Jonatan Wiik

Contract-Based Design of Dataflow Programs

Åbo Akademi
University

Contract-Based Design of Dataflow Programs

Jonatan Wiik

Computer Engineering
Faculty of Science and Engineering
Åbo Akademi University
Åbo, Finland, 2023

Supervisors

Docent Marina Waldén
Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 3, 20500 Åbo, Finland

Docent Pontus Boström
Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 3, 20500 Åbo, Finland

Reviewers

Associate Professor Lionel Morel
INSA-Lyon (Université de Lyon)
6 avenue des Arts
F-69621 Villeurbanne, France

Associate Professor Jani Boutellier
School of Technology and Innovation
University of Vaasa
Yliopistonranta 10, 65200 Vaasa, Finland

Opponent

Professor Klaus Schneider
Department of Computer Science
RPTU Kaiserslautern-Landau
P.O. Box 3049, 67653 Kaiserslautern, Germany

ISBN 978-952-12-4339-4 (printed)
ISBN 978-952-12-4340-0 (digital)
Painosalama, Åbo, Finland 2023

Abstract

Quality and correctness are becoming increasingly important aspects of software development, as our reliance on software systems in everyday life continues to increase. Highly complex software systems are today found in critical appliances such as medical equipment, cars, and telecommunication infrastructure. Failures in these kinds of systems may have disastrous consequences. At the same time, modern computer platforms are increasingly concurrent, as the computational capacity of modern CPUs is improved mainly by increasing the number of processor cores. Computer platforms are also becoming increasingly parallel, distributed and heterogeneous, often involving special processing units, such as graphics processing units (GPU) or digital signal processors (DSP) for performing specific tasks more efficiently than possible on general-purpose CPUs. These modern platforms allow implementing increasingly complex functionality in software. Cost efficient development of software that efficiently exploits the power of this type of platforms and at the same time ensures correctness is, however, a challenging task.

Dataflow programming has become popular in development of safety-critical software in many domains in the embedded community. For instance, in the automotive domain, the dataflow language Simulink has become widely used in model-based design of control software. However, for more complex functionality, this model of computation may not be expressive enough. In the signal processing domain, more expressive, dynamic models of computation have attracted much attention. These models of computation have, however, not gained as significant uptake in safety-critical domains due to a great extent to that it is challenging to provide guarantees regarding e.g. timing or determinism under these more expressive models of computation.

Contract-based design has become widespread to specify and verify correctness properties of software components. A contract consists of assumptions (preconditions) regarding the input data and guarantees (postconditions) regarding the output data. By verifying a component with respect to its contract, it is ensured that the output fulfils the guarantees, assuming that the input fulfils the assumptions.

While contract-based verification of traditional object-oriented programs has been researched extensively, verification of asynchronous dataflow programs has not been researched to the same extent. In this thesis, a contract-based design framework tailored specifically to dataflow programs is proposed. The proposed framework supports both an extensive subset of the discrete-time Simulink synchronous language, as well as a more general, asynchronous and dynamic, dataflow language.

The proposed contract-based verification techniques are automatic, only guided by user-provided invariants, and based on encoding dataflow programs in existing, mature verification tools for sequential programs, such as the Boogie guarded command language and its associated verifier. It is shown how dataflow programs, with components implemented in an expressive programming language with support for matrix computations, can be efficiently encoded in such a verifier. Furthermore, it is also shown that contract-based design can be used to improve runtime performance of dataflow programs by allowing more scheduling decisions to be made at compile-time. All the proposed techniques have been implemented in prototype tools and evaluated on a large number of different programs. Based on the evaluation, the methods were proven to work in practice and to scale to real-world programs.

Sammanfattning

Kvalitet och korrekthet blir idag allt viktigare aspekter inom mjukvaruutveckling, då vi i allt högre grad förlitar oss på mjukvarusystem i våra vardagliga sysslor. Mycket komplicerade mjukvarusystem finns idag i kritiska tillämpningar så som medicinsk utrustning, bilar och infrastruktur för telekommunikation. Fel som uppstår i de här typerna av system kan ha katastrofala följder. Samtidigt utvecklas kapaciteten hos moderna datorplattformar idag främst genom att öka antalet processorkärnor. Därtill blir datorplattformar allt mer parallella, distribuerade och heterogena, och innefattar ofta specialla processorer så som grafikprocessorer (GPU) eller signalprocessorer (DSP) för att utföra specifika beräkningar snabbare än vad som är möjligt på vanliga processorer. Den här typen av plattformar möjligör implementering av allt mer komplicerade beräkningar i mjukvara. Kostnadseffektiv utveckling av mjukvara som effektivt utnyttjar kapaciteten i den här typen av plattformar och samtidigt säkerställer korrekthet är emellertid en mycket utmanande uppgift.

Dataflödesprogrammering har blivit ett populärt sätt att utveckla mjukvara inom flera områden som innefattar säkerhetskritiska inbyggda datorsystem. Till exempel inom fordonsindustrin har dataflödesspråket Simulink kommit att användas i bred utsträckning för modellbaserad design av kontrollsystem. För mer komplicerad funktionalitet kan dock den här modellen för beräkning vara för begränsad beträffande vad som kan beksrivas. Inom signalbehandling har mera expressiva och dynamiska modeller för beräkning attraherat stort intresse. De här modellerna för beräkning har ändå inte tagits i bruk i samma utsträckning inom säkerhetskritiska tillämpningar. Det här beror till en stor del på att det är betydligt svårare att garantera egenskaper gällande till exempel timing och determinism under sådana här modeller för beräkning.

Kontraktbaserad design har blivit ett vanligt sätt att specifiera och verifiera korrekthetsegenskaper hos mjukvarukomponeneter. Ett kontrakt består av antaganden (förvillkor) gällande indata och garantier (eftervillkor) gällande utdata. Genom att verifiera en komponent gentemot sitt konktrakt kan man bevisa att utdatan uppfyller garantierna, givet att indatan uppfyller antagandena.

Trots att kontraktbaserad verifiering i sig är ett mycket beforskat område, så har inte verifiering av asynkrona dataflödesprogram beforskats i samma utsträckning. I den här avhandlingen presenteras ett ramverk för kontraktbaserad design skräddarsytt för dataflödesprogram. Det föreslagna ramverket stödjer så väl en stor del av det synkrona språket Simulink med diskret tid som ett mera generellt asynkront och dynamiskt dataflödesspråk.

De föreslagna kontraktbaserade verifieringsteknikerna är automatiska. Utöver kontraktets för- och eftervillkor ger användaren endast de invarianter som krävs för att möjliggöra verifieringen. Verifieringsteknikerna grundar sig på att omkoda dataflödesprogram till input för existerande och beprövade verifieringsverktyg för sekventiella program så som Boogie. Avhandlingen visar hur dataflödesprogram implementerade i ett expressivt programmeringsspråk med inbyggt stöd för matrisoperationer effektivt kan omkodas till input för ett verifieringsverktyg som Boogie. Utöver detta visar avhandlingen också att kontraktbaserad design också kan förbättra prestandan hos dataflödesprogram i körningsskedet genom att möjliggöra flera schemaläggningsbeslut redan i kompileringsskedet. Alla tekniker som presenteras i avhandlingen har implementerats i prototypverktyg och utvärderats på en stor mängd olika program. Utvärderingen bevisar att teknikerna fungerar i praktiken och är tillräckligt skalbara för att också fungera på program av realistisk storlek.

Acknowledgements

Completing this thesis has been a long journey. While the cover of the thesis bears my name only, many people and organizations have contributed in different ways to make this thesis possible.

First of all, I would like to express my gratitude to my supervisors Pontus Boström and Marina Waldén for your invaluable support, which has been instrumental in completing this thesis. In particular, I would like to thank Pontus for all the encouraging guidance, support, and fruitful discussions during the initial stages of this research work. I would like to thank Marina especially for your patience and untiring encouragement during the last years of my studies, when the time and energy I was able to allocate to this thesis was very limited.

I would like to thank Associate Professor Lionel Morel and Associate Professor Jani Boutellier for finding time to review my thesis and providing valuable comments and insights that helped me improve the quality of the thesis. I would also like to thank Professor Klaus Schneider who has kindly agreed to act as opponent at the public defence of this thesis.

