the static and dynamic parts of a model, putting them into distinct yet dependent components called a context and a machine. Similarly, for our theory, we must first distinguish static and dynamic concepts and then do, if necessary, a further classification of them to translate the resulting cases into specific Event-B elements. The obvious criterion for such a separation is the direct dependence of the concept in question on the type Σ denoting the system state space. To be precise, if its type depends on Σ (see Table 1 for particular cases of such dependence), we consider a concept is a dynamic one and it must be represented as one of the elements (e.g., state variables or events) in the model machine component(s). Otherwise, it is considered static and becomes one of the elements of a model context.

Representation of static concepts of goal-oriented MAS in Event-B. Using the above criterion, the static notions of our theory include the types for all possible goals and agents ($\mathcal{G}$ and $\mathcal{A}$) and their types (*GType* and *AType*) as well as different structures defining various classifications and interdependencies between elements of these types. The latter include *G_graph*, *GT_graph*, *atype*, *gtype*, *A_goals*, *G_agents*, *AG_Rel*, *A_Sub*, *AS_goals*, *GS_agents*, and so on.

We introduce static notions as sets and constants of a model context and define their properties as a number of context axioms. For instance, the following excerpt (Fig.3) defines the sets *Goals* and *Agents* (i.e., $\mathcal{G}$ and $\mathcal{A}$) as well as the constants *G_graph*, *GraphClosure* (i.e., G_graph^+), and *Subgoals*.

Note that, since both our theory and Event-B are based set theory and

Context C1 **extends** C0
Sets $GType$, $AType$
Constants $atype$, $gtype$, GT_graph, $ROBOT$, $B\_STATION$, $ZONE\_CLEANING$, AG_Rel
$SECTOR\_CLEANING$, $Robots$, $BStations$, $ZCleaning$, $SCleaning$
Axioms
axm1: $ZCleaning \subseteq Goals \ \wedge SCleaning \subseteq Goals$
axm2: $Robots \subseteq Goals \ \wedge BStations \subseteq Goals$
axm3: $(Goals = ZCleaning \cup SCleaning) \ \wedge \ (ZCleaning \cap SCleaning = \varnothing)$
axm4: $(Agents = Robots \cup BStations) \ \wedge \ (Robots \cap BStations = \varnothing)$
axm5: $GType = \{ZONE\_CLEANING, SECTOR\_CLEANING\}$
axm6: $AType = \{ROBOT, B\_STATION\}$
axm7: $gtype \in Goals \rightarrow GType$
axm8: $atype \in Agents \rightarrow AType$
axm9: $gtype[Robots] = \{ROBOT\}$
axm10: $gtype[BStations] = \{B\_STATION\}$
axm11: $atype[ZCleaning] = \{ZONE\_CLEANING\}$
axm12: $atype[SCleaning] = \{SECTOR\_CLEANING\}$
axm13: $GT\_graph \in GType \leftrightarrow GType$
axm14: $GT\_graph = \{ZONE\_CLEANING \mapsto SECTOR\_CLEANING\}$
axm15: $AG\_Rel \in AType \leftrightarrow GType$
axm16: $AG\_Rel = \{B\_STATION \mapsto ZONE\_CLEANING, ROBOT \mapsto SECTOR\_CLEANING\}$
...

Figure 4: Context C1

predicate calculus, the considered definitions are translated in a rather straightforward way. Such a translation gives a generic context that may be used for modelling of a class of suitable systems or, alternatively, used in very abstract models which are later refined by constraining (instantiating) the defined structures for concrete cases.

In the following excerpt of a refined context (Fig.4), we constrain the abstract definitions of *Goals* and *Agents* to those of our running example (the multi-robotic cleaning system described in Section 2), as defined in the axioms 1–4. We also give concrete definitions for the introduced types *GType* and *AType* (axioms 5–6), the functions *gtype* and *atype* (axioms 7–12), and the relation structures *GT_graph* and *AG_Rel* (axioms 13–16). It can be easily demonstrated that these axioms are proper instantiations of their general definitions given in Definition 8 (*atype, gtype*), Definition 9 (*AG_Rel*) and Definition 11 (*GT_graph*).

Representation of dynamic concepts of goal-oriented MAS in Event-B. In general, the system dynamics (formalised as state transitions on state space Σ constrained by the relation $Trans$) is represented as machine events in Event-B. However, various introduced concepts that affect this dynamics (e.g., connecting particular state transitions with goal and agent structures, supervision and reconfiguration mechanisms, the properties to be preserved, etc.) can be represented as different elements of an Event-B machine, such as model variables, invariants, predicate expressions, or specific events. Table 1 gives a summary of such possible representations.

For instance, a number of dynamic system attributes (such as *Active*,

Table 1: Translation guidelines

Theory definition	Event-B counterpart
$Trans : \Sigma \leftrightarrow \Sigma$	initialisation and events of a machine
functions of the form $\Sigma \to T$	machine variables of the type T
$GMap(g)$, $SGMap(g) : \mathcal{G} \to \mathcal{P}(\Sigma)$	a predicate over machine variables
a property of the form $\forall \sigma : \Sigma.\ P(\sigma)$	a machine invariant
a property of the form $\forall \sigma : \Sigma.\ P(\sigma) \Rightarrow \exists \sigma'.\ \sigma \mapsto \sigma' \in Trans \wedge R(\sigma, \sigma')$	a specific machine event

Machine M1
Sees C1
Variables $Active, Attached, ...$
Invariants
`inv1:` $Active \in \mathbb{P}(Agents)$
`inv2:` $Attached \in \mathbb{P}(Agents \times Agents)$
`inv3:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \ \Rightarrow\ a1 \in Active \wedge a2 \in Active$
`inv4:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \ \Rightarrow\ atype(a1) \mapsto atype(a2) \in A\_Sub$
`inv5:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \Rightarrow \neg(\exists a3.\ a3 \neq a1 \wedge (a3 \mapsto a2) \in Attached)$
...
Events
...
end

Figure 5: Model variables and invariants

Attached, *Responsible*, *Assigned*, etc.) are formalised as functions of the form $\Sigma \to T$. They can be naturally represented as model variables of the type T. In their definitions, these attributes are usually associated with some defining properties that are supposed to be preserved in each reachable state. These properties then become invariants of the resulting Event-B model.

As an example, let us consider the definition of agent attachment (Definition 14). It introduces a dynamic attribute (function) $Attached : \Sigma \to \mathcal{P}(\mathcal{A} \times \mathcal{A})$ with the following property (14.2)

$$\begin{array}{l} \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ (a_1 \mapsto a_2) \in Attached(\sigma)\ \Rightarrow \\ \quad a_1 \in Active(\sigma)\ \wedge\ a_2 \in Active(\sigma)\ \wedge\ atype(a_1) \mapsto atype(a_1) \in A\_Sub\ \wedge \\ \quad \neg(\exists a_3 : \mathcal{A}.\ a_3 \neq a_1\ \wedge\ (a_3 \mapsto a_2)\ \in\ Attached(\sigma)). \end{array}$$

In its turn, *Attached* depends on another dynamic attribute (state variable) *Active* defined as $Active : \Sigma \to \mathcal{P}(\mathcal{A})$. The following excerpt from an Event-B machine (Fig.5) demonstrates how both *Active* and *Attached* can be represented.

Another kind of dynamic properties is often expressed in the form

$$\forall \sigma : \Sigma.\ P(\sigma) \Rightarrow \exists \sigma'.\ \sigma \mapsto \sigma' \in Trans \wedge R(\sigma, \sigma').$$

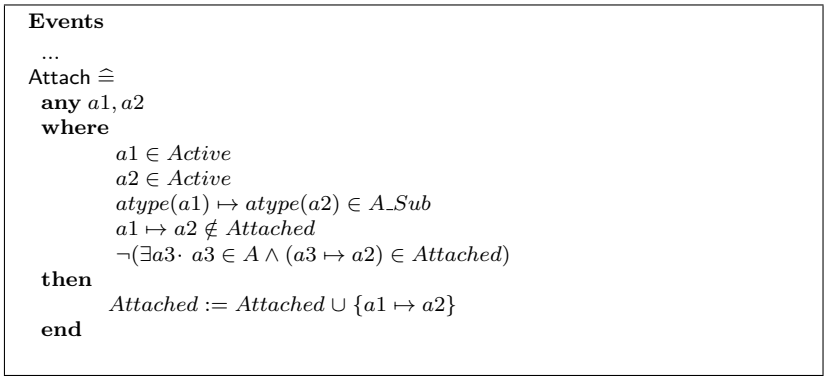


Figure 6: Model events

Essentially, such properties require existence a particular kind of state transitions in the system. Since state transitions are represented as model events in Event-B, this is an indication that a specific model event should be constructed, thus implementing the given property.

Let us go back to the definition of agent attachment (Definition 14). The last definition property (14.3) requires that

$$\begin{array}{l} \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ a_1 \in \mathit{Active}(\sigma)\ \wedge\ a_2 \in \mathit{Active}(\sigma)\ \wedge \\ \qquad \mathit{atype}(a_1) \mapsto \mathit{atype}(a_2) \in \mathit{A\_Sub}\ \wedge\ \neg(\exists a_1' : \mathcal{A}.\ (a_1' \mapsto a_2)\ \in\ \mathit{Attached}(\sigma)) \\ \Rightarrow \\ \qquad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma'). \end{array}$$

As a result of event construction, the left hand side of implication then becomes the event guard, while the right hand side defines the required action of the event (see Fig.6).

Finally, in our formalisation the functions $GMap : \mathcal{G} \to \mathcal{P}(\Sigma)$ and $SGMap : \mathcal{G} \to \mathcal{P}(\Sigma)$ relate goals with specific states where goals (or some expressions on their subgoals) are considered as reached. In Event-B, we can model these functions as particular predicates on state variables storing such information about reached goals. Often, such information is partitioned (stored on distinct variables) according to the involved goal type.

Let us consider again our example of the system with cleaning robots and coordinating base stations. We have two types of goals – zone cleaning and sector cleaning – which are responsibilities of base stations and robots respectively. The information about reached goals can be stored in two distinct boolean array variables: *zone_is_cleaned* for the goals in $CleaningZones$, and *sector_is_cleaned* for the goals in $CleaningSectors$. Then $GMap(g)$ can be represented as the predicate $\mathit{zone\_is\_cleaned}(g) = TRUE$, if $g \in CleaningZones$, and $\mathit{sector\_is\_cleaned}(g) = TRUE$ otherwise.

Recall that $SGMap(g)$ serves as the precondition for reaching the goal g, while the goal completion of g may not be officially recorded yet. In our

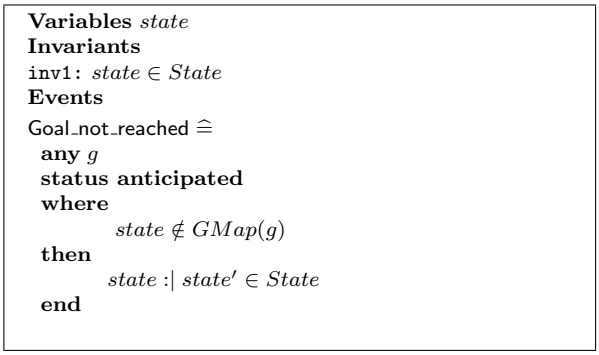


Figure 7: Anticipated goal reachability

example system, $SGMap(g)$ for $g \in CleaningSectors$ may be represented, e.g., as

$$CoverageSensor(r \mapsto s) = TRUE,$$

for some robot r and sector s, indicating that the whole sector area has been covered by the cleaning robot r, which may not yet reacted on that.

Goal reachability . In Definition 1, we define a goal-oriented multi-agent system as such that has ability to reach any of its goal from its initial states:

$$\forall g : \mathcal{G}.\ \exists \sigma, \sigma' : \Sigma.\ \sigma \in Init \ \wedge\ (\sigma \mapsto \sigma') \in Trans^* \ \wedge\ \sigma' \in GMap(g).$$

How can we enforce this property in Event-B? One possibility is to start with a very abstract system with a single event (see Fig.7).

Here the single state variable *state* is completely non-deterministically updated in the event *Goal_not_reached*. The anticipated status of the event indicates that we promise to prove convergence of this event, thus showing reachability of any system goal. The actual proof of such convergence is postponed until some later refined model, which has enough implementation details prove overall convergence based on a formulated variant expression.

Alternatively, we can rely on ProB, a model checker for Event-B, and verify goal reachability by formulating and checking the corresponding temporal logic property for the considered system model.

