Masoumeh Parsa

Diagrammatic Languages and Formal Verification: A Tool-Based Approach

Åbo Akademi University

Diagrammatic Languages and Formal Verification: A Tool-Based Approach

Masoumeh Parsa

COMPUTER SCIENCE
FACULTY OF SCIENCE AND ENGINEERING
ÅBO AKADEMI UNIVERSITY
TURKU, FINLAND 2022

Supervisors

Marina Waldén
Department of Information Technologies
Åbo Akademi University
Vattenborgsvägen 3
FIN-20500 Turku, Finland

Reviewers

Professor Philipp Rümmer
Faculty of Informatics and Data Science
University of Regensburg
93040 Regensburg, Germany

Professor Einar Broch Johnsen
Department of Informatics
University of Oslo
Gaustadalléen 23B
PO Box 1080, NO-0316 Oslo, Norway

Opponent

Professor Philipp Rümmer
Faculty of Informatics and Data Science
University of Regensburg
93040 Regensburg, Germany

ISBN 978-952-12-4213-7 (Printed)
ISBN 978-952-12-4214-4 (Digital)
Painosalama, Turku, Finland 2022

Abstract

The importance of software correctness has been accentuated as a growing number of safety-critical systems have been developed relying on software operating these systems. One of the more prominent methods targeting the construction of a correct program is formal verification. Formal verification identifies a correct program as a program that satisfies its specification and is free of defects. While in theory formal verification guarantees a correct implementation with respect to the specification, applying formal verification techniques in practice has shown to be difficult and expensive. In response to these challenges, various support methods and tools have been suggested for all phases from program specification to proving the derived verification conditions. This thesis concerns practical verification methods applied to diagrammatic modeling languages.

While diagrammatic languages are widely used in communicating system design (e.g., UML) and behavior (e.g., state charts), most formal verification platforms require the specification to be written in a textual specification language or in the mathematical language of an underlying logical framework. One exception is invariant-based programming, in which programs together with their specifications are drawn as invariant diagrams, a type of state transition diagram annotated with intermediate assertions (preconditions, postconditions, invariants). Even though the allowed program states—called situations—are described diagrammatically, the intermediate assertions defining a situation's meaning in the domain of the program are still written in conventional textual form. To explore the use of diagrams in expressing the intermediate assertions of invariant diagrams, we designed a pictorial language for expressing array properties. We further developed this notation into a diagrammatic domain-specific language (DSL) and implemented it as an extension to the Why3 platform. The DSL supports expression of array properties. The language is based on Reynolds's interval and partition diagrams and includes a construct for mapping array intervals to logic predicates.

Automated verification of a program is attained by generating the verification conditions and proving that they are true. In practice, full proof automation is not possible except for trivial programs and verifying even simple properties can require significant effort both in specification and proof stages. An animation tool which supports run-time evaluation of the program statements and intermediate assertions given any user-defined input can support this process. In particular, an execution trace leading up to a failed assertion constitutes a refutation of a verification condition that requires immediate attention. As an extension to Socos, a verificion tool for invariant diagrams built on top of the PVS proof system, we have developed an execution model where program statements and assertions can be evaluated in a given program state. A

program is represented by an abstract datatype encoding the program state, together with a small-step state transition function encoding the evaluation of a single statement. This allows the program's runtime behavior to be formally inspected during verification. We also implement animation and interactive debugging support for Socos.

The thesis also explores visualization of system development in the context of model decomposition in Event-B. Decomposing a software system becomes increasingly critical as the system grows larger, since the workload on the theorem provers must be distributed effectively. Decomposition techniques have been suggested in several verification platforms to split the models into smaller units, each having fewer verification conditions and therefore imposing a lighter load on automatic theorem provers. In this work, we have investigated a refinement-based decomposition technique that makes the development process more resilient to change in specification and allows parallel development of sub-models by a team. As part of the research, we evaluated the technique on a small case study, a simplified version of a landing gear system verification presented by Boniol and Wiels, within the Event-B specification language.

Sammanfattning

Vikten av programvaras korrekthet har accentuerats då ett växande antal säkerhetskritiska system, vilka är beroende av programvaran som styr dessa, har utvecklas. En av de mer framträdande metoderna som riktar in sig på utveckling av korrekt programvara är formell verifiering. Inom formell verifiering avses med ett korrekt program ett program som uppfyller sina specifikationer och som är fritt från defekter. Medan formell verifiering teoretiskt sett kan garantera ett korrekt program med avseende på specifikationerna, har tillämpligheten av formella verifieringsmetod visat sig i praktiken vara svår och dyr. Till svar på dessa utmaningar har ett stort antal olika stödmetoder och automatiseringsverktyg föreslagits för samtliga faser från specifikationen till bevisningen av de härledda korrekthetsvillkoren. Denna avhandling behandlar praktiska verifieringsmetoder applicerade på diagrambaserade modelleringsspråk.

Medan diagrambaserade språk ofta används för kommunikation av programvarudesign (t.ex. UML) samt beteende (t.ex. tillståndsdiagram), kräver de flesta verifieringsplattformar att specifikationen kodas medelst ett textuellt specifikationsspåk eller i språket hos det underliggande logiska ramverket. Ett undantag är invariantbaserad programmering, inom vilken ett program tillsammans med dess specifikation ritas upp som sk. invariantdiagram, en typ av tillståndstransitionsdiagram annoterade med mellanliggande logiska villkor (förvillkor, eftervillkor, invarianter). Även om de tillåtna programtillstånden—sk. situationer—beskrivs diagrammatiskt är de logiska predikaten som beskriver en situations betydelse i programmets domän fortfarande skriven på konventionell textuell form. För att vidare undersöka användningen av diagram vid beskrivningen av mellanliggande villkor inom invariantbaserad programming, har vi konstruerat ett bildbaserat språk för villkor över arrayer. Vi har därefter vidareutvecklat detta språk till ett diagrambaserat domän-specifikt språk (domain-specific language, DSL) och implementerat stöd för det i verifieringsplattformen Why3. Språket låter användaren uttrycka egenskaper hos arrayer, och är baserat på Reynolds intevall- och partitionsdiagram samt inbegriper en konstruktion för mappning av array-intervall till logiska predikat.

Automatisk verifiering av ett program uppnås genom generering av korrekthetsvillkor och åtföljande bevisning av dessa. I praktiken kan full automatisering av bevis inte uppnås utom för trivial program, och även bevisning av enkla egenskaper kan kräva betydande ansträngningar både vid specifikations- och bevisfaserna. Ett animeringsverktyg som stöder exekvering av såväl programmets satser som mellanliggande villkor för godtycklig användarinput kan vara till hjälp i denna process. Särskilt ett exekveringspår som leder upp till ett falskt mellanliggande villkor utgör ett direkt vederläggande (refutation) av ett bevisvillkor, vilket kräver omedelbar uppmärksamhet från programmeraren. Som ett tillägg till Socos, ett verifieringsverktyg för invariantdiagram

baserat på bevissystemet PVS, har vi utvecklat en exekveringsmodell där programmets satser och villkor kan evalueras i ett givet programtillstånd. Ett program representeras av en abstrakt datatyp för programmets tillstånd tillsammans med en small-step transitionsfunktion för evalueringen av en enskild programsats. Detta möjliggör att ett programs exekvering formellt kan analyseras under verifieringen. Vi har också implementerat animation och interaktiv felsökning i Socos.

Avhandlingen undersöker också visualisering av systemutveckling i samband med modelluppdelning inom Event-B. Uppdelning av en systemmodell blir allt mer kritisk då ett systemet växer sig större, emedan belastningen på underliggande teorembevisare måste fördelas effektivt. Uppdelningstekniker har föreslagits inom många olika verifieringsplattformar för att dela in modellerna i mindre enheter, så att varje enhet har färre verifieringsvillkor och därmed innebär en mindre belastning på de automatiska teorembevisarna. I detta arbete har vi undersökt en refinement-baserad uppdelningsteknik som gör utvecklingsprocessen mer kapabel att hantera förändringar hos specifikationen och som tillåter parallell utveckling av delmodellerna inom ett team. Som en del av forskningen har vi utvärderat tekniken på en liten fallstudie: en förenklad modell av automationen hos ett landningsställ av Boniol and Wiels, uttryckt i Event-B-specifikationspråket.

Acknowledgments

This thesis would not have been possible without the presence of many people that I have become acquainted and conducted research with during the years of PhD. These years have been enlightening and if not a higher understanding of what surrounds you is the result of such enlightenment then what is. Therefore, I am grateful to Åbo Akademi for giving me the chance to pursue this path and to all of those whom have contributed to this thesis in one way or the other.

First I thank my advisor Marina Waldén for all formal matters involved in finalizing the dissertation. Not limited to this thesis, I have been incredibly lucky to be able to discuss the science of computing in large and in detail with Johannes Eriksson. He has been a source of inspiration through the years to seek the core when it was more convenient to see merely the surface.

I am also grateful to Philipp Rümmer and Einar Broch Johnsen who kindly accepted to review this thesis and additionally to Philipp Rümmer for accepting to act as my opponent in the public defense.

Hereby, I would like to thank Ralph-Johan Back for a sequence of seminars leading to several papers as a result of those discussions. This thesis would not have come about without that conjuncture.

Another thanks also go to my co-authors Colin Snook and Marta Olszewska. It was quite a fruitful collaboration with you within the first six months of my PhD studies.

I would like to express my gratitude to my mother Homa and my brothers Homayoon and Fereydoon for their trust in me and the utmost support that they have given me in every decision I have made. Thank you for your continuous love, encouragements, and belief in me. I would like to extend my gratitude to Azadeh and Faezeh for your continuous friendship. Thanks also to all my friends out there far or near.

And at last but not least, I would like to wholeheartedly thank my husband Johannes for your love and all the years through the path. Every adventure with you has thought me a way of life filled with wit. You have certainly showed me a way on how to outgrow the surroundings.

List of original publications

I M. Parsa, C. Snook, M. Olszewska, and M. Waldén. *Parallel Development of Event-B Systems with Agile Methods.* In M. R. Mousavi and W. Taha, editors, Proceedings of 26th Nordic Workshop on Programming Theory, NWPT '14, pages 1–3. Sweden, Halmstad University, October 29-31, 2014.

II M. Parsa, M. Waldén, and C. Snook. *An Overview of Formal Specification Languages and Tools Supporting Visualisation of System Development.* TUCS Technical Reports 1127, 2014.

III J. Eriksson, M. Parsa, and R. Back. *Proofs and Refutations in Invariant-Based Programming.* In Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings, pages 189–204, 2014.

IV J. Eriksson, M. Parsa, and R.-J. Back. *A Precise Pictorial Language for Array Invariants.* In C. A. Furia and K. Winter, editors, Integrated Formal Methods, pages 151–160, Cham, Maynooth, Ireland, September 5-7, 2018. Springer International Publishing.

V J. Eriksson and M. Parsa. *A DSL for Integer Range Reasoning: Partition, Interval and Mapping Diagrams.* In E. Komendantskaya and Y. A. Liu, editors, Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings, volume 12007 of Lecture Notes in Computer Science, pages 196–212. Springer, 2020.

Contents

Part I

Summary

Chapter 1

Introduction

While the idea of formally reasoning about computer programs originated in the fifties and software verification and the challenges it bears have been studied since the seventies [62], yet as a practice it has not reached widespread support in industry [110]. Whether a program satisfies its specification and is bug-free is undecidable in the general [103]. Hence, software verification cannot be fully automated. However, development of domain-specific techniques in this area has greatly contributed to its advancement, e.g., hardware verification [63, 45] and cryptographic protocols [6].

Formal methods aim at constructing a correct program and may be compared with software testing approaches with respect to the conditions they establish to ensure the correctness. However, the level of assurance each can provide about the correctness level of a program after applying the technique draws the distinction. The pledge of formal methods is to enable the construction of bug-free software [109]. Software testing, however, cannot guarantee the absence of bugs, but only promise to unravel a fraction of those through a systematic exercise of a program execution [21]. In the following, we discuss formal methods and its branches in brief.

Software verification means formally proving the correctness of an algorithm or a system with respect to its specification [49, 105, 53, 42]. For that, a program and its specification are described as a precise mathematical model followed by a proof in some proof system ensuring the correctness by discharging correctness conditions between the program and its formal specification. The specification is typically expressed as either a finite state machine such as timed automata [5], or as pre- and postcondition annotations added to the program code following the design-by-contract principle [72]. These two distinct approaches to specification define the major categories of *model checking* and *deductive software verification*, respectively.

Model checking [23] relies on systematic exhaustive exploration of the state space. Hence, the program is expressed as a finite state model. One representation of such state model is a state transition system. To verify the correctness of a system with model checking, the correctness properties are stated in temporal logic languages—e.g., LTL [90], CTL [22]. A model is verified if the properties hold in all execution paths of that model. If a property does not hold in an execution path, the path is returned as a counterexample. Model checking approaches have been developed to support verification of a limited set of systems and henceforth achieve a high degree of automation. They have been applied successfully to, e.g., hardware designs and embedded control software. As model checking deals with finite-state models, data

structures may require abstraction or bounding to a restricted finite scope.

Deductive software verification establishes the correctness of a system by deriving mathematical verification conditions from a program and its specification, which if true ensure the conformance of the program to the specification. In deductive verification, we can model the semantics of the target programing language without abstracting unbounded data structures—integers, lists, trees—or unbounded programming constructs—loops, recursion. This type of verification can be further categorized into the two groups of *encoding* and *embedding* [19].

Encoding verification is based on mathematical proof assistants in which both models and programs are encoded as theories. Isabelle/HOL [80] and HOL4 [100] theorem provers are supporting the inclusion of program semantics in addition to the programming constructs. Encoding verification requires a deep embedding, i.e., the program is represented by a term in the verification platform's logic and hence the program semantics are fully formalized within that logic.

Embedding verification builds up on the logical rules derived from the programming constructs. Embedding verification typically use a shallow embedding, meaning that the program is translated into verification conditions in the target logic by a process outside that logic. The specification is typically expressed as state assertions within a programming language. Shallow embeddings are generally more tractable and the typical choice for practical verification tools. However, a deep embedding can be considered more "pure". Examples of verification tools in this category are JML [67] in Java and Dafny [68] in C#.

This thesis focuses on methods derived from Hoare logic [52], which was later on translated into first-order logic in the form of the weakest precondition suggested by Dijkstra [35, 34]. In particular, it addresses diagrammatic techniques applied to Dijkstra-style *correct-by-construction* program development. Correct by construction means that the specification and the program are developed hand in hand, with the latter being verified against the former continuously throughout the process (in contrast to being verified after being written). Diagrams are widely used in software engineering, for instance to describe software design (e.g., UML [18] diagrams), or to describe its dynamic behavior (e.g., state charts [46]). However, diagrams have seen less use in formal verification, where the emphasis traditionally has been on specifications written in mathematical notation with diagrams having a documentative role rather than that of the primary object of reasoning. The diagrammatic notations addressed in this thesis are *invariant-based programming* [8, 9], a formal method where a program is structured around its invariants rather than around its statements, and *partition diagrams* [92], a diagrammatic notation for specifying properties over integer intervals. In both cases, the diagrams have precise semantics, meaning that properties can be proved about them, and they can processed by tools to extract verification conditions. Finally, the thesis also explores visualization of systems development in Event-B [1].

As the complexity and scale of programs increase, the cost of generating verification conditions and discharging them grows super-linearly [20]. Even though this complexity is inherent, automation can serve to make deductive verification practically feasible. Automation can be applied both in the generation of verification conditions from the formal specification, and in proving their correctness. The introduction of domain-specific languages [44] into verification simplifies expression of domain properties and in combination with domain-specific theories can make verification

more efficient [59]. A dynamic evaluation environment within a verification platform supports the interactive proving process [65]. Decomposition techniques decrease the verification complexity of each decomposed unit while proving the correctness of the final composition facilitates the automated verifications [3]. This thesis aims at improving verification support for the diagrammatic techniques mentioned in the previous paragraph based on these premises.

1.1 The Research Problems and Contributions

Next, we concretize the software verification research problems that are of relevance to this thesis. We outline the research questions we have focused on and describe the methods with which we have addressed them.

Visualization of system decomposition in Event-B. It is generally accepted that software development benefits from iterative approaches where each iteration is quite short—on the magnitude of days or weeks—in order to support a feedback-driven self-correcting workflow that tolerates changes to the specification during the course of the project [104]. Here a challenge faced by practitioners of formal methods is how to avoid imposing a waterfall-like process due to the strict sequentiality of the specification, implementation and verification phases. One approach addressing this challenge is decomposition, i.e., the breaking down of a formal model into sub-models, allowing for multiple specification, implementation and verification sequences to occur in parallel, the results to be eventually recomposed into a complete verified software. The research question we pose is how to identify separable aspects of the model that can serve as the basis for such a decomposition, allow parallel development, while also ensuring that the final composition is seamless. To address the question, we have suggested a decomposition technique that allows agile and iterative development of multiple abstractions of an Event-B system. The technique centers around first identifying abstractions in the software that can be refined in parallel, modeling each abstraction as a state machine in the Rodin [2] platform, refining each abstraction into an executable machine through a chain of refinements, and finally merging the final refinements of each abstraction into a complete system. We demonstrate this technique in a case study in Paper I. The literature review in Paper II on diagrammatic approaches to verification positions this technique in the broader context of visual tool-based formal development methods.