I would like to thank my co-author Johan Ersfolk, especially for the abundant amount of time used to discuss and implement ideas that eventually came to constitute an essential part of this thesis. It was a great pleasure to work with you. Moreover, I would like to thank all the people at the IT department of Åbo Akademi, who over the years contributed to it being an inspiring place to work and conduct research.

I would like to acknowledge the funding I have received from Åbo Akademi, Academy of Finland, Business Finland, Nokia Foundation, and Ulla Tuomisen Säätiö. The funding I received was essential to be able to focus on this research full-time for several years.

I am grateful to my current colleagues at Awake.AI who paved way for me to finalize this thesis beside a full-time job. In particular, I am grateful to Jussi Poikonen, whose encouraging attitude was greatly helpful in finding inspiration to finalize the thesis.

Finally, this journey had not been possible without the continuous support from my family and close friends. Thank you all!

November, 2023
Jonatan Wiik

List of original publications

I P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, 15(4):1141–1161, 2016

II J. Wiik and P. Boström. Contract-based verification of MATLAB-style matrix programs. *Formal Aspects of Computing*, 28(1):79–107, 2016

III J. Wiik and P. Boström. Specification and automated verification of dynamic dataflow networks. In A. Cimatti and M. Sirjani, editors, *Software Engineering and Formal Methods, SEFM 2017*, volume 10469 of *LNCS*, pages 136–151. Springer International Publishing, 2017

IV J. Wiik, J. Ersfolk, and M. Waldén. A contract-based approach to scheduling and verification of dynamic dataflow networks. In P. Derler and S. Gao, editors, *16th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2018*, pages 1–10. IEEE, 2018

List of other co-authored publications

1. J. Wiik and P. Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. In S. Merz and J. Pang, editors, *Formal Methods and Software Engineering, ICFEM 2014*, volume 8829 of *LNCS*, pages 396–412. Springer International Publishing, 2014

2. J. Wiik and P. Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. Technical Report 1107, TUCS, 2014

3. J. Wiik and P. Boström. Contract-based specification and verification of dataflow programs. In L. Aceto and A. Ingolfsdottir, editors, *Proceedings of 27th Nordic Workshop on Programming Theory, NWPT 2015*. Reykjavik University, 2015

4. J. Wiik and P. Boström. Specification and automated verification of dynamic dataflow networks. Technical Report 1170, TUCS, 2016

5. J. Ersfolk, P. Boström, V. Timonen, J. Westerholm, J. Wiik, O. Karhu, M. Linjama, and M. Waldén. Optimal digital valve control using embedded GPU. In J. Uusi-Heikkilä and M. Linjama, editors, *Proceedings of the 8th Workshop on Digital Fluid Power*. Tampere University of Technology, 2016

Contents

Part I

Research summary

Chapter 1

Introduction

Mathematical approaches for assessing program correctness have been studied since the 1960s [32]. Using these formal techniques, software can be mathematically proven to adhere to desirable properties or to lack undesirable properties for all possible inputs. For a long time, practical adoption of these techniques have been limited by the huge amount of work and expertise that is required to carry out such proofs on programs of realistic size. Over the last years, however, the performance of tools used to automatically carry out this kind of proofs have improved rapidly, making it increasingly feasible to prove also real world software.

In the context of object-oriented software, design-by-contract has become a widespread concept used to specify and verify intended behaviour of software modules. Design-by-contract was introduced as a term by Bertrand Meyer [65, 64] in the 1980s during the design of the Eiffel programming language. Using contracts, properties are provided as preconditions and postconditions stating properties that a program should adhere to. In practice, the preconditions and postconditions are often provided as extra annotations in the program code. Research on the topic has resulted in mature automatic verification tools for many widely used programming languages, including Java [15, 18], C# [6] and C [49].

While object-oriented programming has gained widespread adoption, the paradigm is limited in that it does not exploit parallelism in programs very naturally. Instead, programmers are typically responsible for mapping computations to threads, which is an error-prone and difficult task. This is a limitation of growing magnitude, as today the performance of modern computing platforms are improved to a large extent by increasing the number of computation cores or by utilising special-purpose hardware, such as graphical processing units or digital signal processors to perform certain tasks efficiently. Furthermore, objects do not necessarily provide the most suitable abstraction in all problem domains.

In the embedded domain, dataflow programming has become a widely used paradigm to implement so called reactive systems. As opposed to a traditional transformational system, which computes an output given some input, a reactive system is continuously interacting with its environment. The history of dataflow programming dates back to the 1960s when Petri nets [68] were introduced. Further research was carried out by Dennis [23] and Kahn [48] in the 1970s. In dataflow programming, a program is described as a graph where the nodes, which are commonly called actors, blocks or processes, perform computations, and the edges, which are commonly called channels, signals or buffers, describe the communication of data between the nodes. The nodes communicate exclusively over the edges. This restricted model of computation has several compelling advantages, e.g.:

1. It provides a natural way to express reactive systems.
2. It naturally makes parallelism in the program explicit, which in turn makes it easy to map different actors to processing units.
3. It simplifies reasoning about the programs significantly, as components communicate exclusively over statically defined channels.

The dataflow programs considered in this thesis can broadly be split into two classes; synchronous[1] (or synchronous-reactive) and asynchronous dataflow. In synchronous languages, computations are carried out in synchronous rounds triggered by a signal. This signal is often, but not necessarily, ticks of a clock. In asynchronous dataflow, on the other hand, communication between actors is buffered and the actors can execute (fire) whenever required tokens are available on the incoming buffers.

The objective of this thesis is to propose a contract-based framework supporting specification, verification and scheduling of dataflow programs. The framework supports both asynchronous models of computation as well as an extensive subset of the synchronous language Simulink. Additionally, the framework supports verification of actors implemented in an expressive programming language, with MATLAB-style built-in support for matrix operations.

1.1 Contribution

The most central contributions of this thesis can be summarized as follows:

[1]This is not the synchronous dataflow (SDF) proposed by Lee and Messerschmitt [52, 53], which actually belongs to asynchronous dataflow according to this classification.

1. Specification constructs for both discrete-time Simulink as well as more general, asynchronous and dynamic dataflow models of computation.

2. A unified automatic verification approach for both synchronous and asynchronous dataflow, based on mapping dataflow programs to sequential programs and verifying them using well-established techniques.

3. A static type and matrix shape inference approach, as well as a contract-based verification approach for an expressive imperative programming language with built-in support for MATLAB-style matrix operations.

4. An approach to scheduling of dataflow programs utilising information provided in contracts, allowing scheduling of dataflow programs not belonging to any well-established statically schedulable model of computation.

5. Tool-support for all the implemented verification and scheduling techniques and evaluation on example programs of realistic size.

1.2 Research problem

The focus of this thesis is on specification constructs and automated verification techniques for dataflow programs. Dataflow languages are widely used in the development of embedded software systems, which often are safety-critical with high reliability requirements. This justifies the extra effort involved with using more rigorous specification and verification techniques. This leads to the following research questions:

1. What are good specification constructs for different types of dataflow programs? It is, for instance, important that the specifications are expressive enough to express properties of interest, but at the same time it is also essential that they are intuitive and usable to engineers.

2. How can the specification constructs and dataflow programs be efficiently encoded in an automatic verifier? The encoding should take into account a sufficiently large subset of the considered dataflow language and at the same time, for usability, the verification conditions should be automatically dischargeable with as few user-provided invariants as possible.

1.3 Research methods

The research in this thesis was carried out by first identifying representative programs with properties typical for the kind of programs of interest. Intuitive specification languages for expressing these types of properties were then developed. After this, techniques to verify the program with respect to the specifications were developed. The last stage was to implement the techniques in tools for validation on realistic programs.

When developing automated verification techniques, prototype tool implementations are essential to judge if the technique is well suited for automation. This is primarily due to the fact that one has to consider subtle limitations of proof tools and other backend tools. Therefore, great emphasis in the research of this thesis has been put on implementing all proposed techniques in automated verification tools.