5 Related Work and Conclusions

Related Work. The field of design of multi-agent systems has considerable evolved over the last decade. Surveying the literature on MAS reveals a significant amount of research devoted to different agent organisation concepts, agent specification languages and platforms, modelling and verification agent behaviour, etc. The resulting approaches vary significantly in terms

of the covered topics, such as agent interoperability, communication, roles, goals and beliefs. Below we outline only a few works most relevant to our research.

The Tropos methodology [6] supports analysis and design in the development of agent-based software systems. UML diagrams are used to represent the system goals, agents, their capabilities and interdependencies, as well as system properties and agent interactions. An extension of this work [21] also supports modelling of agent errors and recovery activities.

Another proposed methodology - Multi-Agent System Engineering (MaSe) [8] – guides the designer through the software lifecycle of a multi-agent system. It allows graphically represent the system goals, the associated use cases and agent roles. Finite state automata are used to express communications between agent classes. The accompanied tool, the Agent Tool, supports the agent system development following the MaSe methodology. An extension of this work, Organization-based MaSE (O-MaSe) [9], provides a mechanism for defining agent interactions with the environment via external actors as well as defining the interaction protocols between the system and the actors. O-MaSE makes use of UML class diagrams and does not support formal notation.

Formal modelling of agent systems has been undertaken by [31, 30, 28, 29]. The authors have proposed an extension of the UNITY framework to explicitly define such concepts as mobility and context-awareness. The mobile UNITY [31] extension proposes the notation to express mobile computations and a logic for reasoning about components temporal properties. It also supports formal reasoning about mobile components and their behaviour. On the other hand, the Context UNITY extension [29] formalises context-aware computing, with the proposed notation to represent the system context. The sensed aspects of the environment are used by the system to adjust its behaviour. In our formalisation we have pursued a different goal – we aimed at formally guaranteeing that the specified agent behaviour with the incorporated reconfiguration mechanisms facilitates achieving the defined system goals.

Formal modelling of fault tolerant MAS in Event-B has been also undertaken by Ball and Butler [3]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach we consider fault tolerance as a part of ensuring resilience of MAS. Moreover, we have formalised a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability.

The use of model checking techniques for reasoning about MAS properties has been actively researched as well (see, e.g., [4, 5, 17, 11, 18]). In particular, [5] presents a framework for verification of agent programs against BDI (belief-desire-intention) agent specifications. In the proposed approach,

an agent system is first programmed using the logic-based agent oriented programming language AgentSpeak(F). Then the AgentSpeak(F) programs are translated into Promela – the specification language of the SPIN model checker – to verify the resulting system. Ferrari et al. [10] describe a verification of π-calculus based process algebra for mobile agents, while [18] presents modelling of fault-tolerant agents by stochastic Petri nets. The paper [17] describes the symbolic model checker MCMAS, specifically tailored for verification of MAS. The MCMAS tool takes as inputs models describing both agents and working environment of a multi-agent system and applies the epistemic logic to analyse it. However, model checking approaches typically suffer from the state space explosion problem, which is especially acute for large systems. As demonstrated by the proposed guidelines, our formalisation can be easily represented in the Event-B formalism. Since Event-B is based on theorem proving, this would help to avoid the mentioned problem.

The foundational work on *goal-oriented development* has been done by van Lamsweerde [7, 34, 36]. The proposed KAOS framework [7] provides a goal-oriented approach for requirements modelling, specification, and analysis, to address both functional and non-functional system requirements. Based on the KAOS framework, Lamsweerde [35] has proposed a method for deriving the software architecture from its requirements. Specifically, according to the method, the software specification is developed from the requirements which is then used to build the architectural design. The design is based with consecutive refinements, which take into account constraints and non-functional goals. The KAOS approach is supported by the GRAIL tool [7].

Over the last decade the goal-oriented approach has also received several extensions that allow the designers to link it with formal modelling [14, 25]. In particular, the work [14] presents the technique of translating KAOS operational models into event-based tabular specifications that can be then analysed by SCR* toolset [12]. The technique consists of a number of transformation steps each of which solves semantic, structural or syntactic dereferences between the KAOS and SCR (Software Cost Reduction) languages.

Significant amount of research has been devoted for translating formal specifications of software operations built according to the KAOS goal-oriented method into event-based transition systems. For example, the work [16] presents an approach to use the formal analysis capabilities of LTSA (Labelled Transition System Analyser) to analyse and animate KAOS operational models. The mapping allows designers to translate goal-oriented operational requirements into a black-box event-based model of the software behaviour expressed in a formalism appropriate to reason about behaviours at the architectural level.

One of the first attempt to bridge KAOS operations with B specifica-

tion was presented in [26]. More recently, the study to formalise KAOS in Event-B was attempted in [2]. The paper proposes a constructive approach that allows to link high-level system requirements expressed as linear temporal logic formulae to the corresponding Event-B elements. The notion of a triggered event is used to translate time operators that are used in KAOS models. Similar, Matoussi et al. [19, 20] describe works on coupling requirements engineering methods with formal methods. In contrast, in our work we have relied on goals to facilitate structuring of the system behaviour, while connecting them with agent collaboration and system reconfiguration mechanisms.

In our previous work on goal reachability and agent collaboration, we have investigated a colony of ants [13]. We have formalised the behaviour of cooperative ants in Event-B and verified by proofs that the desired system-level properties become achievable via agent collaboration. The proposed approach allows the designers to rigorously define constraints on the environment and the ant behaviour at different abstraction levels and systematically explore the relationships between system-level goals, environment and autonomous ants.

Conclusions. The main paper contribution is the proposed theoretical study of resilient goal-oriented multi-agent systems. The formalisation gradually defines the main notions of such systems, together with their intricate relationships between different agent and goal structures as well as the incorporated dynamic reconfiguration mechanisms. The latter allow the system to become more resilient with respect to the system goals and also more collaborative with respect to the involved system agents. The final theorem is proved to formally demonstrate that all the introduced notions and mechanisms are sufficient to ensure goal reachability in such a system. The presented work is based on our experience in formal modelling and verification of resilient goal-oriented multi-agent systems, see, e.g., [22, 23, 32, 13].

There is a number of features and properties of such systems that were left out from this formalisation. For instance, it would be interesting to more deeply investigate how the information about goal reachability is stored (distinguishing the local and global knowledge) and later propagated from agents to their supervisors and beyond. This issue is also directly related with the representation of different levels of perception that some goal is now completed, how this perception is propagated through the agent and goal hierarchies, the order of this propagation and delays related with it. In turn, such knowledge can be used to make the system reconfiguration mechanisms more efficient by, e.g, avoiding in some cases to redo an already accomplished goal after the supervisor agent responsible for this goal has failed. The listed topics constitute the basis for our future work in this research area.

References

[1] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[2] Benjamin Aziz, Alvaro Arenas, Juan Bicarregui, Christophe Ponsard, and Philippe Massonet. From goal-oriented requirements to event-b specifications. In *First NASA Formal Methods Symposium - NFM 2009*, pages 96–105, 2009.

[3] Elisabeth Ball and Michael Butler. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In *Methods, Models and Tools for Fault Tolerance*, pages 104–129. Springer, 2009.

[4] Rafael Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model Checking AgentSpeak. In *AAMAS 2003*, pages 409–416. ACM Press, 2003.

[5] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[6] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[7] Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 612–613. ACM, 1997.

[8] Scott A. DeLoach. The mase methodology. 11:107–125, 2004.

[9] Scott A. DeLoach and Juan C. García-Ojeda. O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

[10] Gian-Luigi Ferrari, Stefania Gnesi, Ugo Montanari, and Marco Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, October 2003.

[11] Jianye Hao, Songzheng Song, Yang Liu, Jun Sun, Lin Gui, Jin Song Dong, and Ho-fung Leung. Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In *PRIMA 2012*, volume 7455 of *LNCS*, pages 16–30. Springer, 2012.

[12] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. Scr*: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification, 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 526–531. Springer, 1998.

[13] Linas Laibinis, Elena Troubitsyna, Zeineb Graja, Frédéric Migeon, and Ahmed Hadj Kacem. Formal Modelling and Verification of Cooperative Ant Behaviour in Event-B. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014*, volume 8702 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2014.

[14] Renaud De Landtsheer, Emmanuel Letier, and Axel van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. *Requir. Eng.*, 9(2):104–120, 2004.

[15] J.-C. Laprie. From Dependability to Resilience. In *DSN 2008, Dependable systems and Networks*. IEEE Computer Society, 2008.

[16] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206, 2008.

[17] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *CAV 2009*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.

[18] Michael R. Lyu, Xinyu Chen, and Tsz Yeung Wong. Design and evaluation of a fault-tolerant mobile-agent system. *IEEE Intelligent Systems*, 19(5):32–38, 2004.

[19] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. A first attempt to express KAOS refinement patterns with event B. In *Abstract State Machines, B and Z, First International Conference, ABZ 2008*, page 338, 2008.

[20] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. A goal-based approach to guide the design of an abstract event-b specification. In *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, pages 139–148, 2011.

[21] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 9–16. ACM, 2008.

[22] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. A Case Study in Formal Development of a Fault Tolerant Multi-robotic System. In *Software Engineering for Resilient Systems - 4th International Workshop, SERENE 2012, Pisa, Italy, September 27-28, 2012. Proceedings*, volume 7527 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2012.

[23] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal Development of Critical Multi-agent Systems: A Refinement Approach. In *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 156–161. IEEE, 2012.

[24] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal Goal-Oriented Development of Resilient MAS in Event-B. In *Reliable Software Technologies - Ada-Europe 2012 - 17th Ada-Europe International Conference on Reliable Software Technologies, Stockholm, Sweden, June 11-15, 2012. Proceedings*, volume 7308 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2012.

[25] Christophe Ponsard, Gautier Dallons, and Massone Philippe. From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In *ERCIM News (75)*, pages 22–23, 2008.

[26] Christophe Ponsard and Emmanuel Dieul. From requirements models to formal specifications in B. In *Proceedings of the CAISE*06 Workshop on Regulations Modelling and their Validation and Verification ReMo2V '06*, 2006.

[27] Rodin. Event-B Platform. online at http://www.event-b.org/.

[28] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A formal treatment of context-awareness. In *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 12–36. Springer, 2004.

[29] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. Modeling adaptive behaviors in context UNITY. *Theor. Comput. Sci.*, 376(3):185–204, 2007.

[30] Gruia-Catalin Roman and Peter J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20(1):47–68, 2002.

[31] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. *ACM Trans. Softw. Eng. Methodol.*, 6(3):250–282, 1997.

[32] Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal development and quantitative assessment of a resilient multi-robotic system. In *Software Engineering for Resilient Systems, 5th International Workshop, SERENE 2013, Kiev, Ukraine, October 3-4, 2013. Proceedings*, volume 8166 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2013.

[33] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Integrating stochastic reasoning into Event-B development. *Formal Aspects of Computing*, 27(1):53–77, 2015.

[34] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering*, pages 249–263, 2001.

[35] Axel van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2003.

[36] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Paper IX

Formal Modelling of Resilient Data Storage in Cloud

Inna Pereverzeva, Elena Troubitsyna, Linas Laibinis, Marcus Holmberg and Mikko Pöri

Originally published in: Lindsay Groves and Jing Sun (Eds.), *Proceedings of 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, LNCS 8144, 363–379, Springer-Verlag Berlin Heidelberg, 2013.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-41202-8_24

Formal Modelling of Resilient Data Storage in Cloud

Inna Pereverzeva[1,2], Linas Laibinis[1], Elena Troubitsyna[1],
Markus Holmberg[3], and Mikko Pöri[3]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] F-Secure, Helsinki, Finland
{inna.pereverzeva, linas.laibinis, elena.troubitsyna}@abo.fi
{ext-markus.holmberg, mikko.pori}@f-secure.com

Abstract. Reliable and highly performant handling of large data stores constitutes one of the major challenges of cloud computing. In this paper, we propose a formalisation of a cloud solution implemented by F-Secure – a provider of secure data storage services. The solution is based on massive replication and the write-ahead logging mechanism. To achieve high performance, the company has abandoned a transactional model. We formally derive a model of the proposed architectural solution and verify data integrity and consistency properties under possible failure scenarios. The proposed approach allows the designers to formally define and verify essential characteristics of architectures for handling large data stores.

Keywords: Formal modelling, Event-B, refinement, replication, data integrity, large data stores

1 Introduction

Rapid development of digital technology puts a high demand on reliable handling and storage of large volumes of data. It is forecasted that worldwide consumer digital storage needs will grow from 329 exabytes in 2011 to 4.1 zettabytes in 2016 [3]. Often algorithms for data storage in cloud reuse the ones that have been proposed for databases. The transactional model adopted in databases guarantees ACID properties – Atomicity, Consistency, Isolation and Durability, and as such delivers high resilience guarantees. However, in a pursue of high performance, cloud data storages rarely rely on the transactional model and hence deliver weaker guarantees regarding data integrity. In this paper, we undertake a formal study of data integrity and consistency properties that can be guaranteed by several different architectures of cloud data stores.