Combining static and runtime verification. From a software engineering perspective, developing formally correct programs is more challenging than conventional test-driven development. It demands significant rigor during the development phases, and involves the use of specialized tools such as automatic theorem provers. Firstly, verification requires formally specifying the software using a mathematical modeling language; secondly, specification and implementation are combined into verification conditions; thirdly, discharging the verification conditions requires both interactive and fully automated theorem provers to discover the proofs and maintain them. As in test-based development, these phases do not complete in sequence but rather previous phases are iteratively re-visited. For instance, an inadvertent underspecification may

not become evident until we try to build the proof of a verification condition derived from it and note a missing critical assumption that requires amending the specification and re-checking all proofs derived from it. Hence, practicable verification tools need to accommodate non-linear workflows. Experience reports from the application of formal methods in industry case studies, such as [4], underscore the importance of integrating existing semi-formal diagrammatic methods (UML) into the verification workflow, as well as debuggable specifications, both when building the specification and when ensuring its consistency. The question we are addressing is how a combination of static and runtime verification can help in identifying errors early in the verification phase, thereby saving time wasted on attempting to prove invalid verification conditions. Our extension to the verification tool Socos [38] described in Paper III, allows the developer to exercise the runtime behavior of a program while verifying it. Programs are executed within the theorem prover framework, meaning that a failed test case is a logically sound refutation of the corresponding verification condition. Such refutations safeguard the developer from futile attempts at proving a false verification condition.

Diagrams as formal specification languages. While diagrams are extensively used by software developers and have proven to be an effective means of communicating system properties and intent, they are typically considered informal. Even standardized diagrammatic languages, such as UML, tend to lack the expressiveness, precise semantics and good tool support that makes them useful throughout the software development process. Lack of expressiveness and semantic precision limits their usefulness to planning, documentation and presentation. Inadequate tool support—e.g., offering only syntactic analysis—and insufficient semantics to express precise formal specifications prevents using the diagram as a primary code artifact as the result derived from it must be semantically enriched using some other (usually textual) language. Due to these limiting factors, the diagram essentially becomes a throw-away as it goes out of sync with the program code. The stance taken in this work is instead to consider diagrams as primary software artifacts, and the question asked is how to design diagrammatic languages that are semantically precise enough for formal specification and verification, while at the same time retaining the clarity and legibility that a diagram provides. We explore this question by combining two formal diagrammatic languages, invariant diagrams and partition diagrams, into a single diagrammatic language suitable for formal specification and verification of programs over arrays. Invariant diagrams is the language used in the invariant-based programming approach, where invariants are defined before the program statements, while partition diagrams is an in-line notation allowing many common array properties to be stated pictorially. The combined diagrammatic language and its semantics is introduced in Paper IV, and we also evaluate the language by a number of case studies.

Diagrams as domain-specific languages (DSLs). Formal verification requires building a model of the intended behavior of a system under verification. A successful verification guarantees only that the runtime behavior of the program is consistent with the model. Therefore, the model must on one hand be abstract enough to be admissible as an axiomatic description of the system, while on the other hand must contain enough detail to make possible a formal machine-checkable proof between the model and

the program. Constructing and validating the model is a major challenge of formal methods, as general-purpose modeling languages require significant expertise from the user. In contrast, domain-specific languages (DSLs) allow succinct, comprehensible and precise specification, but within a narrowly-scoped domain. The questions we are interested in here are whether the integration of a diagrammatic DSL into an existing (text-based) software verification platform is feasible, and whether such a DSL can facilitate writing specifications in its domain. As part of our work to address these questions, we have implemented support for the language introduced in Paper IV in the Why3 verification platform [16]. The implementation extends the Why3 language lexically by modifying its parser, and semantically by a number of Why3 theories embodying the DSL. Paper V describes the theoretical and technical aspects of the work, and gives a number of verified examples illustrating the use of the DSL.

1.2 Overview of the Thesis

The remainder of the thesis is structured as follows. In Chapter 2, we overview the major verification concepts relevant to the thesis, including deductive verification principles, automatic theorem proving, as well as some related techniques. Chapter 3 provides an introduction to the verification platform Socos and invariant-based programming. Further, we present a detailed view of Socos architecture, verification condition generation, the underlying theorem prover PVS [82] and the PVS ground evaluator [27] utilized in building the evaluation mechanism. We demonstrate the Rodin verification platform and model decomposition techniques in Chapter 4. Chapter 5 describes the Why3 platform, including its specification language, theory extension and proof transformations. Chapter 6 summarizes each of the papers included in this thesis. Chapter 7 concludes the thesis with a discussion and outlines current limitations to be addressed in future work.

Chapter 2

Verification Approaches

In this chapter, we overview verification approaches relevant to the thesis. The emphasis is on deductive verification and automatic theorem proving, with a brief coverage of other significant techniques.

2.1 Deductive Software Verification

Turing [105] was aware of the need for proofs of program correctness and provided early stage solutions already in the forties. He raised the question in the paper that

> "How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows."

Floyd [43] initiated an important step towards deductive verification by suggesting the inclusion of assertions in programs. The program is abstracted into a control flow graph with inductive assertions. To prove the correctness, each path through the flow graph should be proven to maintain the inductive assertions over the program variables.

Following this, Hoare [52] presented a triple rule for *partial correctness* assuming the precondition P, program S and the postcondition Q:

$$\vdash \{P\}\ S\ \{Q\}$$

The Hoare triple is valid if for any given state satisfying the precondition P, executing the program S—assuming that it terminates—results in a state that satisfies Q. Consider next the following language of simple sequential programs:

$$S ::= \texttt{SKIP} \mid x := E \mid S_1; S_2 \mid \texttt{IF}\, B\, \texttt{THEN}\, S_1\, \texttt{ELSE}\, S_2 \mid \texttt{WHILE}\, B\, \texttt{DO}\, S$$

The Hoare inference rules, defined below, allow deriving the verification conditions for any statement expressed in this language:

$$\frac{}{\{P\}\ \texttt{SKIP}\ \{P\}} \tag{2.1}$$

$$\frac{}{\{Q[E/x]\}\ x := E\ \{Q\}} \tag{2.2}$$

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \tag{2.3}$$

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{IF}\, B\, \texttt{THEN}\, S_1\, \texttt{ELSE}\, S_2\ \{Q\}} \tag{2.4}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \texttt{WHILE}\, B\, \texttt{DO}\, S\ \{P \wedge \neg B\}} \tag{2.5}$$

$$\frac{\{P\}\ S\ \{Q'\} \quad Q' \to Q}{\{P\}\ S\ \{Q\}} \tag{2.6}$$

$$\frac{P \to P' \quad \{P'\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}} \tag{2.7}$$

In the inference rule 2.2, $Q[E/x]$ stands for the syntactic substitution of all free occurrences of the variable x with the expression E throughout the formula Q. Rule 2.3 allows dividing a sequential composition proof into separate lemmas for each statement through the introduction of a mid-condition Q, while rule 2.4 captures the two mutually exclusive branches of the `IF-ELSE` statement. Rule 2.5 states that proving the correctness of a loop with condition B and repeating statement S reduces to proving that the statement S maintains the *loop invariant* P while B is true. After termination of the loop, condition B is false while invariant P remains true. Rule 2.6 states that a postcondition can be weakened, while rule 2.7 states that a precondition can be strengthened. The rules define a verification method for programs: if a complete derivation tree can be constructed for a Hoare triple, the triple is valid.

When constructing a derivation tree to prove the triple, there may be several choices in the forward reasoning which is not desirable in automated proving. To address this, Dijkstra [34, 35] proposed a backward search by computing a precondition for a program S and desired postcondition Q. In the *weakest precondition calculus*, a program can be semantically represented by a predicate transformer. A predicate transformer is a function from predicates to predicates, or equivalently, from subsets of the state-space to subsets of the state-space. The *weakest liberal precondition* function

$wlp(S,Q)$ is a predicate transformer that returns the weakest predicate from which a statement S when executed either terminates in a state satisfying Q or does not terminate. To establish the validity of a triple $\vdash \{P\}\ S\ \{Q\}$ using the weakest liberal precondition the following implication should be proven:

$$P \rightarrow wlp(S,Q)$$

The first four Hoare rules presented earlier corresponds to the following rules of weakest liberal precondition defined recursively over the language structure:

$$wlp(\texttt{SKIP}, Q) = Q \tag{2.8}$$

$$wlp(x := E, Q) = Q[E/x] \tag{2.9}$$

$$wlp(S_1; S_2, Q) = wlp(S_1, wlp(S_2, Q)) \tag{2.10}$$

$$\begin{aligned} wlp(\texttt{IF}\, B\, \texttt{THEN}\, S_1\, \texttt{ELSE}\, S_2, Q) = {} & (B \rightarrow wlp(S_1, Q)) \\ & \wedge (\neg B \rightarrow wlp(S_2, Q)) \end{aligned} \tag{2.11}$$

Since the verification of the `WHILE` statement requires the discovery of a loop invariant—typically by the programmer—we first modify the language by adding an invariant annotation to the `WHILE` statement:

$$S ::= \ldots \mid \texttt{WHILE}\, B\, \texttt{INV}\, I\, \texttt{DO}\, S$$

The weakest liberal precondition for the `WHILE` statement can then be computed as follows:

$$\begin{aligned} & wlp(\texttt{WHILE}\, B\, \texttt{INV}\, I\, \texttt{DO}\, S,\, Q) = I \\ & \quad \wedge \quad \forall y.(B \wedge I \rightarrow wlp(S, I))[y/x] \\ & \quad \wedge \quad \forall y.(\neg B \wedge I \rightarrow Q)[y/x] \end{aligned} \tag{2.12}$$

Rule 2.12 captures that the iterated statement S of the loop must preserve the invariant I while the postcondition Q must be established upon termination. Here x stands for the variable(s) modified by S and y is a fresh logical variable that does not occur in S, I or B.

Hoare triples and the *wlp* transformer capture partial correctness: the program when executed in a state satisfying P either terminates satisfying Q or loops forever. *Total correctness*, a stronger condition, means that the program additionally always

terminates satisfying Q. The *weakest precondition* transformer *wp* can be defined as *wlp* restricted to the preconditions from which the program terminates:

$$wp(S,Q) = wlp(S,Q) \wedge terminates(S)$$

The rules for the *wp* transformer differ from the rules of the *wlp* transformer only for the `WHILE` statement. Proving termination of this statement requires establishing an upper bound for the number of iterations by introducing a *well-founded relation* $<$ on the state space. A *variant* is a function from the program state, selected so that its value decreases by each iteration with respect to $<$. By proving that $<$ holds between the variant at the start and the variant at the end of the loop body, the number of iterations must be finite, since a well-founded relation contains no infinite descending sequences. After extending the `WHILE` statement with a variant annotation

$$S ::= \ldots \mid \texttt{WHILE}\, B\, \texttt{INV}\, I\, \texttt{VAR}\, V\, \texttt{DO}\, S$$

the weakest precondition for the `WHILE` statement can then be computed as follows:

$$\begin{aligned} & wp(\texttt{WHILE}\, B\, \texttt{INV}\, I\, \texttt{VAR}\, V\, \texttt{DO}\, S,\, Q) = I \\ & \quad \wedge \quad \forall y, z.(B \wedge I \wedge z = V \rightarrow wp(S, I \wedge V < z))[y/x] \\ & \quad \wedge \quad \forall y.(\neg B \wedge I \rightarrow Q)[y/x] \end{aligned} \tag{2.13}$$

The additional proof obligation in the second conjunct ensures that the variant is decreased by S with respect to the well-founded relation $<$ ensures termination and consequently the total correctness of the `WHILE` statement (here y and z are fresh logical variables).

Applying software verification in practice requires automating both the generation and discharging of proof obligations. The latter task is typically delegated to specialized automatic theorem provers.

2.2 Theorem Proving

Formal proofs as opposed to proofs written in a natural language require rigid symbolic logical reasoning. A traditional proof is written in natural language, validated by peer reviews, and is meant to convey a message by providing an intuition of which steps the proof consists of. In contrast, a purely formal proof is written in a symbolic language and the proof steps applied follow deductive proof reasoning to ground each sentence into an axiom or a previously proven sentence [47]. Most conventional proof methods can be reduced to a small set of axioms and rules.

Proving the validity of verification conditions in deductive software verification can be automated by assistance of theorem provers. Most theorem provers incorporate algorithmic search mechanisms to apply the proof rules and deduce conclusions.

The early work in computer-assisted proving centered around fully automated approaches to enable machines to prove assertions automatically and without human interaction [29, 30, 108, 77]. However, the limitations of fully automated provers in proving non-trivial theorems initiated the movement towards interactive provers which

support human-guided proofs. To name but a few of the major interactive provers, we can refer to Isabelle [80], PVS [82], ACL2 [64], HOL4 [100] and Coq [15].

One approach to assisting the automated provers is by constructing proofs using tactics [73]. A tactic or a proof strategy is a set of pre-defined instructions that are applied on a goal to construct a proof tree. In general, no proof strategy may be guaranteed to prove or disprove an arbitrary formula in a given formal system (as the underlying logic, usually some type of first- or higher-order logic, is not decidable). However, some classes of formulas are decidable and user-defined strategies can reduce the search effort that is otherwise required by the automated provers.

Theorem provers construct a proof tree in order to prove a goal with the goal being the root node of the tree. Each node in the tree constitute a proof goal that is a sequent consisting of a sequence of formulas called antecedents and a sequence of formulas named consequents. Each proof command either evaluates the current sequent to true and terminates the branch, or it introduces new child sequents to the branch and continues by proving each of these. Inference rules are applied backwards; that is, applying an inference rule R where

$$\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta} R$$

on a proof node $\Gamma \vdash \Delta$ creates n new leaves each containing a sequent $\Gamma_i \vdash \Delta_i$. An example of a basic inference rule is the *cut rule*, which introduces a case-split into a proof where A is introduced as an antecedent along one branch and as a consequent along the other:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta} cut$$

Another fundamental set of inference rules captures the natural deduction rules on logical connectives and quantifiers, such as introduction and elimination. These rules constitute the base mechanism for applying logical inferences when proving a sequent. In addition to codifying the basic inference rules and ensuring that each proof step is a valid inference, an automatic theorem prover also automates, among others, the following tasks:

resolution to introduce a new clause implied by two clauses containing complementary literals.

unification to match two terms by providing all substitution of variables under which the two terms are equal.

substitution to automatically find instantiations for quantified variables.

rewriting by using a result proved in another sequent as a lemma.

induction to automatically apply the induction lemma to a quantified proposition, branching on hypothesis and base cases.

Finally, decision procedures for decidable fragments of first-order logic, are incorporated to increase the automation further. These decision procedures are typically implemented by specialized, highly optimized solvers called *SMT solvers*.

2.2.1 SMT Solvers

The Boolean satisfiability problem (SAT) is defined as finding the values for the variables in a propositional formula for which the formula evaluates to true. If there is at least one such assignment of values for the variables to evaluate all clauses of the formula to true, the formula is stated as satisfiable. SAT problem is known to be NP-complete [25], however practically efficient algorithms have been developed known as SAT solvers.

SMT solvers extend SAT solvers with additional theories. The additional theories extend the decision procedures for the formula to include arrays, real and integer arithmetic, and uninterpreted functions. Some of the core algorithms underlying SAT solvers are Davis-Putnam-Logemann-Loveland (DPLL) [29] and Nelson-Oppen algorithm for combining decision procedures [78]. Examples of widely used SMT solvers are Z3 [31], CVC4 [12] and Yices [36]

By verifying that the negation of a proposition is unsatisfiable, a theorem prover can incorporate SMT solvers as automatic endgame provers. For an SMT solver to reason about such propositions, a sound translation of the theorem prover logic to the logic of the SMT solver is required. To facilitate such translation, the SMT solving community has introduced a standard interchange language known as SMT-LIB [13].

2.3 Other Techniques

Except the verification techniques described in previous sections, other approaches have been developed in order to analyze the correctness of programs. Some of the techniques are presented in this section.

2.3.1 Extended Reasoning on Hoare Rules

The main challenge of the Hoare approach is capturing the semantics of modern programming languages, e.g., Java. This is due to the complex programming languages constructs such as building a new object, writing to a field of an object, and shared mutable data structures. For proving the correctness using Hoare rules, one approach is to define meta-level transformations describing the behavior of an object rather than state transformations [96]. Other techniques such as separation logic have been developed in order to address the verification of shared mutable data structures [95].