1.4 Thesis structure

The research conducted as part of this thesis is reported in four peer-reviewed papers, which are included in Part II of this thesis. Part I is a research summary of the papers. The research summary is divided into 8 chapters and is aimed at linking the research papers together. In this section, the content of each research paper as well as the research summary is briefly outlined. The contribution by the thesis author in each of the paper is also described.

Paper I

Paper I presents an approach to contract-based verification of discrete-time Simulink models based on translation into functionally equivalent SDF (static dataflow) networks. Well-established scheduling methods are then used to obtain sequential programs which are verified to be correct with respect to the contracts. To formally argue for the correctness of the translation from Simulink to SDF, a semantics for Simulink based on Kahn Process Networks is presented. A refinement-based approach is used to verify correctness with respect to contracts and the approach is evaluated on a pressure relief controller for a digital hydraulics system. The approach was implemented in a prototype verification tool named VerSÅA.

Author's contribution The initial work for the paper was done in the scope of the author's master's thesis [78], for which the co-author acted as supervisor. The work was further extended for the paper and the work, including development of the verification technique, improved tool implementation, and writing of the paper, was split evenly between the two authors.

Paper II

Paper II extends Paper I by adding support for matrix computations. The main contributions of the paper are identification of a large subset of Embedded MATLAB that can be efficiently encoded in verifiers, as well as a static type and shape inference approach for the considered subset. Additionally, two approaches to encoding MATLAB's builtin matrix functions and operators in a verifier is presented and they are evaluated on examples. Both the Boogie intermediate verification language as well as direct encoding in the SMT solver are evaluated as backends on a number of examples. Finally, k-induction in conjunction with matrix operations is also evaluated as a way of reducing the number of required loop invariant annotations.

Author's contribution The work on development of the specification and verification technique, tool implementation, and writing of the paper, was split evenly between the two authors. The evaluation on examples was performed by the author.

Paper III

Paper III extends Paper I by considering specification and verification of more general, dynamic and asynchronous dataflow networks, for which a static schedule cannot be obtained in the general case. SDF is a subset of the dataflow networks considered in this paper. Hence, also the SDF graphs obtained from the Simulink translation in Paper I can be handled in this approach. The contributions of the paper include a method to specify the behaviour of dataflow networks based on the reaction of the network to individual input tokens. An encoding into the Boogie intermediate verification language is also presented, as well as a method to automatically generate invariants for a common class of actors. The approach was implemented in a prototype verification tool named Actris[2] and evaluated on a number of examples.

Author's contribution The initial research idea is due to the thesis author. The early development of the specification and verification technique was done jointly by the two authors. The thesis author was

[2]https://github.com/jwiik/actris-verifier

responsible for refining the idea, developing the detailed specification language, the verifier encoding and the invariant generation technique, as well as implementing tool-support for the verification approach. The research paper was written by the thesis author with feedback from the co-author.

Paper IV

Paper IV is an extension of Paper III also considering compile-time scheduling based on contracts. The contract language proposed in Paper III is reused in this work. The obtained schedules are then also utilised in verification to reduce the number of needed invariant annotations. The contributions of the paper include showing that contracts can aid the scheduling of dataflow networks and enable scheduling of actors that do not conform to any well-known statically schedulable model of computation. A scalable hierarchical scheduling method based on contracts, which can be used to obtain schedules optimal with respect to a cost function is also presented. Finally, a method that verify actors and networks by using static schedules is also presented. The approach was implemented as an extension of the Actris tool developed for Paper III.

Author's contribution The initial idea and the scheduling technique was co-developed by the thesis author and one of the co-authors. The verification approach and tool implementation was developed by the thesis author. The paper was written by the thesis author with feedback from the other co-authors.

This thesis

In this thesis it is described how the four research papers together form a unified contract-based framework for both the synchronous dataflow language Simulink as well as asynchronous dataflow languages. This is achieved by using the translation from Simulink to SDF presented in Paper I to map Simulink to asynchronous dataflow. Based on this translation, Simulink models can, essentially, be verified using the more general approach for dynamic dataflow programs presented in Paper III. It is also outlined on an informal level how the contracts for Simulink models presented in Paper I can be mapped into the contracts for asynchronous dataflow presented in Paper III. Together with the contributions of Paper II and Paper IV a versatile framework for contract-based design of dataflow programs is achieved.

Chapter 2

Dataflow programming

In the dataflow programming paradigm, a program is described as a static network of actors connected via channels, forming a directed graph. The actors communicate exclusively through these explicit channels. Actors implement algorithms which they execute on the data on the incoming channels and output the result on the outgoing channels. Different variants of dataflow programs have proven useful in a number of different domains. In e.g. the automotive domain the language Simulink[1] has become a de facto standard for describing control software. Also in the signal processing domain dataflow graphs are widely used, as they provide a natural way of expressing operators and the flow of data between them.

The dataflow models discussed in this thesis can be divided into two distinct classes of models of computation. These classes are *asynchronous* and *synchronous* models of computation. In asynchronous dataflow, actors fire in response to tokens being available on the input channels, while in a synchronous model of computation actors fire synchronously in response to a signal. This synchronous signal is often a clock, but can also be some other event. In asynchronous dataflow, the channels are asynchronous order-preserving channels, i.e. buffers, while in synchronous dataflow, the value on a channel is updated each synchronous round. In this thesis, the focus is on Simulink as a synchronous language and on a variant of the Cal Actor Language (CAL) [26], as a more general, asynchronous language.

In this chapter, the dataflow programming languages in focus are first described on an informal level. After this, the models of computation relevant to the thesis and the relationship between them are reviewed through these languages.

[1]https://www.mathworks.com/help/simulink/

2.1 Dataflow languages

In this section, the two dataflow languages focused on in this thesis are informally presented. These are the synchronous language Simulink and the dataflow language RVC-CAL [63], a version of the original CAL language [26] that was adopted in an MPEG standard [63].

A dataflow programming language can be coarsely divided into two parts; an actor language which is used to declare actors, networks and their interconnection, and a host language which is used to implement algorithms in actors. The actor language is often graphical, and is largely independent of the host language, while the host language can be, for instance, a standard imperative programming language.

2.1.1 Simulink

Simulink has become one of the most widely used tools in model-based design of control systems. In some industries, e.g. the automotive, it has even become a de facto standard. Simulink has a user-friendly modelling notation and good simulation tools for testing and validation of control software together with models of the controlled plant.

Simulink is a graphical programming language based on hierarchical dataflow diagrams. The diagrams consists of blocks[2] connected by signals. The signals describe the flow of data between the blocks. While Simulink can be used to describe a wide range of dynamic systems, this thesis considers an extensive subset of *discrete-time* Simulink models. Such models can be considered an instance of the synchronous language model of computation, which means that each block is evaluated periodically with a fixed sampling period. A model where not every block is updated with the same period is considered to be *multi-rate*.

As an example, consider the Simulink diagram in Figure 2.1. It models a 4th order finite impulse response (FIR) filter. It additionally has a *Rate Transition* block named *RT* which is used to modify the rate at which the filter output y is updated. As the output sampling rate is different than the input sampling rate, this diagram describes a *multi-rate* system. In the diagram *D1*, *D2* and *D3* are blocks of type *Unit delay*, which delay their input with one sampling step. The *Gain* blocks multiply their input by a constant given as parameter. Connected to the *Gain* blocks is an *Add* block which outputs the sum of the input signals. Assuming that the input sampling rate T_x is faster than or equal to the output sampling rate T_y, i.e. $T_y = nT_x$ for an integer $n > 1$, the diagram in Figure 2.1 calculates the following difference equation, where $y(nk)$

[2]In the context of Simulink, the term block is commonly used for actors.