Our work is motivated by a cloud solution developed by F-Secure – a provider of secure data storage services. To achieve a high degree of fault tolerance, the company has combined write-ahead logging (WAL) [7, 10] – a widely used mechanism for database error recovery – and massive data replication. As such, this

combination gives very high resilience guarantees (usually in the form of eventual consistency). However, these guarantees are different in non-transactional settings typical for cloud. Moreover, data integrity and consistency properties vary in the synchronous, semi-synchronous and asynchronous architectures used for data replication. Therefore, it is useful to rigorously define and compare the properties that can be ensured by different solutions.

In this paper, we use the Event-B method and the associated Rodin platform to formally model write-ahead logging in replicated data stores. Event-B [1] is a formal framework that is particularly suitable for the development of distributed systems. System development starts from an abstract specification that is transformed into a detailed specification in a number of correctness-preserving refinement steps. In this paper, we separately model the synchronous, semi-synchronous and asynchronous replication architectures. Event-B and the Rodin platform [11] allow us to explicitly define the data integrity and consistency properties as model invariants and compare them in all three models. We believe that the proposed approach allows the designers to gain formally grounded insights on properties of cloud data stores and their resilience.

The paper is structured as follows: in Section 2 we give a brief overview of the Event-B formalism. In Section 3 we describe the WAL mechanism as well as the general architecture of a cloud data store. In Section 4 we present a formal development of the asynchronous replication model. In Section 5, we briefly (due to similarity with the asynchronous model) overview the models for the synchronous and semi-synchronous architectures and compare the data consistency properties of three replication modes. Finally, in Section 6 we overview the related work, discuss the obtained results and outline future work.

2 Modelling in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [1]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important system properties are defined as model *invariants*. Usually, a machine has an accompanying component, called a *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$e \;\widehat{=}\; \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, a is the list of local variables, and (the event *guard*) G_e is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation R_e.

The event guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, refine old events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. In the latter case, we need to define *gluing invariants*, which define the relationship between the abstract and concrete variables. The proof of data refinement is often supported by supplying *witnesses* – the concrete values for the replaced abstract variables. Witnesses are specified in the event clause **with**.

The consistency of Event-B models, i.e., verification of model well-formedness, invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform [11] provides an automated support for modelling and verification. In particular, it automatically generates the required proof obligations and attempts to discharge them.

Event-B adopts an event-based modelling style that facilitate a correct-by-construction development of a distributed system. Since cloud data storage is a large-scale distributed system, Event-B is a natural choice for its formal modelling and verification. In the next section, we give an overview of the general data storage architecture that we will formally develop in Event-B.

3 Resilient Cloud Data Storage

Essentially, a cloud data storage can be seen as a networked online data storage available for its clients as a cloud service. Data are stored in virtualised data stores (pools) usually hosted by third parties. Physically, the data stores may span across multiple distributed servers. Cloud data storage providers should ensure that their customers can safely and easily store their content and access it from their computers and mobile devices. Therefore, there is a clear demand to achieve both resilience and high performance in handling data.

Write-ahead logging (WAL) is a standard data base technique for ensuring data integrity. The main principle of WAL is to apply the requested changes to data files only after they have been logged, i.e., after the log has been stored in the persistent storage (disk). The WAL mechanism ensures fault-tolerance because, in case of a crash, the system would be able to recover using the log. Moreover, the WAL mechanism helps to optimise performance, since only the log file (rather than all the data changes) should be written to the permanent storage to guarantee that a transaction is (eventually) committed.

The WAL mechanism has been thoroughly studied under the reliable persistent storage assumption, i.e., if the disk containing the log never crashes. However, in the cloud implementing such a highly-reliable data store is rather unfeasible. Therefore, to ensure fault tolerance, F-Secure has proposed a solution that combines WAL with replication. The resulting system – distributed data

store (DDS) – consists of a number of nodes distributed across different physical locations. One of the nodes, called *master*, is appointed to serve incoming data requests from DDS clients and report on success or failure of such requests. As a result, for instance, the client may receive an acknowledgment that the data have been successfully stored in the system. The remaining nodes, called *standby nodes*, contain replicas of the stored data.

Each request received by the master is translated into a number of reading and writing commands. These commands are first recorded in the *master log* and then applied to the stored data. After this, an acknowledgement is sent to the client. (In the non-replicated version of WAL widely used in the databases, an acknowledgement to the client is sent already after the request is written in the log). The standby nodes are constantly monitoring and streaming the master log records into their own logs, before applying them to their persistent data in the same way. Essentially, the standby nodes are continually trying to "catch up" with the master. If the master crashes, one of the standby nodes is appointed to be the master in its stead. At this point, the appointed standby effectively becomes the new master and starts serving all data requests.

DDS can implement different models (architectures) of logging. In the asynchronous model, the client request is acknowledged after the master node has performed the required modifications in its persistent storage. The second option – the cascade master-standby – is a semi-synchronous architecture. The client receives an acknowledgement after both the master and its warm standby (called upper standby) has performed the necessary actions. Finally, in the synchronous model, only after all replica nodes have written into their persistent storage, i.e., fully synchronised with the master node, the transaction can be committed. Obviously, such different logging models deliver different resilience guarantees.

In our formal modelling, we aim at formally defining and comparing data integrity and consistency properties that can be ensured by each architecture. In the next section, we present a development of the asynchronous architecture.

4 Modelling the Asynchronous Architecture

In the asynchronous model of replication, the standby nodes may stream the master log records only after the required changes have been committed and reported to the client. If the master crashes shortly after committing the required modifications, some changes will not be replicated thus leading to an inconsistent system state. In particular, this might happen because a standby node has not yet received (streamed) all the master log records when the master failed. To minimise such a data loss, the node that has the freshest (and hence the most complete) copy of the master log is chosen to become the next master. A graphical representation of the system architecture is shown in Fig.1.

Abstract specification. The initial model – the machine Replication1_m0 abstractly describes the behaviour of the master node – receiving and processing of the received requests. The overall model structure is given on Fig.2.

The variable *comp*, $comp \subseteq COMP$, represents the dynamic set of active system nodes (data stores), where $COMP$ is a set (type) of all available data

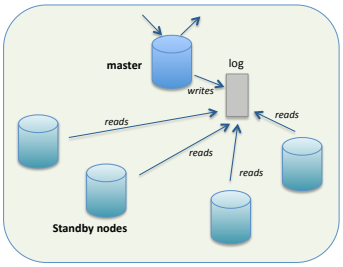


Fig. 1. Asynchronous model

stores. The variable *master*, such that $master \in comp$, represents the master node. The other variables *buffer*, *inprocess* and *processed* represent the received data requests at different stages of their processing by the master. They are modelled as disjoint sets of the abstract data type $REQUESTS$. In particular, the variable *buffer* stores the requests that have been received by the master and are waiting to be handled. The variable *inprocess* contains the requests that the master node is currently processing, while the variable *processed* keeps the requests that are completed and acknowledged to the client.

The event RequestIn specifies arriving of a new request to the master. Processing of the received requests and sending notifications to the client are modelled by the events Process and RequestOut respectively. The events update the variables *buffer*, *inprocess* and *processed* to reflect the progress in request handling.

Machine Replication1_m0
Variables $comp, master, buffer, inprocess, processed$
Invariants $comp \subseteq COMP \ \wedge \ master \in comp \ \wedge \ buffer \subseteq REQUESTS$ ...
Events

RequestIn $\widehat{=}$ // arriving of a new request
any r
where $r \in REQUESTS \ \wedge \ r \notin buffer \ \wedge \ r \notin inprocess \ \wedge \ r \notin processed$
then $buffer := buffer \cup \{r\}$
end

Process ... // request processing

RequestOut $\widehat{=}$ // completion of a request
any r
where $r \in inprocess$
then $processed := processed \cup \{r\} \parallel inprocess := inprocess \setminus \{r\}$
end

ChangeMaster $\widehat{=}$ // changing the master
any $new\_master, n\_buffer, n\_inprocess, n\_processed$
where
$new\_master \in comp \ \wedge \ new\_master \neq master \ \wedge \ n\_inprocess \cap n\_buffer = \varnothing \ \wedge$
$n\_inprocess \cap n\_processed = \varnothing \ \wedge \ n\_processed \cap n\_buffer = \varnothing \ \wedge$
$n\_buffer \cup n\_inprocess \cup n\_processed \subseteq buffer \cup inprocess \cup processed$
then
$master := new\_master \parallel buffer := n\_buffer \parallel inprocess := n\_inprocess \parallel$
$processed := n\_processed \parallel comp := comp \setminus \{master\}$
end

CompActivation ... // activation of a new system component
CompDeactivation ... // deactivation of a system component

Fig. 2. Asynchronous model: the abstract model

The event ChangeMaster models a crash of a master and selection of a new master. One of the remaining nodes is non-deterministically chosen to become a new master, while the old master is removed from the set of active nodes. Due to possible data loss, the requests being handled by the new master may be only a subset of those of the failed master. This is reflected by the guard condition:

$$n\_buffer \cup n\_inprocess \cup n\_processed \subseteq buffer \cup inprocess \cup processed,$$

where n_buffer, $n\_inprocess$, and $n\_processed$ are the corresponding data structures of the new master.

Finally, the last two events, CompActivation and CompDeactivation, model a possibility to add new data storage nodes from the cloud and remove some currently active nodes from the system respectively. Only standby nodes can be activated and deactivated in this way.

First Refinement. In the first refinement step (defined by the machine Replication1_ref1), we extend the abstract model by explicitly representing the behaviour of the standby nodes.

To accomplish this, we lift the abstract variables *buffer*, *inprocess*, *processed* to become node-dependent functions. In Event-B, this is achieved by data refinement that replaces these variables with the new variables *comp_buffer*, *comp_inprocess* and *comp_processed*. The following gluing invariants are defined to to prove correctness of data refinement:

$$comp\_buffer \in comp \to \mathbb{P}(REQUESTS) \ \wedge \ comp\_buffer(master) = buffer \wedge$$
$$comp\_inprocess \in comp \to \mathbb{P}(REQUESTS) \ \wedge \ comp\_inprocess(master) = inprocess \wedge$$
$$comp\_processed \in comp \to \mathbb{P}(REQUESTS) \ \wedge \ comp\_processed(master) = processed.$$

The overview of the refined model is presented in the Fig. 3. The set of model events includes the refined versions of the abstract events (RequestInMst, ProcessMst, RequestOutMst, ChangeMaster, CompActivation, and CompDeactivation) as well as new events describing the behaviour of standby nodes.

We refine the event ChangeMaster to a deterministic procedure of choosing the node with the freshest log as a new master to the failed master. We formulate this condition as a new guard of the event ChangeMaster in the following way:

$$\begin{aligned}(\forall c \cdot\ & c \in comp \wedge c \neq new\_master \wedge c \neq master \ \Rightarrow \\ & comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq \\ & comp\_buffer(new\_master) \cup comp\_inprocess(new\_master) \cup \\ & comp\_processed(new\_master)). \quad (1)\end{aligned}$$

The standby nodes are continuously streaming the master log. Essentially, this means that, as soon as the master node completes the request(s), i.e., performs the required modifications in its persistent storage, the standby nodes start copying the corresponding entries in the master log. This behaviour is modelled by the new event RequestInStb. Similarly as for the master node, the processing of requests and their completion by the standby nodes are respectively modelled by the events ProcessStb and RequestOutStb.

In our model, we assume that the nodes might become temporary unavailable (i.e., crush and recover). The new variable *failed*, $failed \subseteq comp$, is introduced to

Fig. 3. Asynchronous model: the first refinement

store such failed nodes. The new event CompFailure and CompStbRecovery model possible node crashes and recoveries correspondingly.

Now we are ready to formulate and prove some data consistency properties expressing the relationships between the requests handled by the master and those handled by the standby nodes. Since any standby node is continuously copying the master log, we can say that any standby node is logically "behind" the master node. Mathematically, this means that all the standby requests (no matter what stage of processing they are in) are subset of those of the master node. Moreover, all the requests that are now handled by a standby node should have been already completed by the master before. We can formulate these two properties as the following system invariants:

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_buffer(master) \cup comp\_inprocess(master) \cup comp\_processed(master)), \quad (2)$$

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(master)). \quad (3)$$

As it turns out, the last property cannot be proven as an (unconditional) invariant of the system. Indeed, it can be violated right after one of the standby nodes is appointed the new master. A short transitional period may be needed for the new master to "catch up" with some of the standby nodes that got ahead by handling the requests still not committed by the new master. It is easy to show termination of this transitional period, since all such standby nodes are blocked from reading any new requests from the master until the master catches up with them by processing its requests.