Concurrency is integral to many applications but verifying the correctness of concurrent programs is difficult. The basic rules of Hoare do not support reasoning about parallel programs as it is not sufficient to add assertions only to the initial and final states to prove the correctness of the program due to interleaving of concurrent processes [81]. Depending on the semantics of the concurrency, various approaches have been suggested. Process calculi, e.g., CSP [54], provide an algebraic language for describing interactions and communications between a collection of independent agents and processes. The other category of logical expressions on concurrent programs is to rationalize on shared variables. Rely-guarantee method is one such approach. The rely-guarantee method models the atomic actions describing the interference of programs which allows other threads to modify the state of a thread, provided that they respect the rely constraints [111].

2.3.2 Model Checking

Model checking [23] refers to a family of verification techniques that exhaustively explore the state space of a finite-state model in order to assert properties of the model. Example of such properties are correctness and liveness. Linear temporal logic [90] and/or branching temporal logic [22] are used to describe the properties of a model. Linear temporal logic defines behavior along a single execution path, while branching temporal logic allows quantifying over all paths from a state. A well-known issue in model checking is the state-space explosion problem. Various solutions have been applied to reduce the states of the model in computation such as symbolic model checking with efficient data structures such as binary decision diagrams (BDDs). Infinite data structures require abstraction or bounding to a restricted finite scope before they can be reasoned about. Depending on the specification to be proven and the source of infinity, the finiteness can be introduced, for example, by defining explicit bounds on the input parameters or the maximum number of times to unroll a loop.

2.3.3 Static Analysis

Static analysis techniques allow reasoning about the behavior of programs without executing them [79]. An abstraction of the program code, for instance in a form of a directed graph, is built and analyzed to check if desired properties hold. The violation of a property in the abstract model may indicate the violation of the property in the original program. Analyzed properties range from adherence to coding standards (linting) to ensuring that assertions are true (extended static checking). *Control flow analysis* can detect poorly structured code, e.g., dead code, and multiple exits from a loop statement. *Data flow analysis* can inspect the flow of the data to ensure a sound programming practice, e.g., that variables are read before declaration. *Program slicing* is a technique to reduce the complexity by scoping down the analysis to only those program statements that are of concern for the analysis, e.g., those which modify a subset of variables. Frama-C [28] and F-Soft [56] are two examples of extended static checkers for the C programming language. These tools can detect many common violations, such as dereferencing a null pointer or indexing outside the bounds of an array. However, extended static checkers are often incomplete and may produce both false positives and negatives: either a violation is reported which cannot occur in the actual program, or a violation stays undetected.

2.3.4 Symbolic Execution

Symbolic execution is a form of forward reasoning by symbolizing the variables of a program as constants and the values of program variables as expressions [55]. The program paths construct the assumption on the states (path conditions) by accumulating constraints on the symbolic constants that are to be satisfied when executing that path. This approach is challenging because of an exponential number of symbolic paths. The symbolic execution may fork a new state at each branch of a program. Having all the branches to explore increases both the memory requirement and execution time. Though, this technique may be used less ambitiously in practice by trading soundness for performance (e.g., bounding the number of iterations). Other verification techniques

have employed symbolic execution such as model checking. Abstract interpretation [26] statically asserts properties of a program by abstracting its concrete semantics so that the program can be feasibly evaluated with respect to the properties.

2.3.5 Contract-Based Verification

Contract-based verification approaches the decomposition of proof obligations in a software system through the use of provided or synthesized contracts. A *contract* is software specification unit that stipulates an *assumption* that the component adhering to the contract may rely on, and a *guarantee* that it must uphold [48]. For example, a pre- and postcondition specification of a procedure constitutes a contract between the caller of the procedure and the procedure implementation: the caller guarantees the precondition while assuming the postcondition, and the procedure implementation assumes the precondition while guaranteeing the postcondition. Generally, contracts and contract-based reasoning can be applied to many types of software units, e.g., classes and modules, in order to decompose proof obligations when verifying a larger system.

Chapter 3

The Socos Environment

Modern practical software verification methods build on the *constructive approach* by Dijkstra [35, 33], where a program is developed jointly with its specification and proof. By constructing program and proof together, correctness concerns can be addressed in the design, for instance in the separation of a program unit into procedures with pre- and postcondition contracts. In contrast a posteriori verification—verifying a program after it has been written—quickly becomes intractable as programs grow larger. In the original constructive approach, the smallest unit of verification is the procedure: a procedure with parameters and return values is specified with pre- and postconditions, then implemented, and finally the code is proved to satisfy the contract. Except for trivial programs, the last step requires annotating the code with intermediate assertions and loop invariants, in order to divide the proof into lemmas of manageable complexity.

Invariant-based programming (IBP) is an approach for verifying programs where the specification and invariants are written before the actual program statements [94, 106, 91, 8]. IBP makes correctness constraints the primary program building blocks by requiring assertions and loop invariants to be written before the statements. This means that assertions are no longer annotations added to existing code, but instead determine the structure of the program. IBP introduces a new type of diagram called an *invariant diagram*, consisting of *situations* partitioning the total state space of the program. A *transition graph* through the situations replaces the traditional control flow constructs of imperative programming (if- and while-statements). Situations can be likened to the nodes of a flowchart or state chart, but in addition to a label they also introduce a state predicate that should hold when the program flow reaches a situation, and they have a set-theoretic nesting semantics similar to that of Venn diagrams. Only after the situation structure has been determined, are the statements—in the form of guarded transitions between the situations—added and verified one by one. A benefit of this approach is that the program and its proof can be developed as a single entity, and the lemma for each statement is typically small enough to be discharged by an automatic theorem prover.

The theoretical foundation for IBP is quite well developed [7], but as a practical formal method IBP has only been studied in small case studies [8] and as a teaching tool [9]. The Socos [38] tool is an editor and prover front-end built from the ground up to support IBP. It consists of a graphical editor for invariant diagrams, a verification condition extractor and translator targeting the PVS specification and verification system [82]. This allows using PVS and its associated provers to automatically or

interactively discharge conditions. Part of this thesis centers around the integration of dynamic (runtime) evaluation into the Socos tool. Our extension to Socos implements the following:

Operational semantics: the extension produces a PVS theory containing a datatype state representation, where each labeled constructor corresponds to a location in the diagram, and a function that evaluates one step in the diagram.

Efficient execution: the extension employs the PVS ground evaluator [27], allowing efficient compiled evaluation of expressions in statements.

Animation: the extension supports graphical debugging by allowing the developer to step through the execution of a program statement by statement, visualizing the program state in each step.

The execution mechanism has been designed from the ground up to support the IBP workflow. Practical IBP is iterative, meaning that the programmer typically goes through several unsuccessful verification attempts of intermediate incomplete versions of the program, before reaching the final verified program. While correctness cannot be achieved by testing alone, tests and proofs are complementary during this process. A failed execution serves as a counterexample to, or in other words, a *refutation* of, the correctness of the program under development. In the early stages of developing a program, testing combined with runtime assertion evaluation effectively identifies trivially incorrect invariants and statements, whereas invoking an automatic theorem prover to discover the same error would be less efficient. At later stages, an unsuccessful proof attempt may indicate underspecification or more subtle bugs that would have beed missed by testing. This means that the primary use mode of the execution mechanism is to support verification of programs in their early stages. The PVS ground evaluator allowed us to have both the verification and runtime mechanisms tightly integrated within the same verification framework. In the remainder of this chapter, we give a brief overview of the IBP notation and verification methodology, as well as the PVS verification system and its application in Socos.

3.1 Invariant Diagrams

Invariant diagrams consist of rounded rectangles called situations, which are connected by transition arrows. Situations represent the subsets of the program state space satisfying the predicates written inside the situation. Transitions represent *guarded statements*: the *guard* is the conditions under which the transition becomes enabled, triggering the rest of its statement to update the state of the program. Socos invariant diagrams are structured into procedures, where the body of each procedure is comprised of a situation and transition graph. A schematic example of a Socos diagram is given in Figure 3.1.

The outer rectangle declares the procedure: its name, parameters and local variables. The precondition is the situation without incoming transitions and the postconditions are the situations without outgoing transitions. A situation may be labeled, written in the upper left corner of the rectangle. The procedure body in this case contains four situations: the precondition PRE, the intermediate situation LOOP, and the two postconditions POST_1 and POST_2. A procedure always declares a single precondition, but may declare one or more postconditions. There may be any number of intermediate

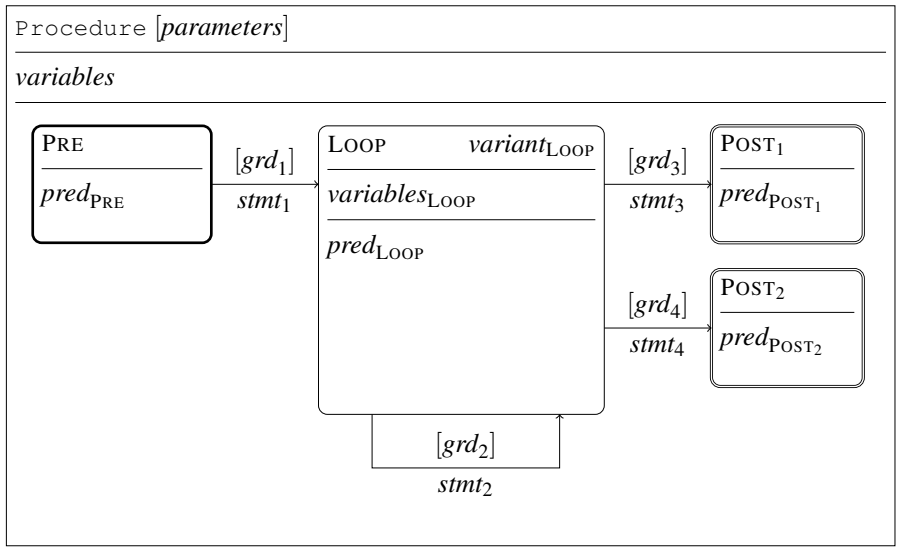


Figure 3.1: Invariant diagram

situations, and there are no restrictions on transitions between them.

A situation may contain a predicate stating what should be true when the program flow enters the situation. The predicate of the precondition should be established by the caller of the procedure. An intermediate situation may include transitions back to itself, in which case the predicate serves as the loop invariant, i.e., the predicate should be established before entering the situation and should be maintained by the loop transition back to the situation. The variant in the upper right corner serves as the termination condition of a loop situation. The variant should be chosen from a well-ordered set and be decreased by every transition that re-enters the situation. The predicate of a postcondition should be established by the procedure upon termination. In the general invariant diagram, intermediate situations may be nested, meaning that the inner situations inherit the invariants of the outer situations. In other words, the predicate of an inner situation is the conjunction of the predicates in the situation itself and the predicates of all enclosing situations.

3.2 Verification Conditions of Invariant Diagrams

A diagram is totally correct if it is consistent, live, and terminating. The *consistency* of a transition from situation A to situation B with guard grd and statement $stmt$ is established by proving the implication

$$pred_A \wedge grd \rightarrow wp(stmt, pred_B)$$

Consistency of the invariant diagram means that this condition is true for each transition. The complete consistency condition for the diagram in Figure 3.1 is then the

conjunction:

$$\begin{aligned}&(pred_{\text{PRE}} \wedge grd_1 \rightarrow wp(stmt_1, pred_{\text{LOOP}}))\\ \wedge\ &(pred_{\text{LOOP}} \wedge grd_2 \rightarrow wp(stmt_2, pred_{\text{LOOP}}))\\ \wedge\ &(pred_{\text{LOOP}} \wedge grd_3 \rightarrow wp(stmt_3, pred_{\text{POST}_1}))\\ \wedge\ &(pred_{\text{LOOP}} \wedge grd_4 \rightarrow wp(stmt_4, pred_{\text{POST}_2}))\end{aligned}$$

The *termination* of a diagram is ensured by proving that no infinite loop exists. Termination can be established by proving that each cycle of transitions through a situation decreases a given variant function, which remains bounded from below. Socos supports variants that are functions from the program state to the natural numbers. The termination condition for the diagram in Figure 3.1 then becomes:

$$pred_{\text{LOOP}} \wedge grd_2 \wedge v_0 = variant_{\text{LOOP}} \rightarrow wp(stmt_2, 0 \leq variant_{\text{LOOP}} < v_0)$$

In this formula, v_0 records the value of the variant before entering the transition, and the postcondition states that the updated variant remains greater than zero while being strictly smaller than its original value v_0.

Liveness means that in all situations except those where the program is intended to terminate—i.e., the postconditions—at least one outgoing transition is enabled. For a single situation, liveness is established by proving that the situation predicate implies the disjunction of the outgoing transitions' guards. The liveness condition of the diagram in Figure 3.1 then becomes:

$$\begin{aligned}&(pred_{\text{PRE}} \rightarrow grd_1)\\ \wedge\ &(pred_{\text{LOOP}} \rightarrow grd_2 \vee grd_3 \vee grd_4)\end{aligned}$$

We note that the consistency condition for each transition depends only on the source situation, the guard, the statement, and the target situation. This means that once the situation structure is completed, transitions can be added and verified one by one. Liveness and termination, on the other hand, generally require a complete transition structure and are therefore preferably verified at the end. Finally, we note that transitions in invariant diagrams may contain intermediate branching nodes (that are not situations) from which multiple guarded transitions further extend to form a tree, where the edges are labeled with transition-statement pairs and the leaves are situations (an example of such a diagram is given later in this chapter). Each path through a transition tree can be verified independently for consistency and termination, while liveness must be established at each branching node.

3.3 Verifying Invariant Diagrams in Socos

The Socos tool was developed to support invariant-based programming by providing a frontend editor for drawing the invariant diagram, attaching background theories and interacting with the PVS prover and a backend to generate and discharge the verification condition generated from the diagram. The frontend is distributed as a plug-in to the Eclipse IDE. A user interacts with the Socos environment through the diagram editor, which is available as an additional file editor in the Eclipse workbench

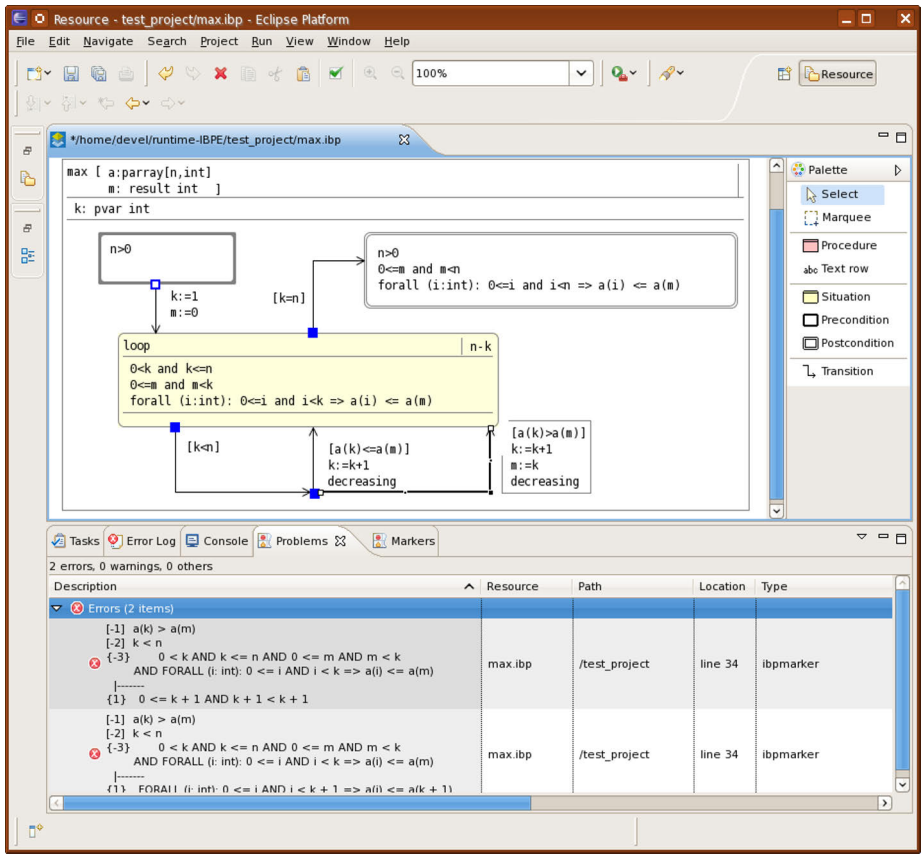


Figure 3.2: Socos Environment with diagram editor and unproved verification conditions

once the user has installed the Socos plug-in. Figure 3.2 shows a screen capture of the diagram editor with an invariant diagram of a program (`max`) for finding the maximal element in a (non-empty) array of integers being verified.

The Socos frontend is implemented in Java, while the backend is implemented in Python and invokes the PVS [82, 86] theorem prover and Yices [36] SMT solver as an endgame prover. The specification language of Socos is higher order logic [69] and it shares the expression syntax with PVS.