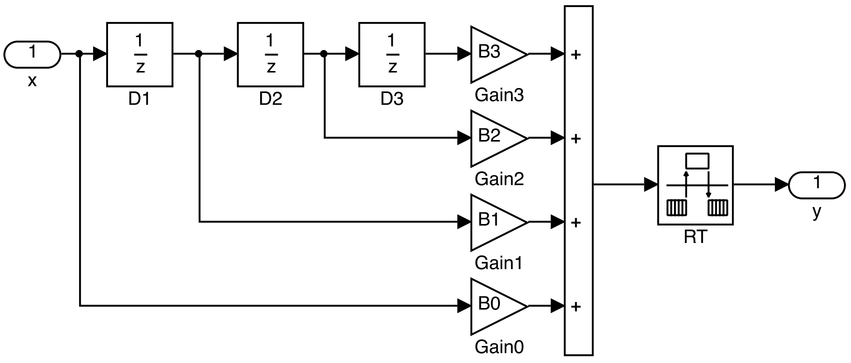


Figure 2.1: A Simulink subsystem modelling a FIR filter with an additional rate transition on the output y.

denotes the filter output at time nk:

$$y(nk) = b_0x(nk) + b_1x(nk-1) + b_2x(nk-2) + b_3x(nk-3)$$

where b_i is the value of the impulse response at the i'th instance, given as parameters to the *Gain* blocks. Note that signal values are not defined for a time t that is not a multiple of the sampling period of the signal.

The basic blocks in Simulink diagrams are fairly primitive operators available via a built-in block library. Diagrams constructed from these basic blocks can then be hierarchically grouped into subsystems, which form reusable modules implementing more complex functionalities. In Simulink it is common practice to use block diagrams to describe functionality down to the level of, e.g., single arithmetic operators, but this is not the case for all dataflow languages. In CAL, for instance, it is common to implement fairly complex algorithms inside a single actor using the host language. While not used to the same extent, this can be accomplished also in Simulink, using blocks containing user-defined code, written in a subset of the MATLAB language.

2.1.2 CAL

CAL [26] is a programming language aimed at describing asynchronous dynamic dataflow programs. In this thesis a language similar to the RVC-CAL [63] variant of CAL is considered. RVC-CAL can be considered a subset of the original CAL language [26], adopted by MPEG as part of the Reconfigurable Video Coding (RVC) standard [63]. One of the most significant differences between RVC-CAL and the original CAL language is that RVC-CAL has explicit type annotations. The actor language

(a)

```
actor Sum int x1, x2 ⟹ int y:
  action x1:[i], x2:[j] ⟹ y:[i+j] end
end
```

(b)

```
actor Delay(int y0) int x ⟹ int y:
  initialize ⟹ y:[y0] end
  action x:[i] ⟹ y:[i] end
end
```

(c)

```
actor Merge int x1, x2 ⟹ int y:
  bool switch := false

  a1:
  action x1:[i] ⟹ y:[i]
  guard switch = false
  do
    switch := true;
  end

  a2:
  action x2:[i] ⟹ y:[i]
  guard switch = true
  do
    switch := false;
  end

end
```

Figure 2.2: Examples of CAL actors.

considered throughout this thesis is essentially RVC-CAL extended with contract annotations.

In CAL, a program consists of a static network of actors, where an actor is a stateful operator having a set of inports, a set of outports, a set of state variables, and a set of actions. An action is enabled or disabled based on the amount and values of tokens available on the inports and the actor state. An actor executes by firing an enabled action, which updates the actor state and produces a number of tokens on the outports.

To concretise this discussion, consider the example actors in Figure 2.2. The actor *Sum* has two inports named *x1* and *x2*, one outport named *y*, and a single action. The action consists of the input patterns `x1:[i]` and `x2:[j]`, which means that, when fired, the action consumes one token on each of the inports *x1* and *x2* and assigns them the variable names *i* and *j*, respectively. The action also has the output pattern `y:[i+j]`, which means that it outputs a token containing the sum $i + j$ on the outport y.

The actor *Delay* has a special initialisation action declared with the keyword **`initialize`**, which outputs an initial token with value *y0* on the outport. An initialisation action is only executed once, when the actor is initialised. As the name suggest, the actor *Delay* effectively delays the data on its input, by prepending the additional token *y0* to its output sequence.

The actor *Merge* forwards the tokens on the two input ports *x1* and *x2* to the output port in turns. In other words, the output stream of the actor *Merge* consists of interleaved tokens from the inports. The actor actions have guards on the actor state variable *switch* to control which action to execute next.

It is worth noting that it is easy to construct inconsistent networks by connecting actors like those in Figure 2.2. For instance, if an actor of type *Sum* only receives input on one of the inports, it will never become enabled. On the other hand, if it continuously receives more tokens on port *x1* than on *x2* the buffer on *x1* will grow indefinitely. It is also worth noting that it is easy to express an actor with non-deterministic behaviour. Consider, for instance, removing the guards from the actor *Merge*. If there are tokens available on both inputs, such an actor would non-deterministically fire one of the actions. As discussed later on, one of the goals of the verification framework presented in this thesis is to statically check for the absence of this kind of inconsistencies, as well as non-determinism.

2.2 Models of computation

In the previous section the dataflow languages CAL and Simulink were introduced through examples. In this section dataflow programming is viewed from a more formal perspective, by considering different models of computation (MoC). A dataflow MoC defines the behaviour of a dataflow network. In other words, a MoC is an interpretation of a dataflow network, which defines in which order and how the actors are executed. In this section several common dataflow MoCs relevant to this thesis and the relations between them are reviewed. The section starts by introducing Kahn Process Networks, which are then used as a foundation to formalise the discussion on the relationships between other, more practical MoCs.

2.2.1 Kahn process networks

Kahn process networks (KPN) [48] differ from the programs expressible in the CAL-based actor language introduced in the previous section in that it lacks the notion of firing. KPN is, nevertheless, useful as a concept to provide semantics for dataflow programs. A KPN is formed of a set of independent processes which communicate through unbounded unidirectional FIFO channels. Each channel x can be considered to carry a stream $\langle x_0, x_1, x_2, \dots \rangle$, where each x_i is a data token. In a Kahn process, reads from a channel are blocking while writes are non-blocking.

Kahn gave KPN a denotational semantics [48] where a process F with m inputs and n outputs is defined as a function $F\colon S^m \to S^n$, where S^i denotes the set of i-tuples of streams. In the semantics, Kahn processes are required to be continuous in the sense explained below. Consider a prefix order relation defined on streams x and y:

$$x \preceq y \tag{2.1}$$

denoting that x is a prefix of y. In other words, the k first elements of y are equal to the k elements of x. Streams are considered equal if they are prefixes of each other. Now consider a chain w of streams, where each stream is comparable using $\preceq$ and let $\sqcup w$ denote the least upper bound of w. A process F is continuous if for every such chain $\sqcup F(w)$ exists and:

$$F(\sqcup w) = \sqcup F(w) \tag{2.2}$$

A continuous process is also monotone [48], which means that:

$$x \preceq y \Rightarrow F(x) \preceq F(y) \tag{2.3}$$

A KPN can be described by a set of equations where each equation is defined by a Kahn process. If each process is continuous, the set of equations has a unique least fixpoint solution which describes the histories of tokens which appeared on the network channels.

That a process F is monotone means, in essence, that previous output tokens of F will not change in response to receiving additional input tokens. In addition to this, a continuous process F will not wait for an infinite amount of input tokens before it produces output tokens.

2.2.2 Dataflow process networks

The dataflow process network (DPN) model of computation was formalised by Lee and Parks [51, 54]. They view DPN as a special case of KPN, where the streams are built up from sequences of firings. Formally, a DPN actor can be considered as a pair $\{f, R\}$, where $f\colon S^m \to S^n$ is a firing function and $R \subset S^n$ is the set of firing rules. The firing rules are expressed as patterns in the form of finite sequences describing when the actor is allowed to fire. A firing rule is satisfied if it forms a prefix of the unconsumed tokens on the corresponding input port.

A Kahn process F based on $\{f, R\}$ can be constructed based on repeated firings of f, where $\perp$ denotes the empty sequence and $s.s'$ denotes the concatenation of the sequences s and s':

$$F(x) = \begin{cases} f(r).F(x') & \text{if there exists an } r \in R \text{ such that } x = r.x' \\ \perp & \text{otherwise} \end{cases} \tag{2.4}$$

Lee and Parks showed [51, 54] that sufficient conditions for a DPN actor to be continuous, as defined in (2.2), is that the process is *functional* and that the set of firing rules is *sequential*. Here, functional means that the actor lacks side effects and that the output tokens are a pure function of the input tokens consumed during that firing. Sequential means that firing rules can be tested in a predefined order using only blocking reads.