We can formally model this transitional stage by introducing the variable $in\_transit$, $in\_transit \in BOOL$. The variable obtains the value $TRUE$ when a new master is appointed, and reobtains the value $FALSE$ (in the new event TransitionOver) when all the remaining standby nodes have the requests already processed by the new master.

Then we can reformulate the property (3) as a system invariant and prove its preservation:

$$in\_transit = FALSE \;\Rightarrow\; (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(master)). \quad (4)$$

Second Refinement. In the previous refinement step we introduced the standby nodes and their interactions with the master. We also modelled how the received data requests are transferred through the different processing stages on the master and standby sides. The variables *buffer*, *inprocess* and *processed* were used to store incoming, processing and processed requests. The goal of our second refinement step is explicitly model the WAL mechanism and the resulting interdependencies between the master and standby logs.

Mathematically, any log can be represented as a sequence, i.e., as a function of the type

$$any\_log \;\in\; 1..k \;\rightarrow\; ELEMENTS,$$

where k is the index of the last written element.

In our case, we want to store in the node log all the requests – received, being processed, or completed. This can be represented as partitioning of the component log into three separate parts. To achieve that, we introduce three variables $index\_written$, $index\_inprocess$, and $index\_processed$:

$$index\_written \in comp \rightarrow NAT,\; index\_inprocess \in comp \rightarrow NAT,$$

$$index\_processed \in comp \rightarrow NAT,$$

such that

$$\forall c \cdot c \in comp \;\Rightarrow\; index\_inprocess(c) \leq index\_written(c),$$
$$\forall c \cdot c \in comp \;\Rightarrow\; index\_processed(c) \leq index\_inprocess(c).$$

For any component c, $index\_written(c)$ defines the index of the last written log entry, $index\_inprocess(c)$ – the index of the last request being processed, and $index\_processed(c)$ – the index of the last completed request. Graphically, this can be represented as shown in Fig.4.

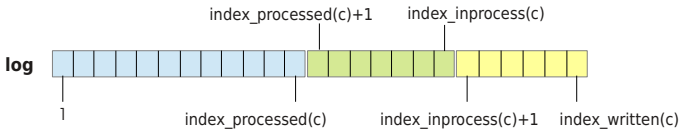


Fig. 4. The log partition

Then the logs of all the components can be defined as the following function:

$$log \in comp \to (NAT \nrightarrow REQUESTS),$$

such that

$$\forall c \in comp \cdot \mathbf{dom}(log(c)) \;=\; 1 \,..\, index\_written(c),$$

where **dom** is the functional domain operator.

The function *log* is introduced to replace (data refine) the abstract variables *comp_buffer*, *comp_inprocess*, and *comp_processed*. To prove correctness of such data refinement, the following gluing invariants are added:

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_inprocess(c) + 1 \,..\, index\_written(c)] = comp\_buffer(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_processed(c) + 1 \,..\, index\_inprocess(c)] = comp\_inprocess(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index\_processed(c)] = comp\_processed(c),$$

where $R[S]$ denotes relational image of R with respect to the given set S.

An introduction of the sequential representation of the component log allows us to refine some proven invariants as well as prove some new ones. For instance, the invariant property (4) now can be reformulated in terms of new variables:

$$in\_transit = FALSE \;\Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow$$
$$log(c)[1 \,..\, index\_written(c)] \subseteq log(master)[1 \,..\, index\_processed(master)]). \quad (5)$$

The formulated data refinement also affects all the events where the abstract variables were used. For instance, the event RequestOutMst (see Fig. 5) now specifies completion of master request processing by recording this in the node log, i.e., by increasing $index\_processed(master)$.

We can refine the procedure of choosing a new master by reformulating the guard condition (1) of the event ChangeMaster as follows:

$$\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq new\_master \Rightarrow$$
$$index\_written(c) \leq index\_written(new\_master). \quad (6)$$

Here we check that the new candidate for the master has the largest $index\_written$, i.e., the freshest log copy. The other events are refined in a similar way. The overview of the refined model is presented in Fig. 5.

Moreover, we can explicitly formulate and prove the log data integrity properties as model invariants:

$$\forall c, i \cdot c \;\in\; comp \wedge i \;\in\; 1 \,..\, index\_written(c) \Rightarrow log(c)(i) \;=\; log(master)(i),$$

$$\forall c1, c2, i \cdot c1 \in comp \wedge c2 \in comp \wedge i \in 1 \,..\, index\_written(c1) \wedge$$
$$i \in 1 \,..\, index\_written(c2) \;\Rightarrow\; log(c1)(i) = log(c2)(i). \quad (7)$$

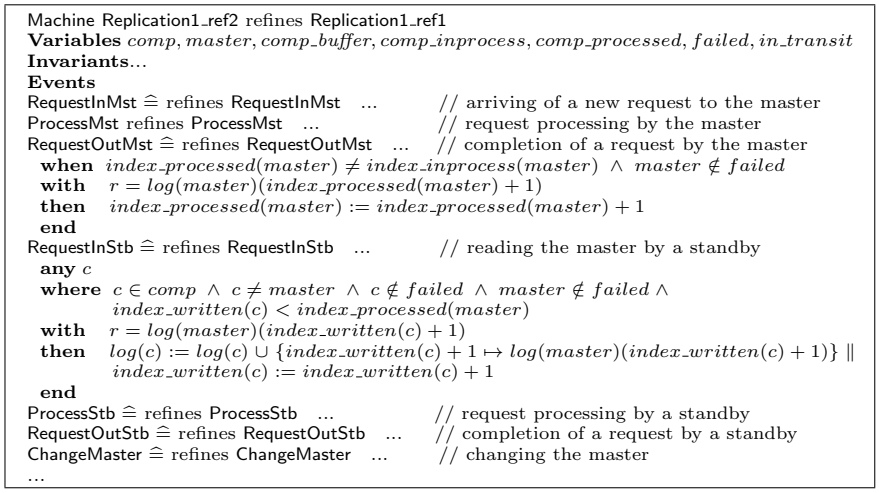


Fig. 5. Asynchronous model: the second refinement

These properties state that the corresponding log elements of any two storage (master or standby) nodes are always the same. In other words, all logs are consistent with respect to the log records of the master node.

5 The Cascade Master-Standby and Synchronous Architectures

An alternative, semi-syncronous replication model is the *cascade master-standby*. Besides the *master* node that serves incoming data base requests, we single out another functional node – *upper standby*. The upper standby node starts streaming the master log as soon as the master records the requests in its log. Moreover, the master node waits until the upper standby reads its processed records and, only after that, commits the changes and reports to the client.

In its turn, the other standby nodes are constantly monitoring and streaming the upper standby log records into their own logs and applying them in the same way as described in Section 4. Essentially, the standby nodes are continually trying to catch up with the upper standby.

If the master node goes down, the upper standby node is automatically appointed to be the master in its stead. Moreover, the next candidate for the new upper standby node becomes the node that is closest (with respect to the copied log file) to the current upper standby.

Let us note that this proposed cascade replication mode allows to decrease the possibility of loss of the committed changes if the master node fails. Indeed, at that point, when the master node fails, the upper standby node had already recorded all the changes that were committed and reported to the client by master before. Therefore, such an architectural solution increases the system resilience. A possibility of data loss leading to an inconsistent system state is still

present. However, for this to happen, the master node and the upper standby node should both fail in a very short time period. A graphical representation of the system architecture is shown on Fig.6.

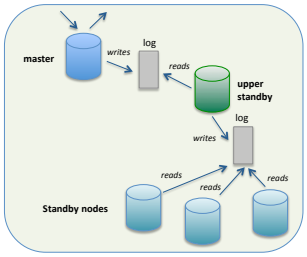


Fig. 6. Cascade system architecture

The formal development of the proposed replication model consists of an initial specification and its two refinements. The initial model abstractly describes the system behaviour focusing on the master and the upper standby nodes. The first refinement step introduces the remaining standby nodes and their interoperation with the upper standby, while the second refinement explicitly models the sequential logging mechanism and the interdependencies between the master, the upper standby and others standby logs. Let us note that the development is similar to that of the asynchronous model. Due to the lack of space we will only highlight the most significant differences between them.

Abstract specification. In the initial model defined by the machine Replication2_m0 we focus on the master and upper standby components and their interoperation. The overall model structure is given on Fig. 7.

In addition to the master node, we single out one more node to serve as an upper standby node. We model this by introducing the variable $ups\_stanbdy$, such that $ups\_stanbdy \in comp$ and $ups\_standby \neq master$.

The variables m_buffer, $m\_inprocess$, $m\_processed$ represent the received requests at different stages of their processing by the master. Similarly, the variables ups_buffer, $ups\_inprocess$, $ups\_processed$ are introduced to model the respective data structures for the upper standby. The events RequestInMst, ProcessMst, RequestOutMst and RequestInUps, ProcessUps, RequestOutUps specify the corresponding request stages for the master and upper standby nodes.

The master node can not commit the changes until the upper standby reads them. We model this requirement by adding the following guard condition in the event RequestOutMst:

$$r \in ups\_buffer \cup ups\_inprocess \cup ups\_processed.$$

The process of changing of the master node by the upper standby is modelled by the event ChangeMaster. The event also specifies the selection procedure of a new upper standby. Due to possible data loss, the requests being handled by the new upper standby may be only a subset of those of the current upper standby:

$$n\_buffer \cup n\_inprocess \cup n\_processed \subseteq ups\_buffer \cup ups\_inprocess \cup ups\_processed.$$

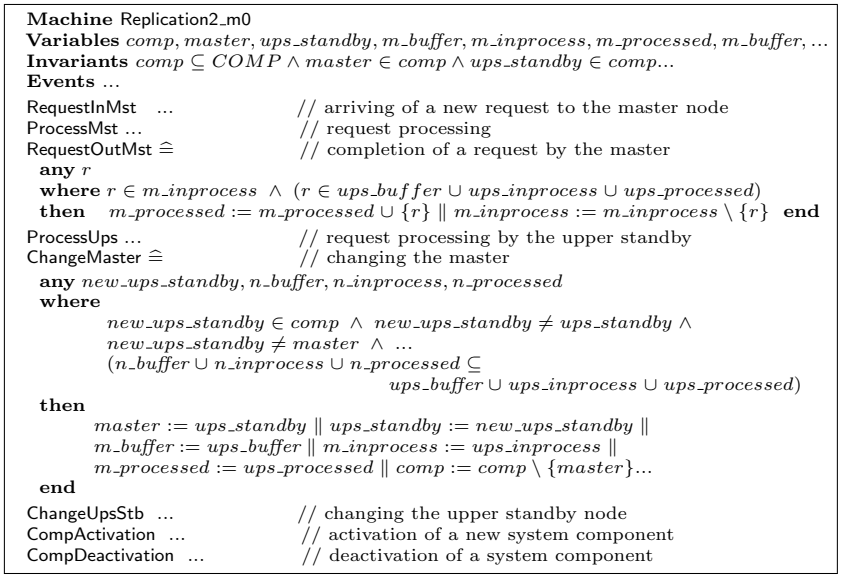


Fig. 7. Cascade architecture: abstract model

Moreover, a similar event, ChangeUpsStb, models the selection of a new upper standby in the case when the current one fails.

First Refinement. In the first refinement step we extend the abstract model by explicitly introducing the behaviour of the remaining standby nodes. Similarly as for the asynchronous model, we data refine the abstract variables *m_buffer*, *m_inprocess*, *m_processed* and *ups_buffer*, *ups_inprocess*, *ups_processed* by the new functional variables *comp_buffer*, *comp_inprocess* and *comp_processed*.

In addition, a number of the new events are added to describe the behaviour of standby nodes, node failures and recovery (RequestInStb, ProcessStb, RequestOutStb, CompFailure, CompStbRecovery).

As for the asynchronous model, we can formulate and prove data consistency properties between the involved components. The property (2) (stating that a standby node is always behind the master in terms of handled requests) corresponds to two properties for the cascade replication mode: the first one stating this property between any standby and the upper standby, while the second one stating the same property between the upper standby and master nodes.

$$(\forall c \cdot c \in comp \wedge c \neq master \ \wedge c \neq ups\_standby \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby)), \quad (8)$$

$$comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby) \subseteq comp\_buffer(master) \cup comp\_inprocess(master) \cup comp\_processed(master), \quad (9)$$

The property (4) for the asynchronous mode expresses the relationships between the processed requests of the master node and read requests of the standby nodes. This property again corresponds to two properties for the cascade mode: one between the upper standby and remaining standbys, and the other one between the master and upper standby nodes. In both cases, the properties may be violated for a short period (indicated by $in\_transit = TRUE$) right after a new upper standby node is chosen to replace a failed one:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups\_standby \Rightarrow (comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(ups\_standby)), \quad (10)$$

$$in\_transit = FALSE \Rightarrow (comp\_processed(master) \subseteq comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby)). \quad (11)$$

Note how the requirement that the master cannot commit a request before it is read by the upper standby reverses the inclusion relationship in the (11).