By the click of a button, Socos generates verification conditions from an invariant diagram and invokes the PVS theorem prover to automatically discharge them. Only the conditions that were not proved automatically are shown to the user. Figure 3.2 shows two verification conditions that were not proved automatically. Upon closer inspection, we can see that these are actually false due to an error in the program; the statements assigning new values to the program variables `m` and `k` in the selected transition appear in the wrong order. After these statements are swapped, and verification is invoked again, all verification conditions of the program are proved automatically.

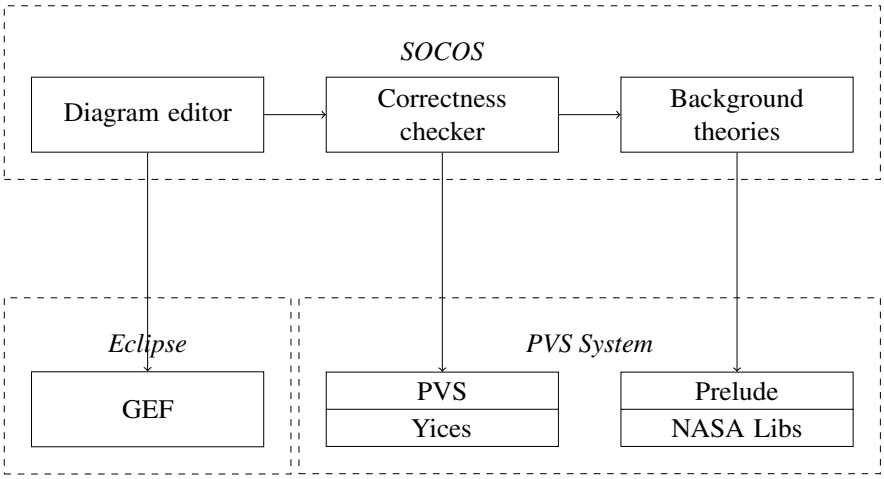


Figure 3.3: Architecture of the Socos environment

3.4 Software Architecture of Socos

Implementation-wise, Socos consists of an IDE integration component that communicates with a backend verification condition extractor and theorem prover driver [38]. The high-level architecture is shown in Figure 3.3. The *diagram editor* is distributed as an Eclipse plug-in, implemented in Java and utilizes the GEF (Graphical Editing Framework), part of the Eclipse Modeling Framework [37], as well as other core libraries of the Eclipse platform to realize the graphical user interface.

The diagram editor integrates with the *correctness checker*, which handles the generation of verification conditions and the communication with the PVS verification backend. The correctness checker can either be deployed locally, in which case it communicates with the diagram editor over a Unix pipe, or remotely over HTTP. When the verify button is clicked in the diagram editor, the correctness checker syntactically and semantically analyzes the diagram, and generates a PVS theory whose validity ensures the correctness of the diagram. This generated theory contains all declarations given in the diagram (context), as well as a lemma for each verification condition (consistency, termination and liveness). Moreover, a PVS proof script is generated for every condition to branch its proof tree so that there is one leaf for each of the predicates in the target situation. On each leaf, an endgame strategy based on the Yices SMT solver is applied. After the theory generation step, Socos launches PVS to run the proof scripts, and parse the result. If the proof fails for a leaf, the unproved sequent is sent back to the frontend and shown to the user. The user can choose to open the lemma in PVS and use its interactive prover to inspect or attempt to prove the lemma using other strategies.

To support the verification of more complex programs, the user can extend Socos with domain-specific PVS *background theories*. The PVS prelude [85] and Nasa Langley libs [76], available as part of the PVS distribution, provide a base for writing custom background theories. These theories can be imported into the diagram, and all definitions and lemmas become available in the diagram as well as to the correctness

checker during verification.

3.5 PVS

PVS (Prototype Verification System) [82] is an open-source[1] verification system consisting of a specification language based on higher order logic and an interactive theorem prover. The semantics of PVS follow set theory where types are interpreted as sets. Higher order logic [69] imposes typing on the language in contrast to set theory. Typing in PVS is based on the simply typed lambda calculus and each expression has an associated type [84].

PVS is implemented in Common Lisp and can be invoked both standalone and as a backend to other verification frameworks. To utilize the PVS theorem prover as a backend, PVS can be invoked via the command line. As a frontend, PVS has an interactive Lisp process with an Emacs editor [88]. The PVS interactive prover can be operated in the Emacs editor.

Specifications are built from theories in PVS. Each theory consists of a sequence of declarations which may introduce types, constants, variables, axioms and formulas. PVS theories are either predefined or user-defined. A library of predefined theories, referred to as the prelude [85], including Boolean operators, equality, real, integer and natural number types, is available immediately for use in user-defined theories. User-defined theories should be imported explicitly in the context of use.

A simplified description of the theory syntax of PVS is given below (for a complete definition, see the PVS Language Reference [87]):

theory_id [⟨ *param_id* **:** **TYPE** | **TYPE**+ | *type* ⟩,...] **:** **THEORY**
BEGIN
⟨ **IMPORTING** *theory_id* [*expression*,...] ⟩,...

⟨ *type_id*: **TYPE** | **TYPE**+ ⟩,...

⟨ *var_id*: **VAR** *type* ⟩,...

⟨ *const_id* (⟨ *param_id* **:** *type* ⟩,...): *type* = *expression* ⟩,...

⟨ *const_id* (⟨ *param_id* **:** *type* ⟩,...): **RECURSIVE** *type* =
expression **MEASURE BY** *expression* ⟩,...

⟨ *formula_id*: **AXIOM** | **THEOREM** | **LEMMA** *formula* ⟩,...

END *theory_id*

A theory is named and can be parametrized with constants and types for instantiation when imported into another theory via the **IMPORTING** declaration. Type expressions in PVS declare a set of values that can possibly be infinite and type constructors include function, record, predicate subtype expressions and datatypes. As subtype predicates can be general higher-order formulas, type-correctness conditions (TCCs) generated by PVS may require interactive proving to be discharged. A named type

[1] https://github.com/SRI-CSL/PVS

can be uninterpreted—if declared with **TYPE**+ the type is required to be non-empty. The function type in PVS is expressed with explicit domain and range type. Functions in PVS are higher order and therefore the domain and range can also be any type expression. Variable declarations associate identifiers with types within a theory, while constant declarations introduce constants and functions visible to importing theories. The **RECURSIVE** keyword allows defining a recursive constants with a **MEASURE** expression to ensure well-definedness (PVS generates a termination TCC based on this expression). Formula declarations introduce formulas to be proved or invoked (as lemmas) in the prover. They are boolean expressions declared as either axioms (**AXIOM**) or theorems (**THEOREM** or **LEMMA**). They differ in that PVS expects proofs for the latter.

Parametric abstract datatypes can be declared in PVS with the **DATATYPE** keyword [83]. For example, a data type representing a binary tree containing elements of the type `T` can be declared as follows:

```
Tree [T: TYPE+] : DATATYPE
BEGIN
  Empty: Empty?
  Node (val: T, left: Tree, right: Tree): Node?
END Tree
```

The datatype has two constructors, `Empty` and `Node` for empty and non-empty subtrees, respectively; `Empty?` and `Node?` are the corresponding recognizer predicates. PVS generates TCCs on the abstract datatypes to ensure that the constructors and the recognizers of a datatype are pairwise distinct and at least one constructor is non-recursive. PVS also provides a **CASES** expression for matching datatype constructors by name.

3.6 Translating Socos Programs to PVS

Next we describe by example how a Socos diagram is translated into a PVS theory containing verification conditions. Figure 3.4 shows an invariant diagram describing a binary search tree operation of determining the presence of the value `x` in `tree`. The diagram makes use of the `Tree` datatype introduced in the previous section, as well as the functions `bst`, `has` and `depth` defined in the PVS background theories shown in Listing 3.1. We note that the LOOP situation is nested inside PRE, making the effective loop invariant:

```
bst(tree)∧bst(cur)∧has(cur,x)=has(tree,x)
```

The PVS theory containing the verification conditions generated from Figure 3.4 is shown in Listing 3.2. The verification conditions are generated according to the rules given in Section 3.2. We note that six lemmas are generated:

- `trs_Pre_to_Loop` for consistency of the transition from PRE to LOOP,
- `trs_Loop_live` for liveness of the intermediate situation LOOP,
- `trs_Loop_to_Loop_1` for consistency and termination of the lower loop transition (case $[\texttt{val(cur)} < \texttt{x}]$) via LOOP,

- `trs_Loop_to_Loop_2` for consistency and termination of the upper loop transition (case $[\texttt{val}(\texttt{cur}) > \texttt{x}]$) via LOOP,

- `trs_Loop_to_Found` for consistency of the transition from LOOP to FOUND, and

- `trs_Loop_to_NotFound` for consistency of the transition from LOOP to NOTFOUND.

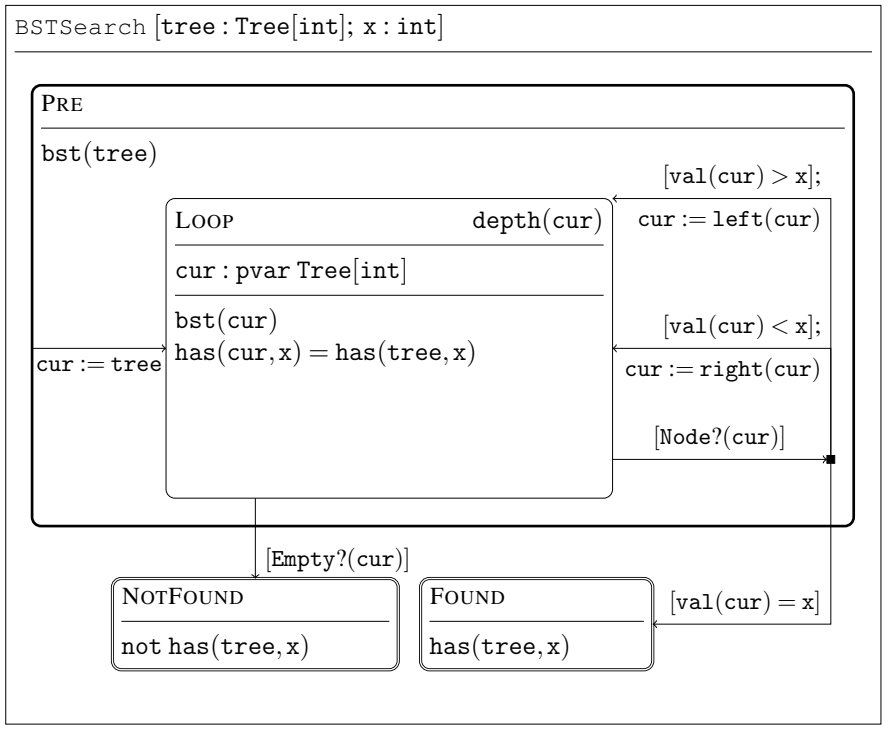


Figure 3.4: Invariant diagram determining the presence of a value in a binary search tree

To achieve a correct diagram, all lemmas should have valid PVS proofs. In addition to generating the verification conditions, Socos creates a default proof script invoking the Yices [36] SMT solver, which is integrated into PVS, on each generated lemma. Yices implements the DPLL algorithm for its SAT core and can handle complex propositional formulas. While the degree of automation depends largely on the specification and background theory, in practice, most lemmas are proved automatically by this proof script without requiring the user to enter the PVS interactive proof assistant at all. For the remaining lemmas, the user must start an interactive proof session in PVS-Emacs and carry out the required proof steps there.

Listing 3.1: Background theory for binary search trees

```
TreeHelpers[T: TYPE+]: THEORY
BEGIN
  IMPORTING Tree;
  t: VAR Tree[T];
  x: VAR T;

  has(t,x): RECURSIVE bool =
    CASES t OF
      Node(y,l,r): y = x ∨ has(l,x) ∨ has(r,x),
      Empty: false
    ENDCASES
    MEASURE t BY <<;

  depth(t): RECURSIVE nat =
    CASES t OF
      Node(y,l,r): max(depth(l), depth(r)),
      Empty: 0
    ENDCASES
    MEASURE t BY <<;
END TreeHelpers

BinarySearchTree: THEORY
BEGIN
  IMPORTING Tree;

  t: VAR Tree[int];
  x: VAR int;

  sup(t,x): RECURSIVE bool =
    CASES t OF
      Node(y,l,r): y ≤ x ∧ sup(l, x) ∧ sup(r, x),
      Empty: true
    ENDCASES
    MEASURE t BY <<;

  inf(t,x): RECURSIVE bool =
    CASES t OF
      Node(y,l,r): x ≤ y ∧ inf(l, x) ∧ inf(r, x),
      Empty: true
    ENDCASES
    MEASURE t BY <<;

  bst(t): RECURSIVE bool =
    CASES t OF
      Node(y,l,r): sup(l,y) ∧ bst(l) ∧ inf(r,y) ∧ bst(r),
      Empty: true
    ENDCASES
    MEASURE t BY <<;
END BinarySearchTree
```

Listing 3.2: PVS theory containing verification conditions for Figure 3.4

```
BSTSearch: THEORY
BEGIN
  IMPORTING TreeHelpers;
  IMPORTING BinarySearchTree;

  x: int;
  tree: Tree[int];
  cur: VAR Tree[int];

  trs_Pre_to_Loop: LEMMA
    bst(tree)
    ⇒ LET cur=tree IN
      bst(tree) ∧ bst(cur) ∧
      has(cur,x) = has(tree,x)

  trs_Loop_live: LEMMA
    bst(tree) ∧ bst(tree) ∧ has(cur,x) = has(tree,x) ⇒
      Empty?(cur) ∨
      (Node?(cur) ∧ val(cur) < x) ∨
      (Node?(cur) ∧ val(cur) > x) ∨
      (Node?(cur) ∧ val(cur) = x)

  trs_Loop_to_Loop_1: LEMMA
    bst(tree) ∧ bst(tree) ∧ has(cur,x) = has(tree,x) ∧
    Node?(cur) ∧ val(cur) < x
    ⇒ LET v0=depth(cur), cur=right(cur) IN
      bst(tree) ∧
      has(cur,x) = has(tree,x) ∧
      0 ≤ depth(cur) ∧ depth(cur) < v0

  trs_Loop_to_Loop_2: LEMMA
    bst(tree) ∧ bst(tree) ∧ has(cur,x) = has(tree,x) ∧
    Node?(cur) ∧ val(cur) > x
    ⇒ LET v0=depth(cur), cur=right(cur) IN
      bst(tree) ∧
      has(cur,x) = has(tree,x) ∧
      0 ≤ depth(cur) ∧ depth(cur) < v0

  trs_Loop_to_Found: LEMMA
    bst(tree) ∧ bst(tree) ∧ has(cur,x) = has(tree,x) ∧
    Node?(cur) ∧ val(cur) = x
    ⇒ has(tree,x)

  trs_Loop_to_NotFound: LEMMA
    bst(tree) ∧ bst(tree) ∧ has(cur,x) = has(tree,x) ∧
    Empty?(cur)
    ⇒ not has(tree,x)

END BSTSearch
```

3.7 PVS Ground Evaluator

Specification languages are generally not designed to be executable. However, the PVS ground evaluator provides an execution mechanism to evaluate PVS specifications while preserving the soundness [27]. The PVS ground evaluator provides a translator from an executable subset of PVS into Common Lisp. A proof goal is generated to evaluate a ground PVS expression with respect to the theory definition to ensure soundness. PVS also provides an interactive environment to accept a PVS expression from the user and print the result of the evaluation by read-eval-print loop.

For defining semantic attachments, one needs to specify the corresponding constant or function in the Lisp programming language. A semantic attachment is a user-defined executable term evaluation function that enhances the functionality of the ground evaluator. The semantic attachment, defined for a constant or function in PVS language, should be registered for the PVS expression that the attachment is defined for.

Here, we have an uninterpreted PVS function *abs* for the familiar function returning an absolute value of a given integer. To be able to execute this function, we define a Lisp function as the semantic attachment to this theory.

```
abs : THEORY
  BEGIN
   x: int
   abs(i:int): {j:int| (i<0-> j=-i) ∧ (i>=0 -> j=i)}
  END abs
```

The Lisp code is then defined by the macro `defattach` in Common Lisp as following.

```
(defattach abs.abs(x) (abs x))
```

The `defattach` associate the `abs` function imported from the `abs` theory—`abs.abs(x)`—to the built-in `abs` function—`(abs x)`—in Lisp.

3.8 Summary

In this chapter, we have given an overview of the theoretical foundations of IBP and described the software architecture of Socos, a verification platform built from the ground up to support the IBP workflow.

Particular to IBP is that programs are constructed as diagrams which are proved correct incrementally transition by transition. Good tool support is essential to enable the user to apply IBP efficiently in practice. State-of-the-art automatic theorem provers are important end-game components of this toolchain, as they provide the final stamp of approval that a program is correct. However, equally important for the outcome of the verification is the tool support for defining, validating, and modifying the invariant structure and background theory of a program. It is during these stages that the tasks of the automatic theorem provers are decided, and these provers' ability to perform well—with regards to time and memory usage—depends on both complete and efficient input. By efficient we mean taking advantage of background theories available to the prover, both built-in and application-specific background theories, to reduce the proof search problem to manageable size.