The requirement that actors be functional seems to exclude actors with state. However, Lee and Parks [51, 54] argue that actor state can be expressed simply as feedback loops on the top-level network. Hence, actors with state expressed in the language considered here can be considered functional. In an actor language such as CAL, it is, possible to express firing rules that are not sequential. However, it can be ensured that actors are sequential by checking that the firing rules are mutually exclusive. Hence, actors described in the CAL language can be restricted to the DPN MoC by requiring that the firing rules are mutually exclusive.

2.2.3 Dynamic dataflow

A dataflow network where computations are described with as a set of firing rules which depend on state variables or data tokens that are evaluated at run-time, can be considered to belong to a MoC called Dynamic dataflow (DDF). However, DDF is vaguely defined in the literature and seems to, in many cases, be used as a name for asynchronous dataflow programs that do not correspond to any other more restrictive model of computation. In this thesis the definition given in [77] is used. According to this definition a firing rule can be any Boolean expression and a DDF program is also allowed to be nondeterministic. Based on the definition, DDF is more general than DPN and essentially all programs expressible in the CAL language introduced in Section 2.1.2 introduced language can be classified as DDF. Consequently, DDF is also a superset of DPN.

2.2.4 Static dataflow

Static dataflow (SDF) [52, 53] was proposed by Lee and Messerschmitt in 1987. The name *synchronous dataflow* is commonly used for SDF, but the term static dataflow is used here to distinguish it from the MoC of synchronous languages such as Simulink, which are synchronous in a different sense. In SDF the term synchronous refers to the fact that the size of communication buffers can be computed statically at compile-time, while in the synchronous MoC synchronous refers to that computations are synchronized by a clock signal. Based on this classification, SDF is an asynchronous model of computation.

SDF is a subset of DPN where actors consume and produce a fixed amount of tokens each time they fire. Formally, this means that in (2.4) each firing rule $r \in R$ is of the form $\langle *, \dots, * \rangle$, where $*$ denotes a wildcard matching any token. This means that, e.g., $\langle *, * \rangle$ is a prefix of any stream with at least two tokens.

In practice, the actor language can be syntactically limited to SDF by restricting actors to have only one non-initializing action without guard. SDF also requires actors to be stateless, hence restricting SDF networks to only have state in the form of delay tokens on feedback loops.

The restrictions SDF imposes allow networks to be statically scheduled. This means that SDF networks can be compiled into a sequence of actor firings, which, when executed, will return the network buffers to the initial state. Scheduling is discussed in more detail in Chapter 6.

2.2.5 The synchronous model of computation

The models of computation of synchronous dataflow languages such as Simulink and Lustre differ significantly from the asynchronous models of computation described above. In a synchronous dataflow MoC computation is carried out in synchronous rounds. These rounds are typically, but not necessarily, triggered by a clock.

Discrete-time multi-rate Simulink has a synchronous model of computation where each actor is given a sampling period at which it is triggered. A channel in a Simulink model that is updated with a sampling period T_s can be considered a stream $x_{T_s} = \langle x(0), x(1), \dots \rangle$, where $x(k)$ is the value at the channel at time kT_s. In Simulink the output $y_{T_s}(k)$ of an actor f is computed based on the input $u_{T_s}(k)$ at the same sampling period. Additionally, Simulink actors can be parametrised and are also allowed to have state. In its most general form, a Simulink actor with equal input and output sampling periods can be described by:

$$\begin{aligned}
&y_{T_s}(k) = f(c, x_{T_s}(k), u_{T_s}(k)) \\
&x_{T_s}(k+1) = f(c, x_{T_s}(k), u_{T_s}(k)) \\
&x_{T_s}(0) = x_{init}
\end{aligned} \tag{2.5}$$

Here c is a set of parameters, u is the set of input streams, y the set of output streams, and x is the set of streams modelling the state vector. The function f gives the k:th values on the output streams y while g gives the next values on the streams x representing the state.

In Simulink, only special *Rate Transition* actors are allowed to alter sampling periods, i.e. have an output sampling period different than the input sampling period. These actors cannot be described by (2.5). Stream sampling operators can be introduced to handle rate transitions (Paper I).

Definition 1. *Assume $T_2 = nT_1$ for $n \in \mathbb{N}_+$. Further, let / denote integer division. Then the upsampling operator $\uparrow_{T_1}$ is defined as:*

$$\forall k \in \mathbb{N} \cdot (\uparrow_{T_1} x_{T_2})(k) = x_{T_2}(kT_1/T_2)$$

The downsampling operator $\downarrow_{T_2}$ is defined as:

$$\forall k \in \mathbb{N} \cdot (\downarrow_{T_2} x_{T_1})(k) = x_{T_1}(kT_2/T_1)$$

Based on these operators, it is possible to define the behaviour of the Simulink rate transitions. In case the input sampling period T_i is greater than the output sampling period T_o, i.e. $T_i = nT_o$, the rate transition actor is defined as:

$$\begin{aligned}
&y_{T_o}(k) = \uparrow_{T_o}(x_{T_i}(k)) \\
&x_{T_i}(kT_o/T_i + 1) = u_{T_i}(kT_o/T_i) \\
&x_{T_i}(0) = x_{init}
\end{aligned} \tag{2.6}$$

In case $nT_i = T_o$:

$$y_{T_o}(k) = \downarrow_{T_o}(u_{T_i}(k)) \tag{2.7}$$

Simulink models can be given semantics based on KPN (Paper I). The least fixpoint solution for a Simulink model consisting of streams $x_1, \ldots, x_m$ can then be expressed analogously to KPNs [48] as:

$$\begin{aligned}
&x_{T,i}(0) = x_{init} \\
&x_{T,i}(k+1) = \tau_i(x_{T,i}(k), \ldots, x_{T,m}(k))
\end{aligned} \tag{2.8}$$

for $i \in 1, \ldots, m$, where each τ_i is based on the continuous operators given by the actors, i.e. Simulink functional blocks. x_{init} denotes the initial values on streams and are supplied as parameters to Simulink actors. However, it is not possible to construct τ directly, as the sampling periods have to be taken into account. In (2.8), T is the base period of the model. For a Simulink model with rates $T_1, \ldots, T_n$, T is defined as:

$$T = \gcd(T_1, \ldots, T_n)$$

The base period gives the time points at which a stream can change values. For a stream x_{T_x} it is then possible to construct the stream at the base period using $\uparrow_T(x_{T_x})$.

In Section 3.4 a mapping from discrete-time Simulink to SDF is presented. This mapping is essential to enable handling Simulink models using the same specification and verification framework used for asynchronous networks.

Chapter 3

Language definition

In this chapter, the dataflow programming language considered in this thesis, and introduced through examples in Chapter 2, is defined more precisely. A dataflow programming language can generally be split into two parts; an actor language and a host language. The actor language is used to declare actors, networks and their interconnection, while the host language is used to implement algorithms in actor actions. The actor language is virtually independent of the choice of host language. In practice it is common that the actor language is, at least partially, a graphical language. For instance, both in Simulink and RVC-CAL the interconnections between actors and networks can be defined graphically.

The actor language part of the proposed dataflow programming language is essentially an extensive subset of RVC-CAL [63], which has been extended with specification constructs like preconditions, postconditions and invariants. RVC-CAL, in turn, is a subset of the CAL Actor Language [26], which was standardised as part of an MPEG standard [63]. As host language, a subset of Embedded MATLAB (Paper II) is used. This language is very similar to the original imperative host language of RVC-CAL. Disregarding some minor syntactical differences, Embedded MATLAB could even be considered an extension of the RVC-CAL host language, with support for matrix functions and operators.

In the end of the chapter, it is additionally covered how Simulink models can be expressed in the proposed dataflow programming language, by mapping a subset of the Simulink language to SDF.