Second Refinement. The goal of our second refinement step is explicitly model the write-ahead logging mechanism and the resulting interdependencies between the master, upper standby and other standby logs.

We data refine the abstract variables *comp_buffer*, *comp_inprocess*, and *comp_processed* by the introduced function *log*. The following gluing invariants allow us to prove correctness of such a data refinement:

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_inprocess(c) + 1 \,..\, index\_written(c)] = comp\_buffer(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_processed(c) + 1 \,..\, index\_inprocess(c)] = comp\_inprocess(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index\_processed(c)] = comp\_processed(c).$$

Introducing the sequential representation of the component log allows us to reformulate some proven invariants as well as prove some new ones. For instance, the invariant properties (10) and (11) now can be reformulated in terms of the new variables as follows:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups\_standby \Rightarrow log(c)[1 \,..\, index\_written(c)] \subseteq log(ups\_standby)[1 \,..\, index\_processed(ups\_standby)]). \quad (12)$$

$$in\_transit = FALSE \Rightarrow log(master)[1 \,..\, index\_processed(master)] \subseteq log(ups\_standby)[1 \,..\, index\_written(ups\_standby)]. \quad (13)$$

Finally, the log data integrity properties (in the exact form as in (7)) are formulated and proved for this replication mode as well.

Synchronous Architecture. The last development formalises the *synchronous replication architecture*, which can be considered as a combination of both asynchronous and cascade models. The essential differences of this model are following. The standby nodes start streaming the master log records as soon as master

records the commands in its log. Moreover, the master node waits until *all the standby nodes* read processed records from its log and, only after that, commits the corresponding changes and reports to the client. If the master goes down, one of the standby nodes is appointed to be the master in its stead. Essentially, it is a generalisation of the cascade model where all the standby nodes play the role of upper standby.

This architecture allows to avoid a possibility of loss of the committed changes if the master fails. Indeed, at that point, all the standby nodes have already recorded all the changes that were committed and reported to the client by master. On the other hand, the necessity for the master to synchronise in such a way with all the standbys may negatively affect the performance of this model.

Developing the formal model of this architecture, we essentially repeat the refinement steps of the asynchronous model. In particular, the initial model is the same as the abstract model presented on Fig.2. In the first refinement step, in the RequestOutMst event modelling the commitment of the changes by master, we have to impose an additional restriction for this behaviour. Namely, the master node can not commit the changes until all the standby nodes have read them. We model this requirement by adding the following guard condition to the event:

$$\forall c \cdot c \in comp \wedge c \neq master \Rightarrow \\ r \in comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c),$$

where we check that the request r has already been recorded by all the standby nodes. Moreover, in the eventRequestInStb, we relax its guard by allowing to copy the master log as soon as the master records requests in its log.

Similarly as for the first two models, we formulate and prove log data consistency properties. Specifically, the property (2), stating that the standby nodes are continuously trying to catch up with the master in terms of handled requests, can be proved for this architecture as well. Moreover, since the master can not commit the changes until the all standbys have read the corresponding log records, it means that all the requests committed by the master have been previously read by all standbys. We can formulate this property as follows:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow \\ comp\_processed(master) \subseteq comp\_buffer(c) \cup comp\_inprocess(c) \cup \\ comp\_processed(c)). \quad (14)$$

Note that, once again, this property can be violated right after a new master is appointed and thus a transitional period is needed. This property is very similar to that of (11) (for the cascade architecture) and is inverse, with respect to the inclusion relation, to that of (4) (for the asynchronous architecture).

As in the previous two developments, in the second refinement step we introduce component logs as sequences. In terms of the new variables, the (14) property can be then reformulated as follows:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow \\ log(master)[1 \mathinner{..} index\_processed(master)] \subseteq log(c)[1 \mathinner{..} index\_written(c)]). \quad (15)$$

Finally, the log data integrity properties (7), stating that the corresponding log elements of any two storage components are always the same, are proved for this model as well. The full formal developments can be found in [9].

Proof Statistics To verify correctness of the presented formal developments, we have discharged around 400 proof obligations for the first model, more than 750 proof obligations for the second model, and around 400 for the third model. In total, around 90% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. The proof statistics in terms of generated proof obligations for the presented Event B developments is shown in the Table 1. The numbers represent the total number of proof obligations and the percentage of manual effort for each model in each refinement step. The whole development and proving effort has taken around one person-month.

Table 1. The proof statistics

	Asynchronous model			**Cascade model**			**Synchronous model**		
step	Total	Manual	Manual %	Total	Manual	Manual %	Total	Manual	Manual %
m0	18	0	0	53	0	0	18	0	0
ref1	145	0	0	257	1	0.3	146	0	0
ref2	193	42	21.7	442	75	16.9	232	42	18.1
Overall	356	42	11.7	752	76	10.1	396	42	10.6

6 Conclusions and Related Work

In this paper, we formalised an industrial approach to implementing resilient cloud data storage. To ensure resilience, F-Secure combined the WAL mechanism with the log replication. We have formally expressed data integrity and consistency properties in three different replication architectures and explicitly identified situations that lead to data loss. These properties can inform industry practitioners on resilience guarantees inherent to each solution. The proposed modelling approach can facilitate early design exploration and evaluate benefits of different fault tolerance mechanisms in implementing resilience requirements.

The problem of data consistency in replicated data stores, in particular for the cloud, has been actively studied in both database and fault tolerance communities, e.g., see [2, 8]. However, most of these approaches aim at proposing new protocols guaranteeing correctness of data replication. For instance, the work [8] presents a Dynamic Multi-Replica Provable Data Procession scheme. It is based on probabilistic encryption used to periodically verify the correctness and completeness of multiple data copies stored in the cloud. In our work we aim at modelling architectural aspects of data stores and verifying qualitative characteristics of data handling in the cloud.

Formal analysis of data base replication protocols has been proposed in [6]. The authors establish safety and liveness correctness criteria that need to be verified by a replication protocol. In contract, in our work we focus on ensuring data integrity and consistency properties based on the concrete write-ahead logging mechanism.

The WAL has been investigated in [4, 5]. In those works the authors analyse the performance aspects of this technique. They distinguish four types of the delays that the WAL mechanism can impose on transaction handling and propose an approach to increase log scalability. In our work, we focus on verifying properties concerning data consistency and data loss under possible failure scenarios using the WAL mechanism.

In [12] the authors investigate data properties in the presence of server failures, however considering the full ACID properties. A formal modelling of fault-tolerant transactions for replicated database systems using Event-B has been also undertaken by D. Yadav et al. [13]. The work focuses on data atomic commitment of distributed transactions.

In our future work we are planning to analyse performance and other quantitative characteristics of different replication architectures. It particular, it would be interesting to integrate probabilistic verification to evaluate the trade-offs between performance and reliability.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. Barsoum, A.F., Hasan, M.A.: Integrity Verification of Multiple Data Copies over Untrusted Cloud Servers. In: CCGRID 2012. pp. 829–834. IEEE Computer Society (2012)
3. F-Secure: http://www.f-secure.com/en/web/operators_global/content-anywhere
4. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: A Scalable Approach to Logging. vol. 3, pp. 681–692. VLDB Endowment (2010)
5. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Scalability of Write-Ahead Logging on Multicore and Multisocket Hardware. The VLDB Journal 21(2), 239–263 (2012)
6. de Mendvil, J., Armendriz-Iigo, J.E., Garitagoitia, J.R., Muoz-Esco, F.D.: A Formal Analysis of Database Replication Protocols With SI Replicas and Crash Failures. The Journal of Supercomputing 50(2), 121–161 (2009)
7. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: Aries: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Transactions on Database Systems 17, 94–162 (1992)
8. Mukundan, R., Madria, S., Linderman, M.: Replicated Data Integrity Verification in Cloud. IEEE Data Eng. Bull. 35, 55–64 (2012)
9. Pereverzeva, I., Laibinis, L., Troubitsyna, E., Holmberg, M., Pöri, M.: Formal Modelling of Resilient Data Storage in the Cloud. Tech. rep., Turku Centre for Computer Science (2013)
10. PostgreSQL: WAL: http://www.postgresql.org/docs/9.2/static/wal-intro.html
11. Rodin: Event-B Platform, online at http://www.event-b.org/
12. Wei, Z., Pierre, G., Chi, C.H.: Scalable Transactions for Web Applications in the Cloud. In: Euro-Par 2009. pp. 442–453. Springer-Verlag Berlin Heidelberg (2009)
13. Yadav, D., Butler, M.: Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event-B. In: M. Butler, C. Jones, A.R., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 343–363. Springer (2006)

Paper X

Integrating Event-B Modelling and Discrete-Event Simulation to Analyse Resilience of Data Stores in the Cloud

Linas Laibinis, Benjamin Byholm, Inna Pereverzeva, Elena Troubitsyna, Kuan Eeik Tan, Ivan Porres

Originally published in: Elvira Albert, Emil Sekerinski (Eds.), *Proceedings of 11th International Conference on Integrated Formal Methods (iFM 2014)*, LNCS 8739, 103–119, Springer International Publishing Switzerland, 2014. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-10181-1_7

Integrating Event-B Modelling and Discrete-Event Simulation to Analyse Resilience of Data Stores in the Cloud

Linas Laibinis[1], Benjamin Byholm[1], Inna Pereverzeva[1,2], Elena Troubitsyna[1], Kuan Eeik Tan[3], and Ivan Porres[1]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] F-Secure Corporation, Helsinki, Finland
{linas.laibinis, benjamin.byholm, inna.pereverzeva}@abo.fi
{elena.troubitsyna, ivan.porres}@abo.fi
kuan.eeik.tan@f-secure.com

Abstract. Ensuring resilience of large data stores in the cloud is a challenging engineering issue. It requires the development techniques that allow the designers to predict the main resilience characteristics — fault tolerance and performance — at the early design stages. In this paper, we experiment with integrating Event-B modelling with discrete-event simulation. Event-B allows us to reason about correctness and data integrity properties of data stores, while discrete-event simulation in SimPy enables quantitative assessment of performance and reliability. Since testing in a real cloud environment is expensive and time-consuming, the proposed approach offers several benefits in industrial settings.

Keywords: Formal modelling, Event-B, discrete-event simulation

1 Introduction

Development and verification of cloud-based data stores constitutes a challenging engineering task. To guarantee resilience and satisfy Service Level Agreement (SLA) that regulates service behaviour with respect to its customers, the developers should ensure two main properties – data integrity and performance. To achieve this goal, F-Secure Corporation – a company providing secure data storage solutions – relies on massive replication and the non-transactional approach.

In our previous work [10], we have undertaken formal modelling of resilient data store and logically defined data integrity properties of different architectural solutions. To analyse performance/fault tolerance ratio of architectural alternatives, we have attempted to integrate quantitative verification. However, complexity of the system turned out to be prohibitive for probabilistic model checkers and the quantitative analysis, which is essential for engineering resilient cloud data stores, has not been performed.

To address this issue, in this paper we propose an approach to integrating formal modelling in Event-B with discrete-event simulation in SimPy [14] – a library and development framework in Python. Event-B [2] is a state-based approach to correct-by-construction system development. A powerful tool support

– the Rodin platform [11] – automates the development and provides us with a scalable proof-based verification. In this paper, we rely on Event-B to formally represent and verify system-level logical properties, while simulation in SimPy is used for the quantitative analysis. SimPy [14] is a popular discrete-event simulation framework offering versatility and attractive visualisation features.

To facilitate an integration with SimPy and discussions with the industrial engineers, we have created a simple graphical notation – a process-oriented model. The notation is light-weight and introduces only the core concepts of the domain together with the key artefacts required for formal modelling and simulation. Such a graphical model defines the component interactions, representation of statistical parameters as well as reactions on faults. The process-oriented model plays the role of a unifying blue-print of the system and allows us to define the structure of the Event-B and simulation models as well as provide an easy-to-comprehend visual representation to the engineers. Once the initial models are derived from the process-oriented model, the Event-B model is refined to represent and verify data integrity properties, while the simulation model is executed to analyse performance/reliability ratio, e.g., under different service and failure rates.