Exploration and validation of the invariant structure of a program is also crucial at the pre-proving stages to reduce the risk that the programmer is verifying a program

which is over- or underspecified. We believe that a verification tool should formalize the runtime semantics in the verification framework, and provide access to the formalization at the user interface level. This allows the programmer to inspect the state and invariants visually while constructing a diagram, while staying in the formal framework of the theorem prover.

The work presented in papers III and IV assumes IBP as the base verification methodology and extends upon it, and Paper III is also an implementation contribution to the Socos environment. Both papers address the research problems outlined in this chapter. The architecture of Socos allows extension of its frontend diagram editor, by adding components to the Eclipse plugin. It also allows extension to the backend by modifying its Python verification driver and prover (PVS) with semantic attachments. The PVS ground evaluator has been utilized in Socos debugger to support validation of invariant-based programs defined in Socos by allowing the constructed specification to be evaluated by running them on concrete (ground) data. Both extension mechanisms were utilized in the work described in Paper III.

Chapter 4

Event-B and Rodin Tool

Event-B [1] is a formal verification method based on the set theory [89] and the theory of refinement [75, 97]. Event-B defines a modeling language expressing a system as a state machine in which transitions between states are represented by a set of guarded events. Each event should preserve the properties—invariants stated as predicates—given for the machine. The types in this modeling language are inferred from the properties, expressed as set membership constraints, on variables of the model. To prove the correctness of Event-B model with respect to the specified properties—e.g., each event preserves the invariants of the machine—proof obligations are derived in the form of sequents. A proof obligation is discharged by applying inference rules to the sequent.

Managing the complexity of proof obligations is of crucial importance when verifying systems such as safety-critical control systems.[1] These are practically always large systems whose verification requires effective decomposition, so that the complexity of individual proof obligation is kept low and the components can be verified in a modular way. Event-B is extended with refinement to make automatic verification more feasible by reducing the complexity of models. Each refinement concretizes the model further towards the implementation. Refinement corresponds to vertical scaling while decomposition [3] can be seen as horizontal scaling. In decomposition, a model is divided into sub-models which can be refined separately.

This chapter gives the background for our Event-B modeling approach presented in papers I and II. The approach focuses on the following modeling stages:

Decomposition: The suggested approach provides a method of model decomposition based on the aspects that are present in the domain.

Simultaneous development: Each sub-model is refined further independently in order to support parallel work within a team, as well as to reduce the complexity of the verification conditions.

Composition: The composition preserves the correctness of the refinement with respect to the initial abstraction.

[1] `http://www.macs.hw.ac.uk/~air/sta/pdflec/lec-4-safety-a5.pdf`

4.1 Event-B Models

Context and *machine* are the basic constructs of Event-B models describing the static and dynamic parts of a specification respectively. The static parts of the model, i.e., constants, carrier sets (user-defined types) and axioms, are defined in the context. Machine is analogous to a state transition system where the state is represented by a set of variables, and transitions by a set of guarded events modifying the variables.

Machines include *invariants* to ensure the consistency of the model as the state changes. Theorems defined in a machine should be derivable from the axioms. *Events* are transitions over the states of a model. Each event is composed of *guards* and *actions*. A guard is a condition that when evaluated to true, triggers the corresponding action which updates the state. An event e in Event-B in a general form is represented by the term

$$e \quad \widehat{=} \quad \texttt{ANY}\ x\ \texttt{WHERE}\ G(x,v)\ \texttt{THEN}\ A(x,v)\ \texttt{END} \tag{4.1}$$

where x is the event's parameter, v denote the machine variables, $G(x,v)$ the guard and $A(x,v)$ the action [50]. A guard that is true ($\top$) in all states of the model does not require an evaluation, in which case the event is effective if it includes at least one action changing the state. The actions associated with events are modeled as generalized substitutions. Generalized substitution expresses the transformations of the values of the state variables by relating the postcondition to its weakest precondition (the largest set of initial states from which the transition is guaranteed to establish the postcondition at the end of the execution). The simplest action is the deterministic assignment

$$a := E(x,v) \tag{4.2}$$

where a is one of the variables in v and $E(x,v)$ is an expression over parameters x and the machine variables v. An action can also be a non-deterministic choice from a set of values

$$a :\in S(x,v) \tag{4.3}$$

where a is a state variable and $S(x,v)$ is a set expression. In the most general form, an action consists of a predicate that relates the pre-action and post-action values of the assigned variables:

$$a :| \ P(x,v,a') \tag{4.4}$$

The variable a in 4.4, can after execution of the action hold any value satisfying the predicate $P(x,v,a')$ relating the values v before the action to the values a' afterwards.

Machines can contain variants for proving the convergence of events. A proof obligation that each convergent event decreases the variant ensures that the set of events cannot take control forever.

Contexts can be extended with other contexts. The extending context inherits all the definitions of the context being extended. Machines may be refined by other machines based on the action system formalism [10]. By refinement, a system can be formalized in different abstraction levels, each abstraction extending the previous model by applying the refinement rules; *data refinement* [74] and *superposition refinement* [11]. In data refinement, concrete variables are introduced to replace the abstract variables. The consistency between the models after data refinement is preserved by

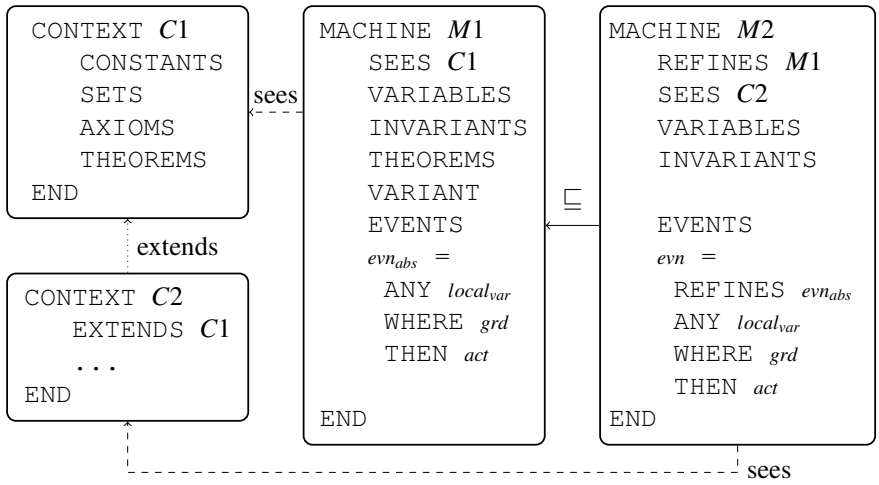


Figure 4.1: Context and Machine in Event-B

the invariants—known as gluing invariants—defining the relation between the refined variables and the abstract ones. In superposition refinement, either new events are introduced to concretize the system or the previously defined events are extended with additional guards, parameters or actions.

The connection between the contexts and machines is established by the SEES relation, stated in the machine, to express which context is visible in that machine. The constants and carrier sets of the context are then accessible to the machine and the axioms and theorems are assumed in the machine in the proving stage. An example of two contexts $C1$ and $C2$ and two machines $M1$ and $M2$ is given in Figure 4.1. The refinement relation between the two machines is stated as $M1 \sqsubseteq M2$.

The proof obligations for an Event-B model include but are not limited to invariant preservation, feasibility, guard strengthening and merging, finite set and decreasing variant, simulation and well-definedness [1]. The invariant preservation ensures that the invariants are still valid for the updated variables after each assignment. The feasibility condition of an action guarantees that there exists at least one value satisfying the before-after-predicate of the action—e.g., that the set of values attainable for the left-hand side of a non-deterministic assignment is not empty. Guard strengthening ensures that a concrete event can occur only when the abstract event can occur—assuming the invariants—while guard merging establishes a correctness condition upon merging abstract events into a concrete event by conjoining their guards. Simulation verifies that the abstract event's behavior corresponds to the concrete event's behavior. Proving the termination condition in Event-B also follows a variant declaration as a finite set that is decreased by each convergent event—named finite set and decreasing variant proof obligations. Well-definedness is stated as a predicate describing a condition in which an expression or predicate can be safely evaluated. Well-definedness of axioms, invariants and guards can be stated as theorems and the validity of them should follow from the model.

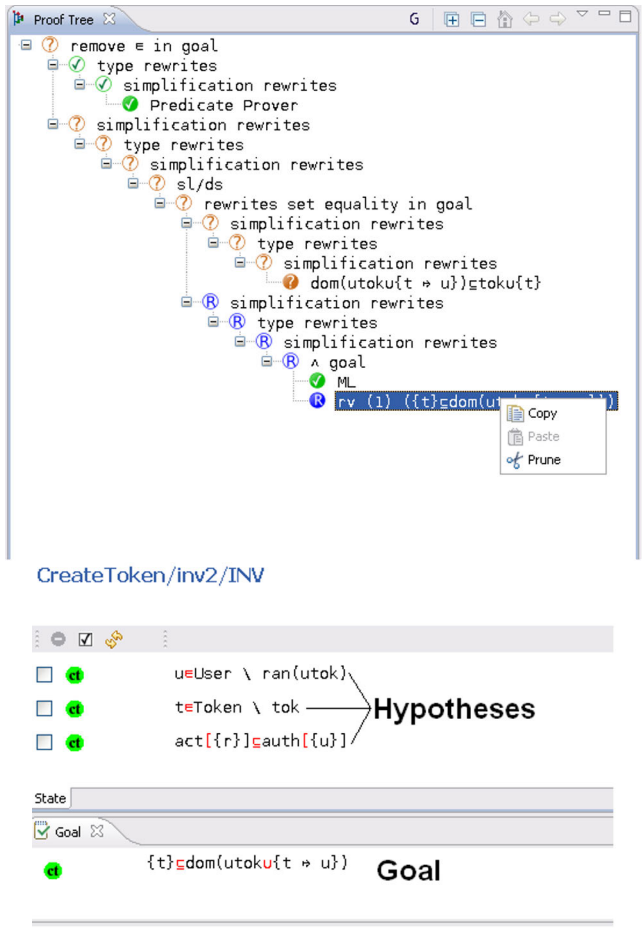


Figure 4.2: Rodin proof tree [60]

4.2 Rodin Tool

Rodin [2] is a Java-based platform supporting the specification and refinement of models written in the Event-B language. The proof obligations for both the consistency of refinement steps and the correctness of each abstraction level is generated and sent to the auxiliary theorem provers such as Atelier-B [71] and SMT solvers, e.g., Z3 [31], CVC4 [12] and Alt-Ergo [24]. The correctness of the refinement between the abstractions is also guaranteed by generated proof obligations.

Proving in Rodin also follows a construction of a proof tree and a list of sequent rules to be applied to deduce the proofs to true. An example of a proof tree is shown in Figure 4.2.

Since the effort in both constructing a model and discharging the proof obligations can easily become overwhelming in larger models, methods have been proposed to componentize a model in such a way that each sub-model could be developed and refined further with respect to the whole model. This is the basis of the third paper of

the thesis. We will discuss decomposition approaches in the next section.

UML-B [101]—an extension to Event-B specification editor in Rodin—provides UML-like diagrams such as class diagram for representing the entities of a system and state machines with transitions indicating the change of state in a system. The state machine in this case can reveal shortcomings in the model while supporting the agile principles [14]. A core principle of the agile manifesto is prioritizing response to change over following a plan. As it is more feasible to address changes to a subset of the system's components [39], alignment with this principle requires an effective decomposition method. The decomposition, in addition, provides a base for independent simultaneous development of each sub-model in the later stage, supporting both vertical and horizontal scaling. The final product is the composition of each of the sub-models.

4.3 Event-B Decomposition

Decomposition is introduced to divide a model into sub-models such that each sub-model can be further refined. Two types of decomposition have been proposed, shared-variable decomposition [51, 3] and shared-event decomposition [99].

In shared variable decomposition, the events get distributed between several machines. Each event may reference a number of variables in guards and statements. The variables that appear in events belonging to only one of the sub-models considered private variables for that model. The variables which appear in events of several models are annotated as shared variables. To not loosen the abstraction, external events are introduced in each of the sub-models including the shared variables which are not to be data-refined further, but to serve as invariant preservation conditions. This is shown in Figure 4.3 for model M including variables v_1, v_2, v_3 and events e_1, e_2, e_3. Machine M is decomposed into two sub-models D_1 and D_2. Distribution of events between the two machines, leading to placement of events e_1, e_2 in machine D_1 and e_3 in machine D_2.

Machine M is a trivial refinement of each machine, D_1 and D_2. Proving that M is a refinement of D_1 is attained by the fact that the events e_1 and e_2 remain unchanged in the refined model. Refinement of the additional event e_3 with a new variable v_3 is proven by proving that event e_3 is the refinement of e_{3ext}. Similar refinement rules are applicable when proving that M is the refinement of D_2. Event e_3 is the refinement of itself while event e_1 is refining $skip$. Event e_{2ext} is refined by event e_2 by introducing a new variable v_2.

After refining each of the sub-model, next is to recompose the refined models. Machine R in Figure 4.4 presents a recomposition of the refined models. Model R is the refinement of each of the refined machines $D_{1_{ref}}$ and $D_{2_{ref}}$. Since the variables and events of both machines—no shared event—are merged in machine R, this machine is trivially the refinement of the machines D_1 and D_2.

Finally, it should be proven that the machine R is the refinement of the original machine M. This proves the monotonicity of the recomposed model with respect to the applied decomposition.

In shared event decomposition, the variables are distributed between machines and based on the variables of each machine, the events are dedicated to the machines containing the variables used in the event. The events containing variables of several

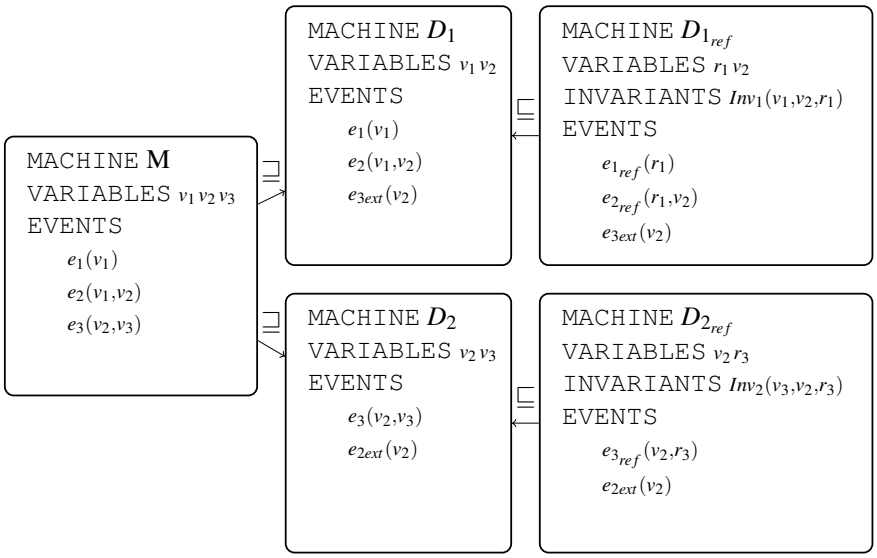


Figure 4.3: Event-B shared-variable decomposition

decomposed machines are split so that the decomposed event contains only the variables of a machine that it will be placed into. After this stage, the sub-models could be refined further without constraints. This approach is inspired by communicating sequential processes (CSP) [54] designed to describe patterns of interaction in concurrent systems.

4.4 Summary

An overview of Event-B formal verification and decomposition methodologies has been discussed in this chapter. Verification of systems becomes a heavy task for the provers as soon as the system grows larger in size. To deal with the complexity of proof obligations, two techniques have been followed.

Refinement, by specifying an abstract model first and refining it step by step.

Decomposition, breaking the model into independent sub-models, refining each separately, and subsequently re-composing them into a model that is a refinement of the initial model

Refinement is reducing the complexity of the proofs but does not conform well with parallel and agile development due to its sequential nature. Decomposition provides ground for including the agile approaches of simultaneous development of the component models. Paper I of this thesis explores identifying and formalizing abstractions that can be modeled, refined, and recomposed in the context of a critical system case study. Paper II surveys existing tools for visual and parallel development of system using this approach.