3.1 Actor language

In this section the grammar of the actor language considered in this thesis is defined. The grammar is given in Figure 3.1. In this grammar, S denotes a program statement in the host language, while e denotes an expression in the host language. Furthermore, A is an assertion used in the contracts defined later in this thesis. The elements *Contract*, *Inv*

$$
\begin{array}{lll}
\mathit{Prog} & ::= & (\mathit{ActorDecl} \mid \mathit{NwDecl})^* \\
\mathit{ActorDecl} & ::= & \textbf{actor}\ id\langle \mathit{VarDecl}^* \rangle\ \mathit{PortDecl}:\ \ \mathit{ActorMem}^*\ \textbf{end} \\
\mathit{NwDecl} & ::= & \textbf{network}\ id\langle \mathit{VarDecl}^* \rangle\ \mathit{PortDecl}:\ \ \mathit{NwMem}^*\ \textbf{end} \\
\mathit{PortDecl} & ::= & \mathit{VarDecl}^* \Longrightarrow \mathit{VarDecl}^* \\
\mathit{ActorMem} & ::= & \mathit{VarDecl} \mid \mathit{Inv} \mid \mathit{Contract} \mid \mathit{Action} \mid \mathit{InitAction} \\
\mathit{NwMem} & ::= & \mathit{Inv} \mid \mathit{Contract} \mid \mathit{Entities} \mid \mathit{Structure} \\
\mathit{Action} & ::= & \textbf{action}\ (id\colon [id^*])^* \Longrightarrow (id\colon [e^*])^*\ (\textbf{guard}\ e)^*\ \mathit{Spec}\ (\textbf{do}\ S)^?\ \textbf{end} \\
\mathit{InitAction} & ::= & \textbf{initialize}\ \Longrightarrow (id\colon [e^*])^*\ \mathit{Spec}\ (\textbf{do}\ S)^?\ \textbf{end} \\
\mathit{Contract} & ::= & \textbf{contract}\ (id\colon n)^* \Longrightarrow (id\colon n)^*\ (\textbf{guard}\ A)^*\ \mathit{Spec}\ \textbf{end} \\
\mathit{Entities} & ::= & \textbf{entities}\ (id = id\langle e^* \rangle)^*\ \textbf{end} \\
\mathit{Structure} & ::= & \textbf{structure}\ (id(.id)^? \longrightarrow id(.id)^?)^*\ \textbf{end} \\
\mathit{Inv} & ::= & \textbf{contract invariant}\ A \mid \textbf{action invariant}\ A \\
\mathit{Spec} & ::= & (\textbf{requires}\ A \mid \textbf{ensures}\ A)^* \\
\mathit{VarDecl} & ::= & \mathit{type}\ id
\end{array}
$$

Figure 3.1: The grammar of the actor language.

and *Spec* are used for specification and will be discussed later. In the grammar, $\langle$ and $\rangle$ are used for concrete parentheses to differentiate them from parentheses used for grouping to describe the grammar. Furthermore, x^* and $x^?$ are used to denote any number of repetitions of x and 0 or 1 repetitions of x, respectively. The elements S and e stands for statements and expressions in the chosen host language.

Based on the grammar, a program consists of a set of actor declarations and network declarations. An actor declaration consists of set of variable declarations, a set of invariants, a set of contracts, a set of actions and optionally an initialisation action. A network consists of a set of invariants, a set of contracts, an entities block and a structure block. The entities and structure blocks declares the networks components and their interconnection, respectively. An action consists of a set of input patterns, a set of output patterns, a set of guards, a specification (contract) and possibly an actor body.

3.2 Host language

While the choice of host language is largely independent of the actor language, it is still important for e.g productivity to choose an appropriate host language for the problem domain in question. In this thesis, an extensive subset of Embedded MATLAB (EML) is considered as host language. EML is different to most general purpose languages in that it has builtin support for matrix computations. This means that e.g. the addition operator can be used both on scalar values and on matrices. For some examples, consider a matrix a:

$$a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

The following piece of code:

```
b := abs(a);
c := 5-a;
```

assigns the variables b and c as follows:

$$b = \begin{bmatrix} abs(a_{11}) & abs(a_{12}) \\ abs(a_{21}) & abs(a_{22}) \end{bmatrix} \quad c = \begin{bmatrix} 5 - a_{11} & 5 - a_{12} \\ 5 - a_{21} & 5 - a_{22} \end{bmatrix}$$

The functions *abs* and the operator '$-$' are said to work *element-wise* on matrices.

In MATLAB, an element-wise function or operator $f(x, y)$ is defined if x and y have equivalent shapes, or if either of x and y is scalar. The resulting matrix has the same shape as the input matrix with the larger shape. Many typical mathematical functions that are usually defined for scalars are in MATLAB generalised to matrices in the form of an element-wise function. There are, however, some important matrix functions in MATLAB that are not element-wise. One such example is the function *sum* that can be considered to belong to a class of functions that are *collapsing*. A collapsing function collapses vectors to scalars and other matrices to a row vector. In MATLAB, a vector is a matrix where either the number of columns or the number of rows is equal to 1, while a scalar is a matrix where both the number of columns and the number of rows is equal to 1. Consider the example code below involving the collapsing functions *sum* and *prod*:

```
d := sum(a);
e := prod(d);
```

which results in the following values being assigned to e and d:

$$d = \begin{bmatrix} a_{11} + a_{21} & a_{12} + a_{22} \end{bmatrix} \quad e = (a_{11} + a_{21}) \cdot (a_{12} + a_{22})$$

Hence, the behaviour of the function *sum* depends on the shape of the input. This is also the case, e.g., for the multiplication operator $*$, which denotes standard matrix multiplication if both arguments are matrices, but element-wise multiplication if one of the arguments is scalar.

S	$::=$ $v := e \mid$	Assigment
	$\texttt{if}\ e\ S_1\ \texttt{else}\ S_2\ \texttt{end} \mid$	If-statement
	$S_1 ; S_2 \mid$	Sequential composition
	$\texttt{while}\ e\ (\texttt{invariant}\ A)^*\ S\ \texttt{end} \mid$	While loop
	$\texttt{for}\ id = e\ (\texttt{invariant}\ A)^*\ S\ \texttt{end}$	For loop

$e ::=$	
$e_1\ (+ \mid - \mid * \mid / \mid .* \mid ./)\ e_2 \mid$	Arithmetic expression
$e_1\ (\wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow)\ e_2 \mid$	Logical expression
$e_1\ (= \mid \neq \mid < \mid > \mid \geq \mid \leq)\ e_2 \mid$	Relational expression
$(\forall \mid \exists)\ (type\ id)^*\ \cdot\ e \mid$	Quantified expression
$\neg e \mid -e \mid$	Unary operators
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{end} \mid$	Conditional expression
$e_1 \langle (e_2 \mid :)\ (e_3 \mid :)^? \rangle \mid$	Matrix accessor
$id\langle e_1, \ldots, e_n \rangle \mid$	Function call
$id \mid$	Identifier
$n \mid$	Numeric literal
$[e_{11}, \ldots, e_{1n}; \ldots ; e_{m1}, \ldots, e_{mn}] \mid$	Matrix literal
$c_1{:}c_2 \mid c_1{:}c_2{:}c_3 \mid$	Range
$\texttt{true} \mid \texttt{false}$	Boolean literal

Figure 3.2: Grammar of the host language with builtin support for matrix operators.

Every datatype in the considered host language is essentially a matrix. A matrix type is considered to consist of an *intrinsic type*, e.g. **double**, **int**, or **bool** and a shape. The syntax $\langle m, n \rangle$ denotes a shape with m rows and n columns. The syntax $\mathsf{matrix}(t, \langle m, n \rangle)$ is then used to express a matrix type with intrinsic type t and shape $\langle m, n \rangle$.

In Figure 3.2, the grammar of statements S and the grammar of expressions e for the considered host language are given. Note that the MATLAB operators $.*$ and $./$ denotes element-wise multiplication and element-wise division, respectively.

The presented host language based on EML is syntactically very similar to the imperative host language of RVC-CAL, but there are some small differences. For instance are normal parentheses used to access matrix elements in EML, while square brackets are used to access array elements in RVC-CAL. However, these differences are not significant to the specification and verification methods presented in this thesis.

3.3 Type system

The focus of this thesis is on static verification with respect to contracts. As discussed later, the proposed verification approach relies on utilising matrix shapes. As a consequence, matrix shapes need to be statically determined and checked. For this purpose, a static type and matrix shape inference method was developed as part of this thesis (Paper II).