We believe that the proposed approach constitutes a promising direction in the development of complex resilient systems. A combination of formal modelling and simulation amplifies the benefits of both approaches. Reliance on formal modelling not only guarantees system correctness but also increases confidence in the created simulation models, while simulation supports quantitative assessment of various design alternatives.

The paper is structured as follows. Section 2 briefly presents our case study, resilient cloud data storage, which serves as a motivation of this work. Section 3 overviews the approaches we aim to integrate – Event-B and Discrete-Event Simulation. Our integration proposal is described in detail and illustrated by a small example in Section 4. In Section 5, we demonstrate how to apply the proposed approach to perform quantitative assessment of our case study. Finally, Section 6 overviews the related work and gives some concluding remarks.

2 Resilient Data Storage in the Cloud

Our work is motivated by an industrial case study – a resilient cloud data storage [10]. The system is developed by F-Secure to provide highly performant and secure storage of client data on the cloud. Essentially, a cloud data storage can be seen as a networked online data storage available for its clients as a cloud service. Cloud data storage providers should ensure that their customers can safely and easily store their content and access it from their devices. Therefore, there is a clear demand to achieve both resilience and high performance in handling data.

Write-ahead logging (WAL) is a standard data base technique for ensuring data integrity. The main principle of WAL is to apply the requested changes to data files only after they have been logged, i.e., after the log has been stored in the persistent storage. The WAL mechanism ensures fault tolerance because, in case of a crash, the system can recover using the log. The WAL mechanism also helps to optimise performance, since only the log file should be written to the permanent storage to guarantee that a transaction is (eventually) committed.

The WAL mechanism has been studied under the reliable persistent storage assumption, i.e., if the disk containing the log never crashes. However, implementing such a highly-reliable data store in the cloud is rather unfeasible. Therefore, to ensure resilience, F-Secure has chosen a solution that combines WAL with replication. The resulting system – distributed data store (DDS) – consists of a number of nodes distributed across different physical locations. One of the nodes, called *master*, is appointed to serve incoming data requests from DDS clients and report on success or failure of such requests. The remaining nodes, called *standby* or *worker nodes*, contain replicas of the stored data.

Each request received by the master is recorded in the *master log* and then applied to the stored data. After this, an acknowledgement is sent to the client. The standby nodes are constantly monitoring and streaming the master log records into their own logs, before applying them to their persistent data. If the master crashes, one of the standby nodes becomes a new master in its stead.

DDS can implement different models of logging. In the *asynchronous model*, the client request is acknowledged after the master node has performed the required modifications in its persistent storage. In the *synchronous* model, the transaction is committed only after all the replica nodes have written into their persistent storage, i.e., synchronised with the master node. Obviously, such logging models deliver different resilience guarantees in cases of component crashes.

In our previous work [10], we have formally defined and verified data integrity properties for each described architecture using the Event-B framework. Our development provided the designers with a *qualitative* assessment of system resilience. However, while developing cloud software, it is also vital to obtain a *quantitative* assessment of resilience to optimise the choice of a resource management strategy. Usually such an assessment is achieved via testing. However, testing in the cloud requires the same usage of the resources as the real system operation, and hence is expensive. Moreover, it is often hard to reproduce the conditions of the peak load and hence obtain the insights on system resilience during the stress conditions. Therefore, there is a strong demand on the integrated approaches that enable both qualitative and quantitative analysis of resilience.

The earlier in the development process such an analysis can be performed, the better architecture can be build. To address this issue, in this paper we propose an approach to integrating formal modelling in Event-B and discrete-event simulation. Next we give a short background overview of both techniques.

3 Background

Event-B. Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The important system properties to be preserved are defined as model *invariants*. A machine usually has the accompanying component, called *context*. A context may include user-defined carrier sets, constants and their properties (defined as model *axioms*).

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$e \;\widehat{=}\; \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, and (the event *guard*) G_e is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation R_e. The event guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract system specification. In a sequence of refinement steps we gradually reduce system nondeterminism and introduce detailed design decisions. The consistency of Event-B models, i.e., invariant preservation, correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform [11] provides an automated support for modelling and verification.

Discrete-event simulation. Simulation is the act of imitating how an actual system behaves over time [3]. To achieve this goal, a simulation generates an artificial system history, thereby enabling analysis of its general behaviour. It also allows sensitivity analysis, which can be highly beneficial in the system design.

Table 1 shows an ad-hoc simulation of a bank with a single teller and 5 customers. Customers arrive at a uniformly distributed rate between 1 and 10 minutes. A customer requires a dedicated service from the teller in between 1 to 6 minutes. Each row represents a customer, with an identifier in column (1). Next columns show the generated random inter-arrival times, (2), specific computed arrival time for each customer, (3), and generated random service times, (4). From this information, we can derive a system history and study its performance. The conducted simulation allows us to obtain the following system estimates:

Average time in system $\frac{18}{5} = 3.6$ min
The clerk is idle $\frac{9}{25} = 36$ % of the time
Average queuing time $\frac{2}{5} = 0.4$ min
Ratio of customers having to queue $\frac{1}{5} = 20$ %
Average queuing time for those that queued $\frac{2}{1} = 2$ min

One type of simulation is known as discrete-event simulation (DES). In a DES, system state remains constant over an interval of time between two consecutive *events*. Thus events signify occurrences that change the system state. Events can be classified as either internal or external. *Internal events*, e.g., the bank teller has finished serving a customer, occur within the modelled system. *External events*, like customer arrivals, occur outside the system, but still affect it. A simulation is run by a mechanism that repeatedly moves simulated time forward to the starting time of the next scheduled event, until there are no more events [13].

Architecturally, a DES system consists of a number of *entities* (e.g., components, processes, agents, etc.), which are either producers or recipients of discrete events. Entities can have *attributes*, e.g., the busy status of the bank teller. There are two kinds of entities: *dynamic entities*, moving into or out of the system, like

Table 1: Ad-hoc simulation of a bank system with one teller

(1) Client Count	(2) Arrival Interim	(3) Arrival Time	(4) Service Time	(5) Service Begins	(6) Service Ends	(7) System Time	(8) Idle Time	(9) Queue Time
1	—	0	5	0	5	5	0	0
2	3	3	2	5	7	4	0	2
3	6	9	5	9	14	5	2	0
4	10	19	3	19	22	3	5	0
5	5	24	1	24	25	1	2	0
						18	9	2

the bank customer, and *static entities*, serving other entities, like the bank teller. Static entities can often be represented as *resources*. Waiting for a particular event to occur can lead to a *delay*, lasting for an indefinite amount of time. In other cases, the time estimate may be known apriori, e.g., when bank customers receive service by the teller. Events can be also *interrupted* and pre-empted, e.g., in reaction to component failures or pre-defined high-priority events.

There are four primary simulation paradigms [3]: process-interaction, event-scheduling, activity scanning, and the three-phase method. Our simulation model uses SimPy [14], a simulation framework based on process-interaction in Python.

Motivation and plan for integration. DES constitutes an attractive technology for quantitative assessment of various characteristics of cloud applications. Firstly, it allows the designers to perform various "what-if" type of analysis that demonstrates sensitivity of the service architecture to changes of its parameters. For instance, it gives an insight on how the system reacts on peak-loads, how adding new resources affects its performance, what is the relationships between the degree of redundancy and fault tolerance, etc. Secondly, while simulating the service behaviour, the designers also obtain the insights on which parameters should be monitored at run-time to optimise a resource allocation strategy. However, to obtain all the above-mentioned benefits, we have to ensure that the simulation models are correct and indeed representative of the actual system. This is achievable via integration of simulation with formal modelling.

To adequately model complex cloud services, we need a framework with a good automated tool support and scalability. We have chosen Event-B because it satisfies these criteria and has been successfully applied to model data stores in the cloud [10]. However, the fine granularity of Event-B models made it cumbersome to communicate the modelling decisions across a diverse team of experts. Therefore, we needed an easy-to-understand light-weight graphical notation that would allow us to generate both Event-B and simulation models in a compatible way.

One alternative to creating a visual system model would be to choose one of the existing architectural languages. However, to achieve simplicity and comprehensibility, we decided against it. The introduced graphical notation, called a *process-oriented model*, is domain-specific and minimal in a sense that each element is introduced only if it is required either in our formal modelling or a DES representation. It captures only the key concepts of the domain and hence alleviates the burden of customising a general-purpose architectural language. The proposed approach also gives us a full control over defining the interpretation of all elements of the introduced graphical notation in Event-B and SimPy mo-

dels, thus ensuring a mutually-compatible derivation of these models. Once the models are derived from the common process-oriented model, each of them is used independently. The Event-B model is refined to reason about the logical system properties, while the simulation model is exercised to perform the quantitative analysis. Obviously, if the simulation indicates that the chosen architecture is unable to fulfil the target SLA, the architecture should be amended together with its process-oriented model. This inevitably leads to redesign of the corresponding Event-B and SimPy models to faithfully represent the changed architecture.

As a result of such integration, we gain more flexibility and control over the simulation models. We can experiment more freely with different service configurations and perform sensitivity analysis in a more efficient way. The formal backbone gives us more confidence in the simulation models and we can clearly see the entire effect of model changes, thus alleviating the verification burden.

4 Process-Oriented Model

In this section we will present a process-oriented system model that serves as a "common ground" for both formal modelling in Event-B and simulation in SimPy. The model has the associated graphical notation for representing system architecture in terms of its units (components and processes) and interaction mechanisms between these units.

In general, we are interested in modelling, simulating, and analysing the systems that have the following characteristics:

- A system consists of a number of parallel processes, interacting asynchronously by means of discrete events;
- System processes can be grouped together into a number of components. The discrete events triggering interactions between the component processes then become *internal process events*, while the remaining discrete events can be considered as the *external component interface* for its interaction with other components or the environment;
- Within a process, execution follows the pre-defined scenario expressed in terms of functional blocks (activities) and transitions between them. Each such functional block is typically associated with particular incoming events the process reacts to and/or outgoing events it produces;
- A system component can fail and (in some cases) recover. In other words, component failures and recovery mechanisms are a part of the component description. They can be described as the special component processes simulating different types of failures and recovery procedures of the component;
- Some events (e.g., component failures) should be reacted on immediately upon their occurrence, thus interrupting the process current activities. Such special events (*interrupts*) are explicitly described in the component description and associated with dedicated functional blocks (interruption handlers).

An example of such a component is graphically presented on Fig.1. The component interface consists of one incoming event (*arrival_evn*) and two outgoing events (*rejection_evn* and *completion_evn*). The component itself contains two processes describing its "nominal" behaviour: the first one stores requests to perform a

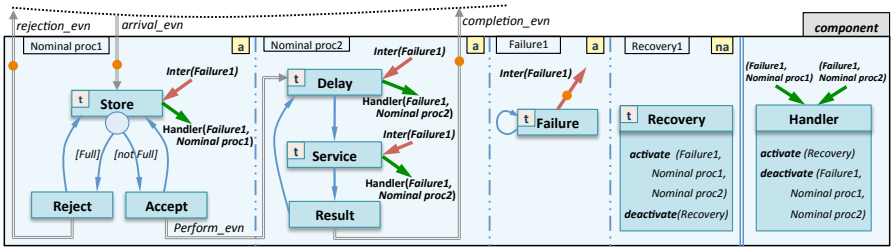


Fig. 1. Example of a system component

certain service, and the second one performs a requested service and returns the produced results. The internal event *perform_evn* triggers the request execution by the second process. In addition, the component includes the *Failure* and *Recovery* processes to simulate possible component failures and its recovery. More specifically, the *Failure* process generates an internal interrupt event for both nominal processes, which is then handled by the *Handler* block. In general, a component can have several such processes and handler blocks.

In our process-oriented model, time progress is associated with either waiting for an incoming event or an internal activity requiring time (e.g., data processing of a received service request, see Service in Fig.1). Only such functional blocks (marked by "t" in a diagram) can be interrupted.

Some functional blocks may indicate activities related to accessing the underlying storage resource (e.g., putting the received data into a buffer). Such blocks are decorated by a circle from below, for example, see Store in Fig.1.

The component state is mostly hidden in a process-oriented model, focusing instead on the required control flow and interaction between the processes. However, sometimes we need to reveal a part of this state to be able to constrain incoming or outgoing events as well as internal transitions between functional blocks. For this purpose, we use the following pre-defined component attributes:

- (for each component) the component unique identifier id;
- (for each process) the process activity status PAS, which can be either *Active* or *Inactive*. Changing this attribute to *Inactive* allows us to explicitly block a particular process. The process will remain blocked and thus irresponsive to any events until the attribute is not changed to Active again;
- (for each storage-related functional block) the storage availability status SAS, which can be either *Full* or *NotFull*. The attribute value reflects whether the operation of adding data into the storage can be successfully completed;
- (for each component) the component operational mode Mode, which can be one of pre-defined values for this type of a component. For instance, the component on Fig.1 can be used in two different modes, either as a "server" or a "slave". This assumption allows us to redefine the description of the Nominal proc1, see Fig.2. If the mode is "server", it forwards the request to another (slave) component by generating the outgoing event *forward_evn*.