The decomposition is also a motivated approach when the verification is done by a single individual. After preparing the sub-models, each is ready for independent developments while still preserving the ties for later merge. The main complexity stems from ensuring that the refined sub-models can be recomposed. This is addressed

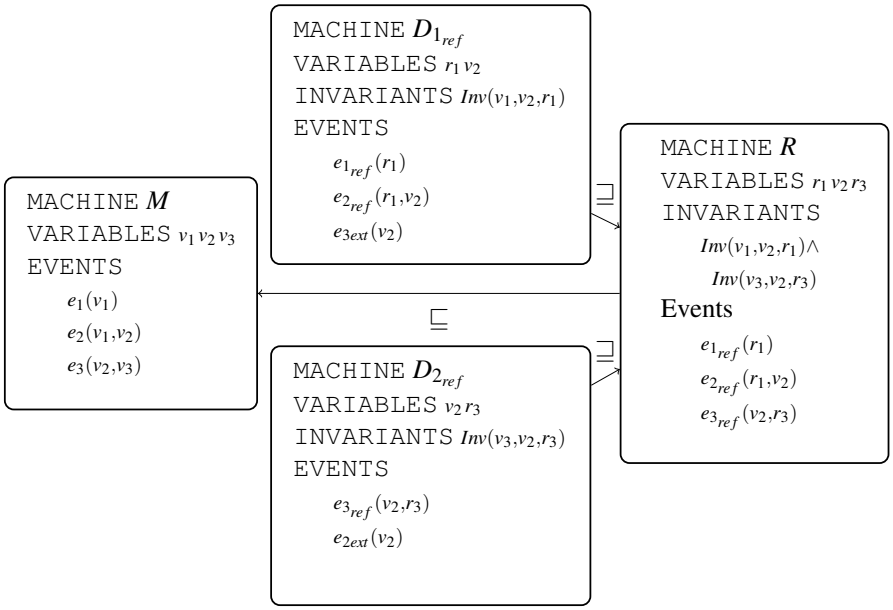


Figure 4.4: Event-B recomposition

when applying the decomposition. The composition will then be proved as a correct refinement of the abstract model.

Chapter 5

The Why3 Platform

Why3[1] [16] is a deductive software verification platform and a frontend to a number of theorem provers. We have chosen this platform for extension for the following reasons:

A well-designed API makes Why3 a good option for extension. The API includes but is not limited to handling parse trees, typing, drivers for external provers and proof tasks. We have utilized these APIs in our extension to the language and the proof mechanism.

The verification languages it provides, both as a pure logical specification platform and a programming language with logical declarations [41]—called WhyML.

Rich range of supported provers such as PVS, Isabelle [80], Coq [15] and SMT solvers [32], e.g., CVC4, Z3 [31] and Alt-Ergo [24].

The formal system of Why3 is comprised of first order logic with polymorphic types, pattern matching, inductive predicates and algebraic datatypes. The WhyML language is quite similar to ML style languages but includes additional constructs for verification purposes. The verification features of the language consists of pre- and postconditions, loop invariants and possibility of writing ghost code or data—code/data which exists strictly for verification purposes. Verification conditions for consistency proofs are then extracted based on the weakest precondition calculus. The termination condition is built based on the variant declaration for iterative constructs including loop, recursive functions and datatypes.

5.1 Why3 Specification Language

This section gives a brief introduction to some core elements of the Why3 language: types, program specification constructs and theories/modules. We refer to the Why3 documentation[2] for a complete description of the language.

Why3 built-in types are Booleans, integers, real numbers, tuples and mappings (functions). For user-defined types, algebraic data type constructors are supported—record and tuple are special case of algebraic types. Type declarations may contain

[1] `http://why3.lri.fr`
[2] `http://why3.lri.fr/doc`

uninterpreted type variables. WhyML extends the typing of Why3 by mutable records and type invariants for user defined data types.

Listing 5.1 shows an example of a WhyML binary search function implemented with imperative statements and annotated with preconditions, postconditions and invariant (the example originates from the Why3 gallery of formally verified programs[3]). WhyML supports imperative statements such as **`while`**—looping through the elements of the array in `BinarySearch` example—and **`for`** loops. To enable termination and consistency proofs of loops, **`variant`** and **`invariant`** annotations are supported.

Preconditions in WhyML are expressed with **`requires`** blocks—e.g., in the example stating that the the array is sorted—and postconditions with **`ensures`** blocks—e.g., that the function return value is an index into the array holding the sought element. The precondition is assumed upon entry, and the postcondition should be established by the program upon termination. WhyML differentiates normal and exceptional exit points of functions. To terminate the program at an exceptional exit, a **`raise`** statement is available—e.g., if the element is not found in the `BinarySearch`, the exception `Not_found` is raised. To be able to reason about their correctness, exceptions are to be stated in the procedure's contract with **`raises`** annotations—which implies the sought value does not exist in the array in the `BinarySearch`. The **`raise`** statement and **`raises`** annotation should be followed by a predeclared exception type (here `Not_found`).

Why3 provides a modularization construct for encapsulating definitions into their own namespace so that they can be combined and reused. There are two types of constructs, *theories* which contain purely logical definitions, and *modules* which may additionally contain WhyML programs. For instance, the `BinarySearch` module contains the declaration `use int.Int`, which imports the type `int` and associated operators from the basic theory of integers. This and the other theories imported in Listing 5.1 are part of the Why3 standard library.[4]

Why3 also features a graphical user interface, consisting of a theory editor and an interactive proof editor. A WhyML program can also be compiled into a correct-by-construction OCaml program.

5.2 Proof Transformation

In Why3, a task is a context derived from the declarations in the theory, a set of axioms, and a formula to be proven. A formula in a proof task, can for instance be a proof obligation that the statement inside the while loop preserves the loop invariant in the `BinarySearch` example. Transformations, defined in the drivers of each prover, are applied on tasks to transform the task into a readable format for each supported theorem prover [40].

Before sending the proof obligations to theorem provers, it is possible to apply a series of transformation rules to reduce the goal to be discharged into a form more suitable for automatic theorem proving. For transformation to be sound, the validity of the transformed task should imply the validity of the original task. Logical transformations can be grouped into computational transformations, eliminating transformations,

[3] `http://toccata.lri.fr/gallery/why3.en.html`

[4] `http://why3.lri.fr/stdlib/`

Listing 5.1: Binary search in WhyML

```
module BinarySearch
  use int.Int
  use int.ComputerDivision
  use ref.Ref
  use array.Array

  (* the code and its specification *)

  exception Not_found (* raised to signal a search failure *)

  let binary_search (a: array int) (v: int) : int
    requires {
      forall i1 i2. 0 ≤ i1 ≤ i2 < length a → a[i1] ≤ a[i2]
    }
    ensures {
      0 ≤ result < length a ∧ a[result] = v
    }
    raises {
      Not_found → forall i. 0 ≤ i < length a → a[i] ≠ v
    }
   =
     let ref l = 0 in
     let ref u = length a - 1 in
     while l ≤ u do
        invariant { 0 ≤ l ∧ u < length a }
        invariant {
          forall i. 0 ≤ i < length a → a[i] = v → l ≤ i ≤ u
        }
        variant { u - l }
        let m = l + div (u - l) 2 in
        assert { l ≤ m ≤ u }
        if a[m] < v then
           l := m + 1
        else if a[m] > v then
           u := m - 1
        else
           return m
        done;
     raise Not_found
end
```

encoding transformations and splitting transformations. In the following, we exemplify some of the transformation rules in Why3.

Computational transformation is applied to replace the functions and predicate symbols by their definitions in the current goal. This transformation eagerly applies all known computation and simplification rules, while there are specific transformations that only apply user-provided rules.

Eliminating transformations replaces one construct by another—e.g., transfers antecedents of implications and universal quantifications of the goal into the premises of the task.

Encoding transformations are employed to eliminate unsupported constructs for the target provers—e.g., an `if-then-else` expression is rewritten as a conjunction of two implications.

Splitting transformation strategies break up a large propositional formulae into smaller ones that are easier to prove—e.g., having a conjunction, splitting would construct a goal for each of the term in the conjunctive term.

Custom transformations can also be defined as OCaml functions in development and registered to be available in the GUI.

5.3 User-Defined Theories

Why3 has a standard library of theories, which may be extended with user-defined theories. For example, a theory may introduce a polymorphic list type as the following algebraic data type:

```
theory List
  type list α = Nil | Cons α (list α)
end
```

Here α is the type of the elements, and `Nil` and `Cons` the two constructors. To define additional data structure properties, predicates, functions and lemmas may be introduced in theories. For instance, in the theory `Length` below, the length of a list is defined as a function returning 0 for an empty list and 1 plus the length of the tail of the list otherwise. Additional lemmas state that the length function returns zero if and only if the list is empty and otherwise the length returned by the function is greater than zero for a non-empty list.

```
theory Length

  use import int.Int
  use import List

  function length (l: list α) : int =
    match l with
    | Nil      → 0
    | Cons _ r → 1 + length r
    end
```

```
  lemma Length_nonnegative: forall l: list α. length l ≥ 0

  lemma Length_nil: forall l: list α. length l = 0 ↔ l = Nil

end
```

The other application of modular definition in Why3 is in importing theories. Theories can be either used or cloned in other theories; the difference lies in the mechanism of import. Cloning copies all the definitions into the current namespace of the theory and provides the possibility of type, function and predicate instantiations that were defined as abstract in the original theory.

5.4 Extending the Why3 Parser

Why3 is an open source project and provides an API that is accessible through OCaml code. For creating Why3 terms, creating tasks for the proofs and calling external provers on the tasks, the Why3 API[5] can be exploited; however, to extend the language, new grammar rules have to be defined as an extension to the Why3 parser and new tokens as an extension to the lexer.

Each grammar rule is then attached to an action producing an OCaml value representing a Why3 term. This will later on be type-checked and subsequently translated into verification goals for the automatic theorem prover. This involves identifying the theories, creating task from the theories, applying transformation rules. Some goals may already get discharged in this stage. The remaining goals are then sent to the installed provers. The Why3 API supports registering a file extension to customize a parser and term transformation implementation.

Parsers employ one of the two techniques of top-down or bottom-up in constructing abstract syntax tree (AST). Bottom-up parsers run efficiently in linear time since they do not involve any backtracking mechanism. In bottom-up parsers the parse tree is constructed from bottom left by composing the tokens based on production rules upwards and to the right. Bottom-up parsers employ the shift-reduce method. The parser maintains a stack of visited tokens; shifting removes the next token and adds it to the parse tree, while reducing applies the next matching grammar rule. Menhir[6], the parser generator of Why3, is a bottom-up parser.

Menhir is a parser generator for OCaml programming language is based on Knuth's LR(1) parser generation technique [66]. LR(1) parser generation are applied on languages that can be parsed from left to right and by looking at a defined finite number of characters ahead without backtracking to consider the previous decision. This is due to the way LR parsers utilize pre-built parse tables in order to build the parse tree.

Why3 does not support mini-language definition—i.e., there is no mechanism for modularly extending the language without modifying the grammar of the Why3 language itself. Hence, the embedded grammar needs to be merged with the overall Why3 grammar which introduces some lexical constraints on it, e.g., having to use the

[5] http://why3.lri.fr/api/

[6] http://gallium.inria.fr/~fpottier/menhir

same token set, and ensuring rules of the embedded language do not conflict with the Why3 language.

5.5 Summary

We have given a brief overview of Why3 specification language, its structure and verification mechanism in this chapter. Many verification platforms do not handle well the transition from verified programs, defined in a context of logical framework, to the executable code. Why3 features the correctness proof together with completeness within the implementation language WhyML. WhyML is an ML style programming language supporting verification by code annotations. The verification conditions for consistency and termination are generated by Why3 and proved by automated provers.

Why3 came as a natural choice in extension of a deductive verification tool. Well-supported theorem provers, transformation mechanism for proofs and modular theory extensions were the main features that prompted us to select this platform as the base of our work.

A diagrammatic notation for array invariants in the context of invariant-based programming is presented in Paper IV. In this paper, we presented a meta theorem for generation of verification conditions for array programs. We hand translated the programs into Why3 theories together with the verification conditions as goal. We established the correctness proof by proving the goals in the context of the theory.

In Paper V, we further developed the language into a domain-specific language without dependency to invariant-based programming. The presented language is an independent language that can be inserted into any logical verification platform. As a reference, we implemented the language as an extension to the Why3 platform.

Chapter 6

Summary of Papers

In this chapter, we summarize the key results in each of the papers that are part of this thesis.

Paper I. Parallel Development of Event-B Systems with Agile Methods

To formally verify a system, the requirements of the system need to be specified in advance; then the model can be built to satisfy the defined requirements. In Event-B, consecutive refinement steps are introduced to transform the first abstract model into an executable system. This type of verification has been criticized because of its inflexibility to reflect changes on the model upon modification of requirements. The second concern is that it has limited scalability.

In this paper, we applied model decomposition methods to facilitate the agile principle of responding quickly to changes. Introducing modularity and isolation of each sub-module allows for better management of changes to the specification. Isolation of Event-B models is provided by distributing the system variables and their invariants between the sub-models. Having decomposed the system model, changes in the requirement are likely to impact only a few sub-models.

The scalability concern has been addressed with refinement. The complexity of proof obligations is reduced by gradually introducing variables and statements to concretize the abstract models. Even though refinement provides vertical scaling, it does not serve well in horizontal scaling. Decomposition of models into sub-models that can be further refined, on the other hand, enables horizontal scaling of Event-B verification.

Traditional decomposition methods relies on an abstract model in the start of the decomposition stage to establish and retain the relation between the initial abstract model and the decomposed model. However, it still suffers from the requirement changes in which affect the initial abstract model. The proposed approach in this paper suggests the separation of concerns, or aspects, of the system, and the development of individual independent abstractions for each of those. The abstractions are subsequently refined in isolation, and the refined models are merged into a single final model. Figure 6.1 introduces two abstraction for each aspect of a model, named Aspect 1 and 2, their refinement and the composition of the two abstractions.

The presented approach in the paper has been studied in the context of the Landing Gear case study of Boniol and Wiels [17], originally proposed in the ABZ'2014 conference as a formal methods case study representative of an industrial needs. We split the original specification into two independent system abstractions: a moving gear system controlling the extension and retraction of the landing gear, and an analogical switch representing the hydraulic control system. The abstractions were modeled in UML-B and data-refined independently. To merge the models, we copied the most refined model of the analogical switch system into the most refined model of the moving gear system to create the final model. Events common to both models were refined into a single event in the final model. The correctness of the models was proven in Event-B.

The approach does not address the decomposition of the initial abstract model into multiple machines. To achieve this, hard assumptions on models are required, e.g., no shared variables. Syntactically, no shared variables could be attained by scoping the variables to the machines and keeping the scoped identifiers in the merged model. The issue with this approach, however, is loosening the concept of refinement by merging the independent refined models.

The contributions of the author in this paper include co-developing the above mentioned model decomposition method, investigating its implied limitations, and implementing the two Landing Gear aspects and their refinements in Event-B.

Paper II. An Overview of Formal Specification Languages and Tools Supporting Visualization of System Development

In this paper, we have explored tool support and particularly visualization tools for various formal specification languages. One aspect investigated in this paper is how the UML modeling language is supported by verification platforms. The reason for selecting UML is to investigate how the different UML diagrams could serve to visualize systems for verification.

Z is a formal specification notation based on set theory and predicate logic [102]. A construct in Z is called a schema and is divided into two parts: a declaration part and a predicate part. The declaration part introduces the variables and their associated types while the relation between variables stated as predicates are included in the predicate part. Schema contains both static and dynamic parts of a system. With respect to visualization there has been development in both animated execution of Z models [61]

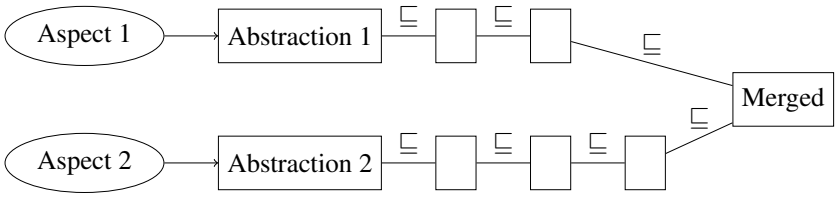


Figure 6.1: Merging the refined models of two abstractions

and Z-based specification UML diagrams [98]. There has been efforts in formalization of UML class structure with Z specification language. The static part of a class is formalized in the static part and the class instantiations and attributes are represented by state variables.

Alloy is another specification language that supports visualization in the form of UML diagrams [57]. Alloy is influenced by Z and the Object Constraint Language (OCL). Even though Alloy was motivated by the Z language, it does not support deductive software verification approach to prove the correctness of a model. Alloy, inspired by model checking techniques [58], is a light weight framework utilizing only a light SAT-based approach to either find a counterexample on a specified model, or to find a model to simulate the specification. Alloy analyzer not finding a counterexample on a model does not guarantee the correctness of the system. Since Alloy supports only a bounded model, the model should be defined on a finite number of objects. In Alloy everything is defined as relations and the model is described by adding constraints as assertions. To handle first order logic, alloy converts a quantified formula to a propositional correspondence. The model specified in Alloy can be visualized as entity-relation diagrams and UML class diagrams. The Alloy analyzer provides an animator for executing the model. The animator supports specifying a starting state (a pre-defined condition that holds) and a backward execution from a pre-defined state (by defining a post-state that holds).