The proposed type and shape inference method is based on providing pre-defined type and shape signatures for all supported builtin MATLAB functions. Consider a simple program:

```
a := [1, 2; 3, 4];
b := 2;
c := a + b;
```

To determine the type and shape of c in this program, we need signatures for the element-wise + operator as well the function *sum*. In this case these type signature declares that, since a is matrix of shape $\langle 2,2\rangle$ and b is a scalar, the output of $a+b$ is also a matrix of shape $\langle 2,2\rangle$. Additionally, the type signature also declares that the intrinsic type (e.g. **int** or **double**) is the same as the intrinsic type of the inputs. More formally this can be stated using the following type signature:

$$\forall\ t \sqsubseteq_t \mathbf{numtype} \cdot \Pi\ m_1, n_1, m_2, n_2 \cdot \mathsf{matrix}(t, \langle m_1, n_1\rangle) \times \mathsf{matrix}(t, \langle m_2, n_2\rangle) \\ \rightarrow \mathsf{matrix}(t, \max_{\mathrm{s}}(\langle m_1, n_1\rangle, \langle m_2, n_2\rangle))$$

In this type signature, $\forall$ denotes universal quantification over intrinsic types, while Π denotes universal quantification over shapes. Hence, in the example above, the type signature for the minus operator is universally quantified over both intrinsic type and shape. The operator $\sqsubseteq_t$ denotes bounds on quantification. Hence, the quantified variable t is bounded to intrinsic types that are subtypes of **numtype**. The type **numtype** is assumed to be an abstract super-type of any numeric type. The variables m_1, n_1, m_2 and n_2 are quantified shape variables. In the type signature, the result shape of the operator is given by a shape function $\max_{\mathrm{s}}$, which is defined as follows:

$$\max_{\mathrm{s}}(\langle m_1, n_1\rangle, \langle m_2, n_2\rangle) = \begin{cases} \langle m_1, n_1\rangle & \text{if } m_1 = m_2 \text{ and } n_1 = n_2 \\ \langle m_1, n_1\rangle & \text{if } m_2 = n_2 = 1 \\ \langle m_2, n_2\rangle & \text{if } m_1 = n_1 = 1 \\ \langle \infty, \infty\rangle & \text{otherwise} \end{cases}$$

Here ∞ is used to denote an error. A program is type-correct only if there are no shapes of infinite size. The function $\max_{\mathrm{s}}$ determines the type for all binary element-wise functions. There are also corresponding functions for e.g. collapsing functions and matrix multiplication.

This type system is discussed in detail in Paper II. It should, however, be noted that the host language considered here is explicitly typed, while Embedded MATLAB which is considered in Paper II is implicitly typed. For an implicitly typed language, types and shapes have to be inferred, while it for an explicitly typed language is enough to check type correctness.

3.4 Translating Simulink models

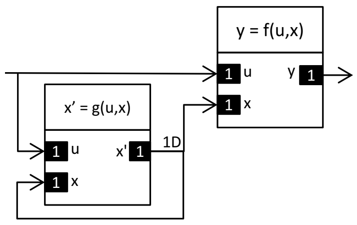
Discrete-time multi-rate Simulink as defined in Section 2.2.5 can be mapped to SDF in a manner such that each sample in a Simulink stream x corresponds to a buffer position $\mathsf{sdf}(x)$ in the SDF representation:

$$x_{T_s}(k) = \mathsf{sdf}(x)(k) \tag{3.1}$$

In this section, it is reviewed how this can be done for different types of Simulink actors. A detailed description of the translation as well as proofs of the correctness of the translation is given in the research paper (Paper I).

Functional blocks Consider a general Simulink functional actor as defined by (2.5). An actor in this general form can be represented in SDF using two actors; one for computing the function f and one for computing the function g. The interconnection of these two actors are illustrated in Figure 3.3. Here the boxes with black background denotes ports and the number inside these boxes denotes the rate, i.e. the number of tokens consumed or produced on the port when the actor fires. The source code of the actors is also given to illustrate how they could be described in the actor language. In the source code, t denotes a type in the type system of the chosen host language and u, x and y denote the set of all inports, outports and block memories, respectively. In Figure 3.4 there are concrete examples how the Simulink functional blocks used in the FIR filter in Figure 2.1 can be mapped to the actor language.

It should be noted that the behaviour of a Simulink functional block could also be described in a single actor with state in the actor language. However, this would not adhere to the traditional SDF definition, which only allows state in the form of delay tokens on buffers. A pure SDF representation was chosen here, as well-established frameworks and scheduling methods developed for SDF can then be reused directly.



```
actor F $t_1$ U, $t_2$ X $\Longrightarrow$ $t_3$ Y:
  action U:[$u$], X:[$x$] $\Longrightarrow$ Y:[ $f(u,x)$ ] end
end

actor G(int $x_0$) $t_1$ U, $t_2$ X $\Longrightarrow$ $t_2$ X':
  initialize $\Longrightarrow$ X':[$x_0$] end
  action U:[$u$], X:[$x$] $\Longrightarrow$ X':[ $g(u,x)$ ] end
end
```

Figure 3.3: SDF representation of a general Simulink functional block.

Rate transition blocks Rate transition blocks are used to connect parts of a model operating at different sampling periods. All blocks except rate transitions are required to have the same output and input sampling periods. In Simulink, the behaviour of a rate transition depends on if the output sampling period is longer than or shorter than the input sampling period. The behaviour of a slow-to-fast rate transition is given by (2.6). SDF blocks implementing this behaviour are given in Figure 3.5a. The blocks F and G are assumed to be connected in the same way as illustrated in Figure 3.3. Note that x^n in Figure 3.5a denotes outputting a token with value x n times, where n is given by $n = T_i/T_o$ assuming that T_i is the input sampling period and T_o is the output sampling period.

The behaviour of a fast-to-slow rate transition is given by (2.7) and an SDF representation of this block is given in Figure 3.5b. Here the block G produces $n-1$ initial tokens x_0, where $n = T_o/T_i$. The block F consumes n tokens when it fires and outputs the last token it consumed.

Multi-rate subsystems Subsystems are in Simulink used to hierarchically structure the diagrams. They hence have the same purpose as networks in the actor language. Simulink subsystems are either virtual or atomic. Virtual subsystems are merely used for visual grouping and have no semantic effect. Subsystems are here assumed to be atomic. The period of a subsystem T_{sub} is the shortest period of its subcomponents, i.e. the greatest common divisor of all subcomponent sampling periods.

(a)

```
actor Gain(t g) t U ⟹ t Y:
  action U:[u] ⟹ Y:[g∗u] end
end
```

(b)

```
actor Add t U1, t U2, t U3, t U4 ⟹ t Y:
  action
    U1:[u1], U2:[u2], U3[u3], U4:[u4] ⟹ Y:[u1+u2+u3+u4]
  end
end
```

(c)

```
actor UnitDelayF t X ⟹ t Y:
  action X:[x] ⟹ Y:[x] end
end

actor UnitDelayG(t x0) t U ⟹ t X':
  initialize ⟹ X':[x0] end
  action U:[u] ⟹ X':[u] end
end
```

Figure 3.4: SDF representations of the Simulink blocks in the FIR filter. The actor in (a) outputs the input multiplied with a constant g. The actor in (b) outputs the sum of its four inputs. The actor in (c) delays its input by one sampling period, i.e. the output $y(n)$ becomes the input $u(n-1)$.

This also means that all inputs and outputs have to be converted to the subsystem period T_{sub}. This is achieved by inserting rate transition actors before subsystem inports and after subsystem outports as illustrated in Figure 3.6. Here *S2F* corresponds to the rate transition in Figure 3.5c, which is equivalent to the the rate transition in Figure 3.5a except that it does not delay the input. The fast-to-slow rate transition, *F2S*, denotes the rate transition described in Figure 3.5b. In Figure 3.6, $k_i = T_i/T_{sub}$ and $k_o = T_o/T_{sub}$, where T_i and T_o are the sampling periods of inport i and outport o, respectively.

Input blocks and output blocks In Simulink, special input blocks and output blocks are used in subsystems to receive input and export output. These have counterparts also in asynchronous dataflow, but the rates should also be converted to the original rates of the signals. Hence, the opposite rate transitions to those connected to the ports on the level above are connected to the input actors and output actors, as illustrated in Figure 3.6.