If the attribute we are referring to is clear from the context, we will use a shorthand notation [attribute_value] to stand for the condition *attribute_name* = *attribute_value*. For instance, in Fig.2 the transitions of *Nominal_proc1* from

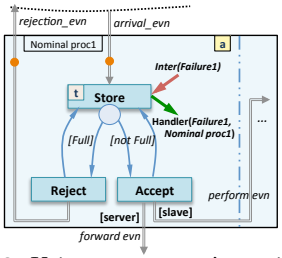


Fig. 2. Using component's attributes

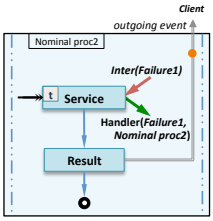


Fig. 3. Dynamic creation of processes

its Store to either Reject or Accept are dependant on whether the used storage is currently full or not. We specify this by the added conditions [Full] and [not Full] on the corresponding process transitions. Similarly, the outgoing events *perform_evn* and *forward_evn* for Accept are created depending on the component role, which is reflected by the conditions [server] and [slave] on the corresponding arrows. We also assume that all the events and internal transitions have implicit conditions [Active], checking that the process in question is not currently blocked.

If we need to change the attribute value, we again use the notation [attribute_value], however within a functional block. In addition, for quickly changing the process activity status for a number of processes, we employ another shorthand notation: **activate(Proc1, ..., ProcN)** and **deactivate(Proc1, ..., ProcN)**.

By default, we assume that the processes are created and terminated together with their encompassing component. However, sometimes we need to dynamically create and terminate component processes. Let us consider an alternative version of *Nominal_proc2*, presented in Fig.3. Here the process *Nominal_proc2* is created each time a request from *Nominal_proc1* is ready to be served. Moreover, once the outgoing event for successful service completion is created, the process is terminated (depicted as a black circle).

To collect the quantitative information about the considered system, we assume the implicit presence of a monitor component. The information is collected about the occurrence of particular events of interest. The arrows representing the events of a component to be monitored by such a component are decorated with small circles in a process-oriented model. If the system contains other components, they should be explicitly composed by matching their external interfaces.

"Common ground". A process-oriented model serves as a basis for both Event-B development and system simulation in SimPy. Translating a process-oriented model into Event-B gives us the starting point of our formal development with the already fixed system architecture and control flow between main system components. The corresponding system properties are explicitly formulated and proved as system invariants. Additional properties (e.g., the relationships between processes being active or inactive) can be verified too. By the definition of Event-B refinement, the following refined models preserve these properties, elaborating only on the newly introduced data structures and intermediate system transitions.

While translating a process-oriented model to SimPy, we augment the resulting code with concrete values for its basic quantitative characteristics, such as data arrival, service, and failure rates. This allows us to compare system performance

and reliability for different system parameter configurations. If a satisfactory configuration values can be found and thus re-design of the base process-oriented model is not needed, the simulation results does not affect the Event-B formal development and can be considered completely complementary to it.

Translating to Event-B. Here we present general guidelines how to proceed from a process-oriented model of a system component to the corresponding Event-B specification. From now on, we refer to Event-B events as operations to avoid confusion with DES events. During the translation, the respective elements of a process-oriented model may become one or several Event-B operations, the corresponding guard or action expressions within a particular operation, or the invariant predicates to be verified for the resulting Event-B model. Moreover, a number of Event-B variables standing for component attributes, incoming or outgoing events, as well as the variables ensuring the required control flow have to be introduced. The translation guidelines are summarised in Fig.4.

We will demonstrate the use of these guidelines on the component example from the previous section (Fig.1). Each functional block of the process-oriented model is translated to one or two Event-B operations. Two operations are needed in the cases when the block in question is interruptible. Then the first operation specifies a possible start of the block, while the second one models its completion. For instance, the block Store will be represented as two Event-B operations:

Store_step1 $\widehat{=}$
when $Nominal\_proc1 = Active$
$Arrived\_env = FALSE$
then $Arrived\_env :\in BOOL$
end

Store_step2 $\widehat{=}$
any j
when $Nominal\_proc1 = Active$
$failure\_interrupt = TRUE$
$Arrived\_evn = TRUE$
$j \in JOBS \setminus \{NILL\}$
then $ArrivedJob := j$
$Arrived\_evn := FALSE$
$Storage\_status :\in \{Full, NotFull\}$
end

Here the boolean variables *Arrived_evn* and *failure_interrupt* model occurrence of the corresponding system events. Moreover, the process activity attributes (e.g., *Nominal_proc1*) are represented as eponymous variables that can take values from the set {*Active,Inactive*}. Similarly, the storage availability attribute is introduced as a separate variable *Storage_status* taking values from {*Full,NotFull*}. The other functional blocks (Reject, Service, Delay, Result) are modelled in a similar way.

During the system execution a failure of the component might happen at any time. In that case, the component halts its nominal processes and is only involved in its recovery process. Upon successful recovery, the component reactivates its nominal processes. The corresponding blocks Failure and Handler are as follows:

Failure $\widehat{=}$
when $Failure\_proc = Active$
$failure\_interrupt = FALSE$
then $failure\_interrupt :\in BOOL$
end

Handler $\widehat{=}$
when $failure\_interrupt = TRUE$
then $Failure\_proc := Inactive$
$failure\_interrupt := FALSE$
$Nominal\_proc1 := Inactive$
$Nominal\_proc2 := Inactive$
$Recovery := Active$
end

Graphical notation	Description	Event B representation	SimPy representation
	A component process	The corresponding group of Event-B operations	A Simpy process based on the given generator method
	The interrupt handler of a component	The operation(s) modelling the Handler behaviour	A Simpy exception handling block
Name	A functional block within a process	An Event-B operation	Activity represented by a SimPy timeout or other generated event
	A triggerred event	The guard (condition) expression in the Event-B operation modelling a recipient activity	Some SimPy event which at least one process waits for, causing it to "wake up"
	Control flow between functional blocks	Control flow between operations (is defined by the event guards and formulated as invariant properties)	Control flow within a SimPy process, or normal Python control flow
Inter(iname) 	An interruption	A variable that models system failure	A SimPy interrupt for a target process
 *Handler(iname, prname)*	Forwarding control to the interrupt handler	The corresponding guard in the Handler operation becomes enabled	Python exception handling around yield statements, catching interrupts and forwarding to interrupt Handler so it can do possible state modifications
t	Time attribute for a functional block	An Event-B operation for such a block is split into two	Duration argument for the SimPy timeout event
a na	The initial activity status of a process (active or non- active)	The initial values for the control variable to store the activity status of the process	A SimPy process is waiting for activation control event or not
activate (name1, ...) *deactivate(name2, ...)*	A shorthand notation for changing the activity status for a processes list	Changing the corresponding control variables, thus enabling /disabling particular operations	Set flag variables or trigger events that affect the desired processes
	Monitoring data	No representation	Sending the current value to special Monitor component
	A functional block with a storage-related activity (i.e., access to a storage resource)	A dedicated variable modelling a storage + a guard condition checking whether the used storage resource is currently full or not	Using one of Python collection types (e.g., list) or SimPy Resource type, if queueing behaviour is desired
	Dynamic creation of a process	Changing the dedicated control variables, thus enabling particular operations	Creating a new SimPy process by env.process (generator)
	Dynamic termination of a process	Changing the dedicated control variables, thus disabling particular operations	SimPy processes automatically “die” when they finish execution

Fig. 4. Guidelines for integration

The Failure operation models non-deterministic occurrence of a system failure, which leads to a creation of the *failure_interrupt* event. The Handler operation models handling of this interrupt by blocking the nominal and *Failure* processes, while activating the *Recovery* process.

In this simple system, the *Recovery* process is active only when all the rest processes are inactive. This is formulated as the following invariant to be verified:

$$Recovery = Active \Leftrightarrow Nominal\_proc1 = Inactive \wedge Nominal\_proc2 = Inactive \wedge ...$$

Similarly, the invariants ensuring the required control flow and component interaction order can be added and verified in the resulting Event-B model.

Translating to SimPy. This section describes how to represent the system in Fig. 1 using SimPy. In SimPy, we represent components as classes. Component processes become SimPy processes, which are based on Python's generators. Functional blocks are represented as sequences of instructions that alter the system state and wait for events. Events are used for inter-process communication and can take optional values as arguments (see Fig. 4 for a detailed guide).

A partial code listing corresponding to the system in Fig. 1 is shown in Listing 1.1. This component has four processes. Lines 3–6 in Listing1.1 create

these processes upon initialisation of the component object, using a method for process creation provided by the SimPy simulation environment. Process creation requires a Python generator as its argument, which will become the body of the process. On Lines 8–9 we initialise two shared events which will be used to activate and deactivate the processes.

Lines 11–26 constitute the generator for Nominal_proc1. A sub-generator used for interrupt handling is defined in Lines 12–15. Upon invocation, this generator will activate the recovery process and wait until the `activation_event` is triggered, which makes Nominal_proc1 inactive while recovery is in progress. Because all processes in this example are non-terminating, every process generator contains an infinite loop, as in Lines 17–27.

Since Nominal_proc1 is interruptible, we need to surround the `yield` statement with exception handling (Lines 18–27), which will catch possible interrupts. Interrupts can only occur at `yield` statements, because an interrupt must come from another process and other processes cannot run until the active process has yielded execution, similarly to cooperative multitasking. Line 19 corresponds to the store activity, which uses a discrete resource, a store, as a pipe for interprocess communication. Execution of the process is halted until a job is put in the pipe. When a job arrives through the pipe, the status of the buffer is checked and the requested is either rejected or accepted and put in the pipe connecting Nominal_proc1 and Nominal_proc2.

Similarly to Nominal_proc1, Nominal_proc2 waits until a job arrives in its incoming pipe (Line 35). Thereafter, Line 36 simulates time required for serving the job. An interrupt may occur at either of these two `yield` statements.

The generator for the failure process spans Lines 44–50. The process waits for an exponentially distributed amount of time (46), the time between failures. Upon failure, the two nominal processes are interrupted (48–49) and the failure process becomes inactive until the recovery process triggers the `activation_event`.

Contrary to the other processes, the recovery process starts in an inactive state. We achieve this by waiting until the `recovery_activation_event` is triggered (Line 54). When activated, the recovery process waits for some time (Line 55) before activating the other processes (Lines 56–57) and becoming inactive again.

5 Applying the Integrated Approach

In this section, we will demonstrate the proposed approach on our case study – a replicated data storage in the cloud. As explained in Section 2, we consider two different system architectures: asynchronous and synchronous models.

In Fig. 5 and Fig. 6, we present the process-oriented models for the node components of these architectures. The component has a similar structure to the example we considered in the previous section. Additionally, there is a separate process *FailureDet_proc* that models reaction to a new master notification. The node components can be also employed either in the master or standby (worker) modes. In addition, each system also contains the (implicit) Monitor component as well as the Failure Detector component, responsible for detecting a failed master and then assigning a new one. The *Failure* and *Recovery* processes are similar to the ones from Fig.1 and are not presented here.

Listing 1.1. SimPy representation of the `Component` class

```
class Component:
    def __init__(self, env, ...):
        self.p1 = self.env.process(self.proc1())
        self.p2 = self.env.process(self.proc2())
        self.failure_proc = self.env.process(self.fail())
        self.recovery_proc = self.env.process(self.recover())

        self.activation_event = self.env.event()
        self.recovery_activation_event = self.env.event()

    def proc1(self):
        def handle_interrupt():
            self.recovery_activation_event.succeed()
            self.recovery_activation_event = self.env.event()
            yield self.activation_event

        while True:
            try:
                rq = yield self.arrival_pipe.get()
                if len(self.buffer) < self.capacity:
                    self.buffer.append(rq)
                    yield self.interproc_pipe.put(rq)
                else:
                    self.rejected += 1
            except Interrupt as interrupt:
                yield from handle_interrupt()
                continue

    def proc2(self):
        def handle_interrupt():
            yield self.activation_event

        while True:
            try:
                yield self.interproc_pipe.get()
                yield self.env.timeout(expovariate(self.service_rate))
                self.buffer.pop()
                self.completed += 1

            except Interrupt as interrupt:
                yield from handle_interrupt()
                continue

    def fail(self):
        while True:
            yield self.env.timeout(expovariate(self.failure_rate))
            self.failures += 1
            self.p1.interrupt('failure1')
            self.p2.interrupt('failure1')
            yield self.activation_event

    def recover(self):
        while True:
            yield self.recovery_activation_event
            yield self.env.timeout(expovariate(self.repair_rate))
            self.activation_event.succeed()
            self.activation_event = self.env.event()
```

Event-B modelling. The resulting Event-B models become starting points of our formal development, with main architectural system elements and their communication already in place. In the following refinement steps, we extend the abstract models by elaborating on the WAL mechanism and explicitly expressing the required data interdependencies between the master and standby logs.