Event-B provides a prototype tool called UML-B (as a plug-in to Rodin), supporting UML-style diagrams [101] which are semantically different from the object model of UML. Machines are represented by UML class diagrams where events of a machine are the methods in UML model. Event-B also provides supports to declare state-machines. State-machines can be animated with the ProB tool, which also provides visualization of the animation. The effect of UML-B as a graphical language in designing complex embedded systems, verified with the B language, has been studied with an approach named PUSSEE [107]. This approach covers all the co-design stages from specification down to the C implementation supporting visualizations, and the authors of this paper conclude that

> "Successful integration is important to the long term success of formal methods."

Event-B tools have been used by industry in several case studies. This has given rise to this paper for the comparison between the available tools in formal methods, their strength and weaknesses; and discovering the domains they are better suited for.

The author structured the paper jointly with the co-authors, wrote the overview of the specification languages and their tool support, and collected most of the related work.

Paper III. Proofs and Refutations in Invariant-Based Programming

This paper presents the extension of the Socos program verifier with an execution environment. As described in Chapter 3, Socos provides a graphical editor for constructing

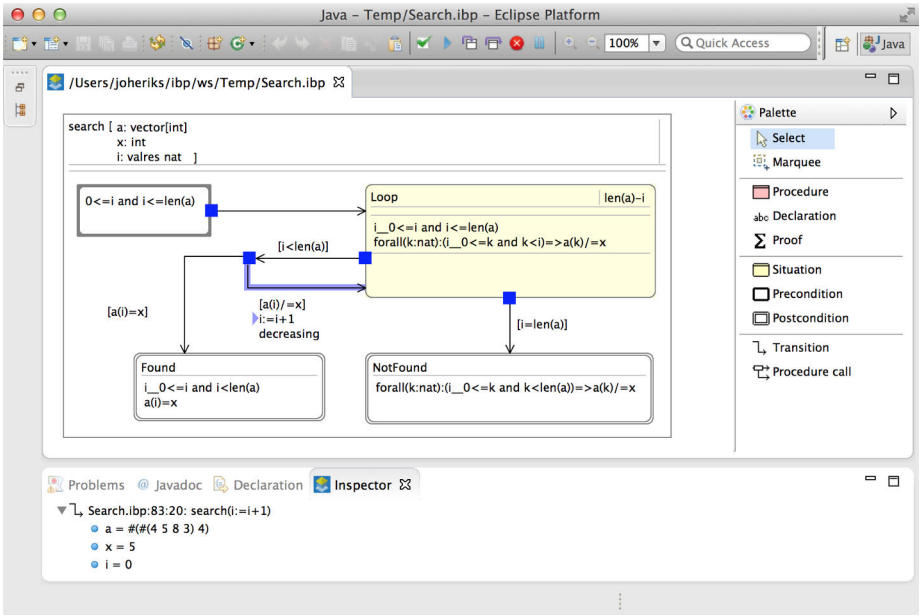


Figure 6.2: Animating a program in Socos

the program as invariant diagrams with the program specification and intermediate assertions stated in higher-order logic. The Socos backend generates the verification conditions and invokes automatic theorem provers on them. Unproved verification conditions are highlighted in the editor. An unproved condition may be due to the theorem prover not finding a proof for a complex condition, in which case interactive guidance may be required; or it may be due to an actual error in the program, which should be corrected. To assist the user in exploring the runtime behavior of their program as a complement to verification, as well as to execute the final program, we integrated an execution mechanism into Socos. The execution mechanism gives possibility of inserting initial values to the programs, and standard runtime debugging features (breakpoints, step into, step over, run, pause) were added to the Socos editor. In each step, the state of the program is displayed in a variable inspector and the next statement to be executed is highlighted in the diagram. A Socos debugging session is shown in Figure 6.2.

To model the execution, we implemented automatic translation of an invariant diagram into a PVS abstract datatype representing the program state. The datatype is comprised of constructors $Loc_1, \ldots, Loc_n$ for each location in the diagram—where a location is an invariant, a transition branch, or a statement on a transition—each of which take as arguments the variables visible in the location:

```
state: DATATYPE
  BEGIN
    Loc_1 ( Vars_1 ) :  Loc_1?
    ⋮
    Loc_n ( Vars_n ) :  Loc_n?
```

```
END state
```

Given the state representation, we generate a step function accepting one state as input and returning the next state. Evaluating this function corresponds to executing a single statement in the diagram. To match the current state of the execution, the function pattern-matches on the location, and for a matching location Loc_i returns the operational definition $[\![Loc_i]\!]$:

```
step (s:state): state =
  CASES s OF
   Loc1 ( Vars1 ): [[Loc1]]
   ⋮
   Locn ( Varsn ): [[Locn]]
  ENDCASES
```

The operational definition $[\![Loc_i]\!]$ follows the standard runtime semantics of the available statements in Socos (assignment, assertion, assumption, and procedure call). For example, an assignment statement at location Loc_i preceding another statement S at location Loc_{i+1}

$$\underset{Loc_i}{X := E;}\ \underset{Loc_{i+1}}{S}$$

constructs the next state by updating the assigned variable and keeping the other variables unchanged:

$$[\![Loc_i]\!] = (\textbf{LAMBDA}\ X : Loc_{i+1}(Vars_i))(E)$$

To execute a diagram, we have created a runtime in Lisp and the PVS implementation language as an integration into the Socos backend. The runtime executes the step function repeatedly until completion, or when called in the debug mode, step by step. In the latter mode, it receives commands from the debugging UI provided by the Socos graphical editor, and returns the stack trace. In the case of a failed assertion or no enabled guard in a situation, the step function repeats the current location. This either indicates a normal termination (if the program has reached its final situation) or an abnormal termination (if it occurs elsewhere).

The concrete evaluation of the step function is delegated to the built-in term evaluation mechanism of PVS, the ground evaluator, which evaluates expressions by first translating them to Common Lisp code and then executing them in the Lisp runtime. In addition to statements, invariants and assertions are evaluated if they are ground terms, i.e., if they do not contain uninterpreted functions or quantification over infinite domains. PVS semantic attachments can be used to enhance the ground evaluator with executable Lisp implementations for non-ground terms. This allows partial evaluation of such invariants and assertions, as well as execution of programs that are not in their final stage of refinement.

As an example, we describe how diagram execution can assist the derivation of a correct program for constructing a binary tree from a list of leaf depths.

The author contributed in designing the execution environment, carried out most of the front-end and back-end implementations, and prepared the case studies presented in the paper.

Paper IV. A Precise Pictorial Language for Array Invariants

In this paper, we proposed a pictorial language for expressing array invariants. The language is based on Reynolds's partition and interval diagrams [93], with some extensions. As Reynolds noted, assertions over arrays tend to get lengthy compared to the program itself. Also, the integer range conditions used to describe typical array properties are error-prone. To address these issues, Reynolds introduced a more intuitive diagrammatic notation based on disjoint integer intervals: if i and j are expressions denoting integers, the rectangle $i\boxed{}j$ when interpreted as an *interval diagram* stands for the for the interval $\{k \in \mathbb{Z} \mid i < k \leq j\}$, while when interpreted as a *partition diagram* stands for the predicate $i \leq j$. Interval diagrams describe integer intervals, while partition diagrams describe the relationship between intervals. In addition to this normal form of the diagram, either or both bounds of the diagram may be written on the right side of the adjacent vertical line to adjust the bound by -1, while a rectangle containing only one integer is short for a singleton interval:

$$
\begin{array}{rcl}
i\boxed{}j & \triangleq & i\boxed{}j-1 \\
\boxed{i}\boxed{}j & \triangleq & i-1\boxed{}j-1 \\
\boxed{i}\boxed{}\boxed{j} & \triangleq & i-1\boxed{}\boxed{j} \\
\boxed{i} & \triangleq & i\boxed{}\boxed{i}
\end{array}
$$

Partition diagrams may be written in chained form when the upper bound of a preceding diagram coincides withe the lower bound of the succeeding diagram. For example, the partition diagram $\boxed{0}\boxed{i}\boxed{}\boxed{n}$ stands for the predicate $0 \leq i \leq n$.

Colorings. In order to label disjoint intervals of an array, such that they can later be assigned predicates, we introduced the concept of a *coloring*: a mapping from intervals to a color. Syntactically, colorings are similar to Reynolds diagrams but are drawn as as filled rectangles rather than as outlines. For example, $i\blacksquare j$ states that the interval $i\boxed{}j$ maps to $\blacksquare$. This notation allows compact combination with partition diagrams when the bounds coincide. For example:

$$
0\blacksquare i\boxed{}n = \boxed{0}\boxed{i}\boxed{}\boxed{n} \wedge 0\blacksquare i
$$

Legends. To assign meaning to colored intervals, we introduced the concept of a *legend*. A legend is a coloring-predicate pair, stating the predicate the elements in all subranges of an array matching the given coloring should satisfy. For example, the legend

$$
A|\ \blacksquare\ : A(i) \neq x
$$

asserts that all indexes i in the array A having the color $\blacksquare$ should satisfy $A[i] \neq x$. Here the pattern describes the predicate over a single element of the specified color. An example of a legend involving two adjacent elements is the following assertion that continuous green ranges are sorted:

$$
A|\ \blacksquare\ \blacksquare\ : A(i) \leq A(i+1)
$$

Verification. We have developed the pictorial language in the context of invariant-based programming, and therefore it was natural to combine the pictorial language with invariant diagrams (however, the language itself is independent from the verification framework and could be reused in other verification tools). We defined the semantics of colorings and legends by translation into predicates over the program state. Nesting allows legends to be shared by multiple situations, as well as to be extended in substitutions with additional coloring predicates. After translation, the verification conditions of the diagram were extracted using the normal proof rules of invariant diagrams. We demonstrated the notation by verifying three fundamental array algorithms: insertion sort, linear search and binary search. After drawing the diagrams, we hand-translated them into verification conditions for the Why3 verification platform and proved them. As an example, figure 6.3 shows an invariant diagram representing binary search, where the invariants are given pictorially utilizing partition diagrams, colorings and legends. Our conclusions from the verification exercise were that the language supports writing clear and concise array specifications, while also being quite expressive, allowing many common array invariants to be stated.

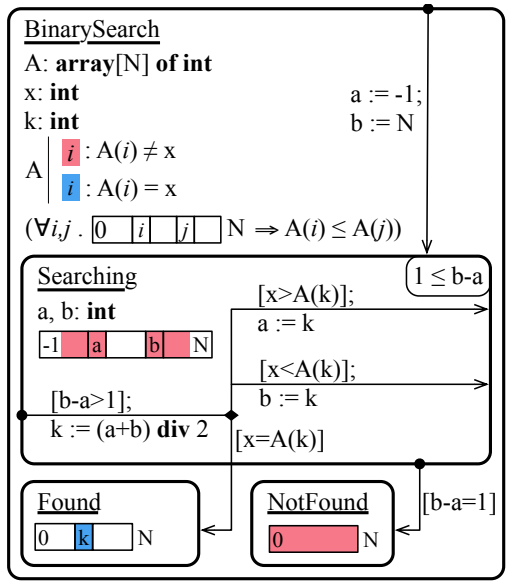


Figure 6.3: Binary search

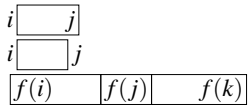
The author contributed in designing the pictorial language, evaluating it on multiple case studies (including all presented in the paper), and summarizing the results in the paper.

Paper V. A DSL for Integer Range Reasoning: Partition, Interval and Mapping Diagrams

In this paper, we presented an implementation for the Why3 verification platform of the notation proposed in Paper IV. The implementation extends the Why3 language with partition diagrams, mapping diagrams (a generalization of colorings) and legends. It introduces an alternative character-based syntax for the pictorial notations, a translation into Why3 terms, and Why3 theories encoding the diagram semantics.

Syntax. Below we show a few examples of character-based partition diagrams (left column) and the pictorial equivalence (right column). Syntactically, a Why3 term can occur on either side of the vertical bars delimiting the intervals:

```
[i| ... j]
[i| ... |j]
[f i ... | f j | ... f k]
```



A *mapping diagram* represents a partial function whose domain consists of a sequence of integer intervals. The mapping diagram syntax extends that of partition diagrams with a argument binder and a value term (preceded by `#`). For example:

```
[x → [i| ...#(-x) |j ...#(x) k]]
```

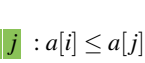
$$\begin{cases} -x & \text{if } i \le x < j \\ x & \text{if } j \le x \le k \end{cases}$$

If each interval is mapped to a constant, the binder may be omitted. This corresponds to a coloring; for example, the following represents a red coloring of the interval (a, b):

```
[[i| ... #R |j]]
```

A legend is declared with the **`legend`** keyword followed by an identifier, a sequence of parameters and list of coloring-to-predicate mappings. For example:

```
legend l(a:array int)(x:int) of col =
  i,j : [[i#G|j#G]] → a[i]≤a[j]
```



where `col` is some type containing the value `G`.

Translation. The Why3 grammar is defined by a collection of production rules, which the parser generator Menhir compiles into OCaml code. We have extended the Why3 grammar with production rules for the syntax of partition diagrams, mappings, and legends. Partition diagrams and mappings are term-level productions, while legends are theory-level productions. After parsing, an instance of the type-parametric algebraic datatype `diag` with the two constructors `P` and `L` is constructed:

```
type diag α =
  | P int (option α) (diag α)
  | L int (option α) int
```

Each diagram consists of at least one base interval `L`. The interval includes two terms expressing the bounds of the interval and and an optional term for mappings. The above datatype represents the normal form of a diagram; other forms are normalized during parsing.

Semantics. Semantics for the different types of diagrams is given by functions over `diag`. For partition diagrams, the function is merely a conjunctive predicate of `Int.`$\leq$ applications. For mapping diagrams, the function produces an **`if`**–**`then`**–**`else`** construct, while for legends it produces a parametric conjunction of universally quantified implications. Why3 supports marking specific theorems as rewrite rules, with the effect that they will be automatically applied on the current proof goal before the goal is sent to external theorem provers. For the specific case of a legend application to a mapping, we have defined a lemma that rewrites the application into an implication where the antecedent is a conjunction of integer inequalities. This rule serves to completely eliminate the meta-layer introduced by the language (namely terms of type `diag` and color types) from the problem given to an endgame prover.

Example. Listing 6.1 shows the Why3 rendition of a binary search procedure analogous to the diagram in Figure 6.3. If `x` is present in the array `a`, the procedure exits normally returning an index containing `x`, while if not present, it raises `Not_Found` ensuring that all elements in the array are different from `x`. These postconditions and the main loop invariants are expressed with the legend `found`. All verification conditions were proved automatically by a combination of Z3, CVC4, and Alt-Ergo after preprocessing by our tool and applying the rewrite rules.
The author contributed in the design and evaluation of the proposed DSL, as well as in the dissemination of the results. The author architected and implemented the Why3 tool support for the DSL, including extending the Why3 specification language and developing the theories formalizing the language.

Listing 6.1: Binary search

```
predicate sorted (a:array int) =
  forall i j : int .
    [0 ... |i| ... |j| ... |length a] → a[i] ≤ a[j]

type found_col = R | G
legend found (a:array int)(x:int) of found_col =
  i: [[i#R]] → a[i] ≠ x ;
  i: [[i#G]] → a[i] = x

let binary_search (a: array int)(x: int) : int
  requires { sorted a }
  ensures { [0 ... |result| ... |length a] }
  ensures { found a v [[result#G]] }
  raises { Not_Found → found a x [[0 ...#R |length a]] }
=
  let l = ref 0 in
  let u = ref (length a - 1) in
  while !l ≤ !u do
    invariant { sorted a }
    invariant { [0 ...|!l] ∧ [!u| ... |length a] }
    invariant {
      found a x [[0 ...#R |!l ... !u| ...#R |length a]]
    }
    variant {!u - !l}

    let m = !l + div (!u - !l) 2 in
    if a[m] < x then l := m + 1
    else if a[m] > x then u := m - 1
    else return m
  done;
  raise Not_Found
```

Chapter 7

Conclusions and Future Work

The development of deductive software verification has since the seventies given rise to a number of challenges for researchers working in the field. A major impediment preventing verification from becoming industry practice is the complexity of constructing formal machine-checkable proofs of program correctness in practice. When seeking to reduce that complexity in order to achieve scalability, challenges faced include improving tool support to automate the verification process to a higher degree, developing and evaluating new modeling languages that support more effective validation, combining formal verification with established software engineering techniques such as testing, and achieving an iterative workflow in the presence of a rigorous sequential verification method. This thesis has addressed the aforementioned concerns and has explored approaches in the forms of reducing proof complexity by decomposition techniques, a runtime environment for invariant evaluation, establishing a DSL for representing array data structures as an extension to an existing verification platform, and supporting visual languages in verification platforms by providing semantic translation of the symbols to the logical language of the framework. In conclusion, we revisit the original research questions as well as identify some current limitations and future work to address these.

7.1 The Research Problems Revisited

In this section, we discuss our findings in light of the research questions posed in Section 1.1.