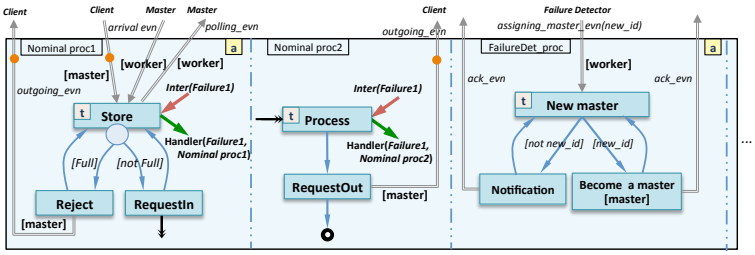


Fig. 5. Asynchronous model

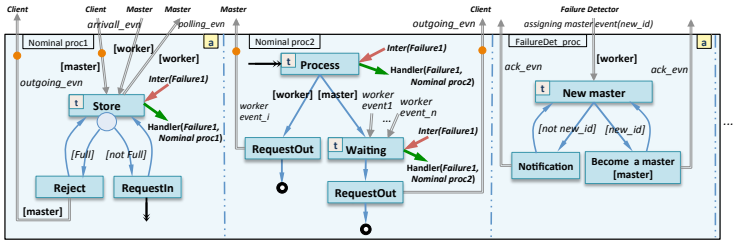


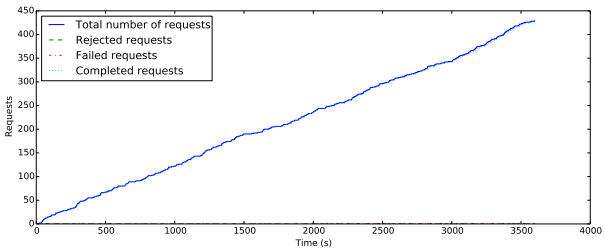
Fig. 6. Synchronous model

For instance, for the *asynchronous* model, we formulate and prove the data consistency property. Specifically, it states that all the requests that are now handled by a standby node should have been already completed by the master before. Moreover, we explicitly formulate and prove (as a model invariant) the log data integrity property stating that the corresponding log elements of any two storage nodes are always the same. In other words, all logs are consistent with respect to the log records of the master node. For more details, see [10].

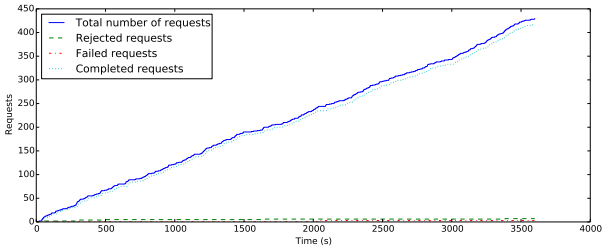
Simulation in SimPy. Creating simulations for the models in Fig.5 and Fig.6 allows us to compare the two architectures as well as evaluate how different parameters affect the results within an architecture. Let us consider the system operating with the master and 3 workers. The workers poll the master 30 times per minute. The buffer capacity is set to 5. The arrival rate is exponentially distributed with mean 7.5/min. Each component requires 5s to process a request. Components randomly fail at an exponentially distributed rate of 1.8/h, rendering them inoperable until they have been repaired, which takes 4 seconds.

Fig.7 shows the results of a simulation involving the two models. With identical operating conditions and parameters, the *asynchronous* model has higher throughput, completing 99.3 % of requests in 1 hour. This is expected, because the *asynchronous* model involves less delay than the *synchronous* one, which completes 97.2 % of requests in 1 hour. Table 2 summarises the results.

Let us take a closer look at the asynchronous architecture. What happens if the buffer capacity was lowered from 5 to 2? What if the mean failure rate then was increased to 18/h? These questions are easy to answer with DES. According to Fig. 8, reducing the buffer capacity has a large negative impact on the throughput of the system, which only manages to complete 87.4% of requests within an hour. Contrary to the previous experiment, a substantial amount of arriving requests, 11.9%, are rejected. Additionally increasing the mean failure rate by a factor



(a) Asynchronous model



(b) Synchronous model

Fig. 7. Comparison of the two models. Mean arrival rate is 7.5/min, service time is 5s, buffer capacity is 5 and mean failure rate is 1.8/h.

Table 2: Results from model comparison

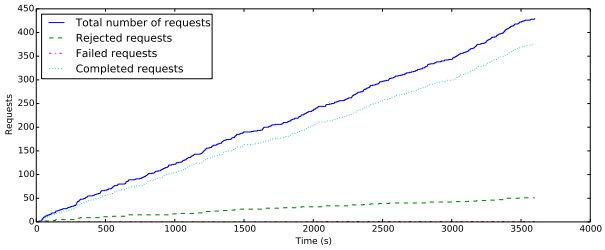
	Completed (%)	Rejected (%)	Failed (%)
Asynchronous	99.3	0	0.2
Synchronous	97.2	1.6	0.7

of 10 has less impact on the system than the previous change, with 83.4% of requests completed. Table 3 summarises the results.

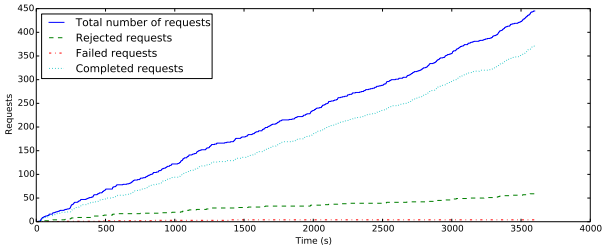
Further experiments can reveal more information about the system. For example, changing the number of workers does not have great impact on the performance of the *asynchronous* model, because its primary work-flow only involves the master. For the *synchronous* model, the number of workers does not affect performance much by itself, as processing on the workers occurs in parallel, but if the repair rate is very low, increasing the number of workers actually results in worse performance. This is because each request requires processing on each component and, whenever a component fails, the system has to wait for it to be repaired before pending requests can be completed. Increasing the number of workers in the synchronous model then results in an effective increase in the mean failure rate of the system.

6 Related Work and Conclusions

This work augments with formal modelling our previous research [5] on using DES to analyse reliability and cost of different session management policies in the cloud. The problem of inadequate support for development of cloud services has also been identified by Boer et al [4]. Similarly to us, they aim at integrating reasoning about correctness with simulation. In our case, Event-B allows us to formally represent and verify system-level properties, while in [4] the stress is put on creating executable specifications and analysis of corresponding traces.



(a) Buffer capacity is down to 2, the mean failure rate stays 1.8/h.



(b) Buffer capacity is down to 2, the mean failure rate is up to 18/h.

Fig. 8. Sensitivity analysis (Async. model). Arrival rate is 7.5/min, service time is 5 s.

Table 3: Results from sensitivity analysis with the asynchronous model

	Completed (%)	Rejected (%)	Failed (%)
Less Capacity	87.4	11.9	0.2
Less Capacity and More Failures	83.4	13.3	0.9

Our proposed process-oriented model is similar to Activity Cycle Diagrams (ACD) [6, 9] – a graphical notation to model discrete events and interactions. In particular, [6] presents an extension of ACS to enable automatic translation to Java programs, while [9] proposes extended ACD to represent the relationship between conditions and events in a discrete event system that are not covered by the classical ACD. In contrast, our process-oriented models allow us to represent a high level system architecture in terms of components, processes and their interactions. Moreover, our proposed models can be both used as a basis to formal modelling and simulation at the same time.

The WAL mechanism has been investigated in [8, 7], where the authors analyse the performance aspects of this technique. They distinguish four types of the delays that the WAL mechanism can impose on transaction handling and propose an approach to increase log scalability. In our work, we focus on integrating formal verification and DES to evaluate the system both qualitatively and quantitatively.

The problem of a formal verification and simulation-based validation is addressed in the ADVANCE project [1]. However, the focus of the proposed methodologies is related to cyber-physical systems, which are characterised by a mixture of discrete-event and continuous-time components. The proposed simulation-based approach combines the Event-B development and co-simulation with tool-independent physical components via the FMI interface [12]. In our work we deal with discrete-event systems and focus on integrating separate approaches for qualitative and quantitative reasoning about such systems.

In this paper, we have proposed a pragmatic approach to integrating formal modelling in Event-B and discrete-event simulation in SimPy. Our aim was to find a scalable solution to integrated engineering of resilient data stores in the cloud. We have succeeded in overcoming the scalability problems experienced while attempting to apply probabilistic model checking and achieved the desired goal – quantitative assessment of system resilience. Since testing cloud services in general is expensive and time consuming, we believe that the proposed approach offers benefits for the designers of cloud services.

Proof of concept integration of formal modelling and DES presented in this paper can be seen as an initial step towards creating an automated tool support for an integrated engineering environment of cloud services. Our future work will continue in two directions: one the one hand, we will create a tool for automatic translation of Event-B models in SimPy as well as work on visualisation of formal and simulation models. On other hand, we will experiment with deriving resilience monitors from system models to enable proactive resilience at run-time.

Acknowledgements. The authors would like to thank the reviewers for their valuable comments. This work is partly supported by the Need for Speed (N4S) Program (http://www.n4s.fi).

References

1. FP7 ADVANCE Project, online at http://www.advance-ict.eu/
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
3. Banks, J.: Principles of simulation. In: Banks, J. (ed.) Handbook of Simulation, pp. 3–30. John Wiley & Sons, Inc. (2007)
4. Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Wong, P.Y.: Formal modeling of resource management for cloud architectures: An industrial case study. In: Service-Oriented and Cloud Computing, pp. 91–106. LNCS 7592, Springer (2012)
5. Byholm, B., Porres, I.: Cost-Efficient, Reliable, Utility-Based Session Management in the Cloud. In: 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 102–111. IEEE Computer Society (2014)
6. De Lara Araújo Filho, W., Hirata, C.M.: Translating Activity Cycle Diagrams to Java Simulation Programs. In: ANSS'04. pp. 157–164. IEEE (2004)
7. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Scalability of Write-Ahead Logging on Multicore and Multisocket Hardware. The VLDB Journal 21(2), 239–263
8. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: A Scalable Approach to Logging. vol. 3, pp. 681–692. VLDB Endowment (2010)
9. Kang, D., Choi, B.K.: The extended activity cycle diagram and its generality. Simulation Modelling Practice and Theory 19(2), 785 – 800 (2011)
10. Pereverzeva, I., Laibinis, L., Troubitsyna, E., Holmberg, M., Pöri, M.: Formal Modelling of Resilient Data Storage in Cloud. In: Groves, L., Sun, J. (eds.) ICFEM 2013. pp. 364–380. LNCS, Springer-Verlag (2013)
11. Rodin: Event-B Platform, online at http://www.event-b.org/
12. Savicks, V., Butler, M., Colley, J., Bendisposto, J.: Rodin Multi-Simulation Plug-in. In: 5th Rodin User and Developer Workshop (2014)
13. Schriber, T.J., Brunner, D.T.: How Discrete-Event Simulation Software Works. In: Banks, J. (ed.) Handbook of Simulation, pp. 765–812. John Wiley & Sons (2007)
14. SimPy: Simulation framework in Python, online at http://simpy.readthedocs.org/

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation
197. **Espen Suenson**, How Computer Programmers Work – Understanding Software Development in Practise
198. **Tuomas Poikela**, Readout Architectures for Hybrid Pixel Detector Readout Chips
199. **Bogdan Iancu**, Quantitative Refinement of Reaction-Based Biomodels
200. **Ilkka Törmä**, Structural and Computational Existence Results for Multidimensional Subshifts
201. **Sebastian Okser**, Scalable Feature Selection Applications for Genome-Wide Association Studies of Complex Diseases
202. **Fredrik Abbors**, Model-Based Testing of Software Systems: Functionality and Performance
203. **Inna Pereverzeva**, Formal Development of Resilient Distributed Systems

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3254-1
ISSN 1239-1883

Inna Pereverzeva

Formal Development of Resilient Distributed Systems

Inna Pereverzeva

Formal Development of Resilient Distributed Systems