Aspect-based model decomposition and refinement. The specification in formal verification is traditionally given after collecting the requirements and is assumed not to change. This is an impediment to the development of software of scale by agile principles as the requirements are prone to change over time. Moreover, the task of the theorem prover in verifying the correctness of a specification demands higher computational resources as it grows in size. The goal of refinement in deductive verification is to alleviate the proof effort through the vertical extension of a model by incrementally extending an abstract verified model with concrete models. Proving the verification conditions introduced through refinement guarantees the correctness of the concrete model. The chain of refinement is developed in a sequential manner. However, the ap-

proach is still limited in horizontal scaling, i.e., the concurrent development of multiple components of a system. The first research problem we address concerns decomposing a model to achieve such horizontal scaling. In the proposed approach described in Paper I and II, we suggest to first analyze the model in order to isolate abstractions that can be refined as independent chains. The recomposition of the final refinements from each chain then constitutes the final verified system. In our case study, a simplified version of a landing gear system, we identified a separation of two abstractions—the moving gear itself and the hydraulic control system— that led to a simpler verification and validation of the model (compared to a single chain of refinements). We confirmed that by dividing a model into multiple abstractions we could effectively decrease the complexity of the model, while at the same time parallelizing the development process. The merging of the sub-models was unproblematic due to the nature of the abstractions in our case study, but this may not be generalizable. Our experience from the case study gave us reason to believe that the incorporation of agile principles into the formal development of critical systems is realistic, and that it can be facilitated by aspect-based decomposition.

Combining static and runtime verification. Deductive verification platforms typically provide a modeling language for defining the program context and its execution under the operational semantics supported by the platform. Depending on the platform, the context and programs are translated to theories including verification conditions to be sent to theorem provers for discharging. The status of each condition—either proved or proof attempt failed—constitutes the prover's output. The output, however, mostly lacks information on the over- or under-specified invariants and other details that may give hints to identify the cause of failures. The second research question we posed was whether a combination of static and runtime verification can assist in identifying errors early, thereby saving time wasted on invalid verification conditions. At the time this problem was undertaken, a verification tool for a diagram-based language with runtime evaluation within the underlying theorem prover's formal logic, and with full animation of the results in the diagram, did not exist to the best of our knowledge. Socos supports specifications as nested diagrams defining nested structure of programs. The runtime execution extension, introduced in this thesis, supports evaluating the specification on the users' provided input as PVS theories and within the PVS theorem prover's platform. The editor also features a debugger displaying the result of the evaluation of each step in the diagram. The case studies we have carried out indicate that static and runtime verification is a powerful combination that not only makes it easier to identify the cause of failed verification conditions, but also makes the verification process more feedback-driven which lessens the gap between verification and testing. As a result of the decision to use an existing ground evaluation mechanism, in addition to simplifying the implementation we observed two advantages: a reduced trusted core, and minimized context-switching between proving and testing for the user.

Diagrams as formal specification languages. The third problem addressed in this thesis concerns diagrammatic languages in software verification, in particular how to design a diagrammatic language that is semantically precise to be useful for formal specification without losing clarity or legibility. We chose to combine two existing

visual languages, invariant diagrams and partition diagrams, into a fully diagrammatic language for verifying programs over arrays. This choice was influenced primarily by our previous experience with invariant diagrams, where we observed that almost all practical verifications of programs involved the programmer constructing one or more hand-drawn illustrations of arrays in order to clarify the constraints that should apply before encoding these in logic notation to be embedded in the invariant diagram. The main challenge with formalizing hand-drawn graphics turned out to be selecting the syntactic primitives. We finally arrived at three: *partition diagrams* expressing predicates over integer intervals, *colorings* for attaching labels to subarrays, and *legends* defining the properties of subarrays. While experimenting with these notations, we noticed that they turned out to be rather expressive visual constructs, allowing many common array invariants to be stated. Encouraged by these initial results, we decided to implement tool support for them. However, as the Socos tool was at that time no longer under active development, we decided to implement the notation as a lightweight extension to an existing verification tool. This led to the next and final research problem addressed in this thesis.

Domain-specific languages (DSLs) in software verification. The final research problem in software verification that we have addressed is the incorporation of the diagrammatic DSL developed in Paper IV in an existing verification platform. DSLs have been developed in different domains to give a minimal expressive representation of a system or property, often independent from the host language. The regular expression syntax for string matching and substitution for instance is one DSL that carries meaning independently of the embedding programming language. The semantics of a regular expression is independent of that of the host language and the runtime environment that evaluates it. Elegance in representation and abstract definition are two advantages of DSLs as target language extensions. An interpreter or compiler integrating the DSL to the target language, can accommodate the most efficient implementation of the DSL as opposed to custom or project-specific implementations. We have added support for a DSL to the formal verification platform Why3 for expressing predicates over sequential data structures, typically arrays, in specifications and invariants. The implemented DSL is a linear textual version of partition diagrams, mappings (a generalization of colorings) and legends. Our extension parses these diagrams and converts them to Why3 predicates. We observed that the integration could be completed in a technically sound manner by modifying the Why3 parser, despite the underlying platform lacking language embedding support as such. Once the parser was modified and the DSL lexically integrated, the semantics could be defined exclusively with Why3 user-level constructs, i.e., requiring no further modifications to the Why3 sources. Hence, we can confidently state that the integration does not compromise the semantic core of Why3. In practice, using the DSL for expressing specifications, invariants and interactive theorem prover sessions felt natural. We also felt that array predicates expressed in the DSL were more readable compared to the same predicates expressed in the native Why3 notation. As the DSL is relatively small and syntactically close to the original box-based diagrams, we believe it could be picked up by new users in a short time.

7.2 Limitations and Future Work

While developing these methods and tools within the fields of visualization, runtime execution, decomposition and DSLs to facilitate the formal verification of software, we also identified some limitations to be addressed in future work.

Runtime invariant evaluation provides a mechanism for animating and executing the program in the verification platform. The runtime execution may provide hints to the users of the faulty statement if they provide the verification platform with an input that reveals the erroneous case. In a simplified example, assuming a linear search algorithm for finding the maximum in a sequence, and a variable named *max* storing the value of the maximum in the portion of the array that has been searched so far, if a statement assigns a smaller element to the variable *max*; the user cannot detect the incorrect statement if the provided input is a monotonically non-decreasing sequence. Therefore, the users provided input—to exercise their program with—affect the detection—or missing—the errors by the evaluator. There has been proposals in development of input generation from the specification that depicts the faulty behavior [70] which we did not include in our earlier work.

The case study we presented on the decomposition technique in Paper II has not been implemented in actual code in the domain. Potential issues in the transformation of the refined machines into executable code has thus not been considered. Event-B relies on refinement principles in order to extend the models. Each refinement provides the model with more detailed implementation. Refining each of the decomposed models loosens the connection between the abstract variables and their invariants by introducing new variables or replacing the abstract ones in which were guaranteeing the correctness of the abstract model's invariants. To introduce less gluing invariants between the models which increases the complexity of composition, we have mainly refined one of the decomposed models in the case study while keeping the rest intact with only minor extensions. To be able to assess the validity of the approach, a larger case study needs to be carried out with and without the decomposition. To asses the benefit of the approach, the workload on the underlying automatic provers needs to be evaluated in addition to the composition effort in comparison with a single model verification. Furthermore, the proportion of conditions that are automatically proved vs. require human interaction could be compared between the two approaches.

The DSL we introduced in formal verification in Paper V provides a compact diagrammatic syntax for expressing predicates over arrays, such as invariants and pre- and postconditions. However, it is so far only developed for linear data structures such as arrays. To be able to fully utilize the power of the approach, non-linear data structures such as trees are yet to be supported by the DSL. Extending the language requires generalizing it into a new representation which is abstract enough to replace the existing language and if not more, equally as expressive. Another impediment in introducing a DSL is resistance by the developer community to adopt a new language. As any other language, DSLs needs to be learnt independently from the programming or logical languages of the framework. Improved documentation and integration of the DSL into other verification frameworks would help to alleviate this issue.

Bibliography

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, USA, 1st edition, 2010.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, Nov. 2010.

[3] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1–2):1–28, Jan. 2007.

[4] M. Altenhofen and A. D. Brucker. Practical Issues with Formal Specifications. In S. Kowalewski and M. Roveri, editors, *Formal Methods for Industrial Critical Systems*, pages 17–32, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] M. Avalle, A. Pironti, and R. Sisto. Formal Verification of Security Protocol Implementations: A Survey. *Formal Aspects of Computing*, pages 1–25, 2012.

[7] R.-J. Back. Program Construction by Situation Analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.

[8] R.-J. Back. Invariant Based Programming. In S. Donatelli and P. S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, pages 1–18, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[9] R.-J. Back. Invariant Based Programming: Basic Approach and Teaching Experiences. *Form. Asp. Comput.*, 21(3):227–244, May 2009.

[10] R.-J. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. *Distributed Computing*, 3(2):73–87, 1989.

[11] R.-J. Back and K. Sere. Superposition Refinement of Reactive Systems. *Formal Aspects of Computing*, 8(3):324–346, May 1996.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[13] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[14] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development. `www.agilemanifesto.org`, 2001.

[15] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[16] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *Workshop on Intermediate Verification Languages*, 2011.

[17] F. Boniol and V. Wiels. The Landing Gear System Case Study. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, pages 1–18, Cham, 2014. Springer International Publishing.

[18] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, USA, 1998.

[19] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland, 1992.

[20] T. Bourke, M. Daum, G. Klein, and R. Kolanski. Challenges and Experiences in Managing Large-Scale Proofs. In J. Jeuring, J. A. Campbell, J. Carette, G. D. Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012.

[21] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee*, Brussels 39, Belgium, April 1970. NATO, Science Committee.

[22] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[23] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[24] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[25] S. A. Cook. The Complexity of Theorem-Proving Procedures. In M. A. Harrison, R. B. Banerji, and J. D. Ullman, editors, *STOC*, pages 151–158. ACM, 1971.

[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[27] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, Testing, and Animating PVS Specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar. 2001.

[28] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] M. Davis, G. Logemann, and D. W. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, 1962.

[30] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[31] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[33] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[34] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

[35] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[36] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 2006. Available at `http://yices.csl.sri.com/tool-paper.pdf`.

[37] Eclipse Foundation. Eclise Modeling Framework (EMF), 2020. Project website `http://www.eclipse.org/emf/`.

[38] J. Eriksson. *Tool-Supported Invariant-Based Programming*. Ph.d. thesis, Turku Centre for Computer Science, Finland, 2010.

[39] D. Faitelson, R. Heinrich, and S. Tyszberowicz. Supporting Software Architecture Evolution by Functional Decomposition. pages 435–442, 01 2017.

[40] J.-C. Filliâtre. One Logic to Use Them All. In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[41] J.-C. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italy, Mar. 2013. Springer.

[42] R. W. Floyd. Assigning meanings to programs. *Proc. of Symposium of Applied Mathematics*, 19:19–32, 1967.

[43] R. W. Floyd. Toward Interactive Design of Correct Programs. In C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, editors, *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, pages 7–10. North-Holland, 1971.

[44] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2011.

[45] T. Grimm, D. Lettnin, and M. Hübner. A Survey on Formal Verification Techniques for Safety-Critical Systems-on-chip. *Electronics*, 7(6), 2018.

[46] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[47] J. Harrison. Formal Proof – Theory and Practice. *Notices of the American Mathematical Society*, 55:1395–1406, 2008.

[48] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, pages 440–451, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[49] W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Ohio State University, Columbus, OH, USA, 1995. UMI Order No. GAX96-12193.

[50] T. S. Hoang. An Introduction to the Event-B Modelling Method. In A. Romanovsky and M. Thomas, editors, *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, July 2013.

[51] T. S. Hoang and J.-R. Abrial. Event-B Decomposition for Parallel Programs. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z*, pages 319–333, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[52] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[53] C. A. R. Hoare. Proof of a Program: FIND. *Commun. ACM*, 14(1):39–45, Jan. 1971.

[54] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 256 pages.

[55] W. E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, SE-3:266–278, 1977.

[56] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software Verification Platform. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, pages 301–306, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[57] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[58] D. Jackson, I. Schechter, and H. Shlyahter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 730–733, New York, NY, USA, 2000. ACM.

[59] P. James. *Designing Domain Specific Languages for Verification and Applications to the Railway Domain*. PhD thesis, Swansea University, 2014.

[60] M. Jastram and P. M. Butler. *Rodin User's Handbook: Covers Rodin v.2.8*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2014.

[61] X. Jia. An Approach to Animating Z Specifications. In *19th International Computer Software and Applications Conference (COMPSAC'95), August 9-11, 1995, Dallas, Texas, USA*, pages 108–113, 1995.

[62] C. B. Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Ann. Hist. Comput.*, 25(2):26–49, 2003.

[63] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[64] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Springer, 2000.

[65] M. Kaufmann and J. S. Moore. Some key research problems in automated theorem proving for hardware and software verification. *RACSAM*, 98(1):181–195, 2004.

[66] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607 – 639, 1965.

[67] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[68] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[69] D. Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 229–321. Oxford University Press, Inc., New York, NY, USA, 1994.

[70] H. Liu and H. B. K. Tan. Automated Verification and Test Case Generation for Input Validation. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, page 29–35, New York, NY, USA, 2006. Association for Computing Machinery.

[71] D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging Proof Obligations from Atelier B Using Multiple Automated Provers. In J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, pages 238–251, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[72] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2 edition, 1997.

[73] R. Milner and R. Bird. The Use of Machines to Assist in Rigorous Proof. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 312(1522):411–422, Oct. 1984.

[74] C. Morgan. Auxiliary Variables in Data Refinement. *Inf. Process. Lett.*, 29(6):293–296, 1988.

[75] C. Morgan. The Refinement Calculus. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992.*, pages 3–52, 1992.

[76] NASA Langley Research Center. NASA Langley PVS Libraries, 2013. Available from `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[77] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali. A survey on theorem provers in formal methods. *CoRR*, abs/1912.03028, 2019.

[78] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, Oct. 1979.

[79] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[80] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[81] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6(4):319–340, Dec 1976.

[82] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK, 1992. Springer-Verlag.

[83] S. Owre and N. Shankar. Abstract Datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1997.

[84] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report NASA/CR-1999-209321, NASA Langley Research Center, Aug. 1999.

[85] S. Owre and N. Shankar. The PVS Prelude Library. Technical Report SRI-CSL-03-1, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar. 2003.

[86] S. Owre and N. Shankar. A Brief Overview of PVS. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 22–27, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[87] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 2020.

[88] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 2020.

[89] L. C. Paulson. Set Theory for Verification: I. From Foundations to Functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.

[90] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[91] J. C. Reynolds. Programming with Transition Diagrams. In D. Gries, editor, *Programming Methodology*, pages 153–165. Springer-Verlag, New York, 1978.

[92] J. C. Reynolds. Reasoning About Arrays. *Commun. ACM*, 22(5):290–299, May 1979.

[93] J. C. Reynolds. *The Craft of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[94] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[95] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[96] A. Ruiz-Delgado, D. Pitt, and C. Smythe. A Review of Object-Oriented Approaches in Formal Methods. *The Computer Journal*, 38(10):777–784, 01 1995.

[97] M. Schmalz. The Logic of Event-B. Technical report, ETH, Department of Computer Science, Zurich, 2011.

[98] M. Shroff and R. B. France. Towards a Formalization of UML Class Structures in Z. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651, Aug 1997.

[99] R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition in Event-B. Technical report, University of Dusseldorf, 2010. Proceedings of the 2nd RODIN User and Developer Workshop.

[100] K. Slind and M. Norrish. A Brief Overview of HOL4. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[101] C. Snook and M. Butler. UML-B: A Plug-in for the Event-B Tool Set. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z*, pages 344–344, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[102] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., USA, 1989.

[103] J. Symons and J. K. Horner. Why There is no General Solution to the Problem of Software Verification. *Foundations of Science*, 25(3):541–557, Sept. 2020.

[104] T. Thesing, C. Feldmann, and M. Burchardt. Agile versus Waterfall Project Management: Decision Model for Selecting the Appropriate Approach to a Project. In *ProjMAN - International Conference on Project Management 2020, Procedia Computer Science 181 (2021)*, pages 746 – 756, 2021.

[105] A. Turing. Checking a large routine. In M. Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[106] M. H. Van Emden. Programming with Verification Conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, March 1979.

[107] N. S. Voros, C. F. Snook, S. Hallerstede, and K. Masselos. Embedded System Design Using Formal Model Refinement: An Approach Based on the Combined Use of UML and the B Language. *Des. Autom. Embed. Syst.*, 9(2):67–99, 2004.

[108] H. Wang. Toward Mechanical Mathematics. *IBM Journal of Research and Development*, 4(1):2–22, 1960.

[109] J. Woodcock. Verified Software Grand Challenge. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, pages 617–617, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[110] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), Oct. 2009.

[111] Q. Xu, W. P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2):149–174, Mar 1997.

ISBN 978-952-12-4214-4