(a)

```
actor F $t$ X $\Longrightarrow$ $t$ Y:
  action X:[$x$] $\Longrightarrow$ Y:[$x^n$] end
end
actor G($t$ $x_0$) $t$ U $\Longrightarrow$ $t$ X':
  initialize $\Longrightarrow$ X':[$x_0$] end
  action U:[$u$] $\Longrightarrow$ X':[$u$] end
end
```

(b)

```
actor F $t$ X $\Longrightarrow$ $t$ Y:
  action X:[$x_1, \ldots, x_n$] $\Longrightarrow$ Y:[$x_n$] end
end
actor G($t$ $x_0$) $t$ U $\Longrightarrow$ $t$ X':
  initialize $\Longrightarrow$ X':[${x_0}^{n-1}$] end
  action U:[$u$] $\Longrightarrow$ X':[$u$] end
end
```

(c)

```
actor F $t$ U $\Longrightarrow$ $t$ Y:
  action U:[$u$] $\Longrightarrow$ Y:[$u^n$] end
end
```

Figure 3.5: SDF representations of different types of rate transition blocks: (a) a slow-to-fast rate transition, (b) a fast-to-slow rate transition, and (c) a slow-to-fast rate transition without delay. Here x^n is used to denote a sequence of n tokens with value x.

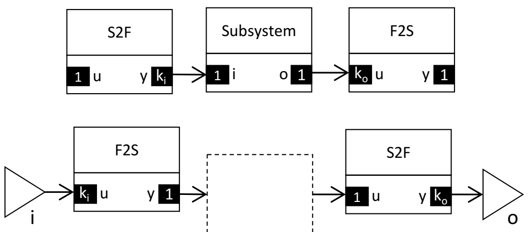


Figure 3.6: The SDF representation of Simulink subsystem interface. The dashed box represents the actual content (components) of the subsystem

Chapter 4

Specification

In this thesis, the main focus is on specification and verification of dataflow programs using contracts. The contracts considered state functional properties that the specified component should adhere to. The contracts presented are traditional assume-guarantee contracts, consisting of preconditions specifying legal inputs and postconditions specifying legal outputs.

In this chapter, the contract formats for both Simulink, proposed in Paper I, and the actor language, proposed in Paper III, are reviewed. A mapping of the Simulink contracts to the more general actor language contracts is also discussed.

4.1 Contracts for dataflow networks

In accordance with the KPN-based semantics discussed in Chapter 2, dataflow channels can be considered as streams. Hence, each channel c in a dataflow network can be viewed as a stream $\langle c_0, c_1, \dots \rangle$ of tokens. The contract format proposed describes the response of a dataflow component to a finite sequence of input tokens as a finite sequence of output tokens. These finite sequences form windows over the input channels and output channels. In the contract format the keyword **requires** is used to denote preconditions and **ensures** is used to denote postconditions. These keywords have become common practice in contracts for object-oriented languages. More precisely, the contract format proposed is defined as follows:

Definition 2 (Contract)**.** *A contract:*

$$C\colon \textbf{contract}\ x\colon n \Longrightarrow y\colon m\ \textbf{guard}\ G\ \textbf{requires}\ P\ \textbf{ensures}\ Q\ \textbf{end}$$

defines a contract with label C, which specifies that, given n input tokens on port x conforming to $G \wedge P$, the component outputs m tokens on port y that conforms to Q.

Here the difference between the guard G and the precondition P is that the contract is *enabled* when G is satisfied, while P is required to hold for any input that enables the contract. A component can have more than one contract, but the contract guards must then be mutually exclusive. In other words, at most one contract can be enabled by any input data. Viewed differently this means that contracts essentially specify different modes of the component, which are triggered based on input tokens.

The contract C describes the behaviour of the component over a window of size n on the input and a window of size m on the output. From now on a window over the channels described by a contract are referred to as the *contract window*. The contract window is a central concept in the specification and verification technique proposed in this thesis. As discussed in more detail in Chapter 5, the proof of correctness with respect to contract is, in essence, done inductively by showing that a component satisfies the contract by considering an arbitrary contract window.

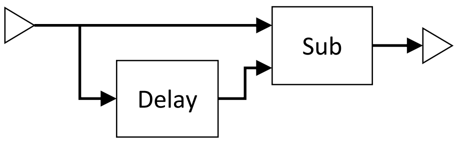
To make the discussion more concrete, consider the network N in Figure 4.1. The functionality of this network is that it subtracts the current input with the previous input. A contract C describing the network is included in the code listing in Figure 4.1. In the contract conditions, the operator $\bullet$ is used to conveniently refer to tokens consumed or produced during a contract window. To discuss the semantics of the contracts, the bullet operator $\bullet(c)$ needs to be defined.

Definition 3 (Bullet). $\bullet(c)$ *for a channel* c *is the total number of tokens that had been consumed on* c *before the current contract window.*

For convenience, the argument can be left out when $\bullet$ is used in a channel index, i.e. $c[\bullet]$ is synonymous to $c[\bullet(c)]$. It is also possible to, for instance, refer to the last token produced during the previous contract window using an offset of -1 with $\bullet$, i.e. $c[\bullet - 1]$. Positive offsets can be used to refer to additional tokens produced inside the contract window.

For the purpose of illustrating contract features, the example contract C in Figure 4.1 requires that inputs are greater than 0 and that each input is greater than the previous one. The example contract also has a postcondition stating that the output on y is always non-negative. Additionally, the second postcondition states that starting from the second output, the output is the difference between input in the considered contract window and in the previous window. Verifying the network N with respect to its contract C means ensuring that the postconditions are always satisfied, assuming that the input satisfies the preconditions.

It is worth noting that a contract describes the behaviour of a component in a similar way as actor actions do. A contract could even be considered as a form of composite action, which consists of a sequence of actions on the hierarchical level below.



```
actor Delay(int y0) int x ⟹ int y:
  initialize ⟹ y:[y0] end
  action x:[i] ⟹ y:[i] end
end

actor Sub int x1, int x2 ⟹ int y:
  action x1:[i], x2:[j] ⟹ y:[i−j] end
end

network N int x ⟹ int y:

  C: contract x:1 ⟹ y:1
    requires 0 ≤ x[•]
    requires 0 < •(x) ⟹ x[•−1] ≤ x[•]
    ensures 0 ≤ y[•]
    ensures 0 < •(y) ⟹ y[•] = x[•]  −  x[•−1]
  end

  contract invariant tokens(c,1)
  action invariant c[0] = 0
  action invariant 0 ≤ c[•]

  entities
    del = Delay(0);
    sub = Sub();
  end

  structure
    a: x ⟶ del.x;
    b: x ⟶ sub.x1;
    c: del.y ⟶ sub.x2;
    d: sub.y ⟶ y;
  end
end
```

Figure 4.1: An example dataflow network and a contract describing its functionality.

$$A \quad ::= \quad A_1 \wedge A_2 \mid \textbf{tokens}(id, e) \mid e$$

Figure 4.2: The grammar of the assertion language.

4.1.1 Invariants

The proof of correctness with respect to contracts is done inductively on contract windows. To enable inductive proofs of properties regarding dataflow components, it is often necessary to annotate the component with a set of invariants. There are two types of invariants: *contract invariants* and *action invariants*. Contract invariants are required to hold between contract windows and are hence used to describe the state of the component between the contract windows. In other words, any sequence of actions executed in a contract window must return the component to a state where the contract invariants hold. Action invariants are required to hold when contract invariants are required to hold and are additionally also required to hold between each action firing inside the contract window. In the case of a network, this means that all firings of subcomponents have to preserve the action invariant.

Consider again the example in Figure 4.1. The network N is annotated with two action invariants and one contract invariant. The construct `tokens(c,1)` used in the contract invariant expresses that there should be exactly 1 token on the channel named c between each contract window. The action invariant `c[0] = 0` expresses that the initial token produced on c is equal to 0. This follows from the initialization action of the actor instance `del`. The second action invariant essentially states that any token produced on channel c is non-negative.

4.1.2 Assertion language

An assertion language is used to express conditions, i.e. the predicates, in contracts. The assertion language should be expressive enough to allow expressing the desired properties, but on the other hand it should also be easy to use for e.g. engineers. In this work, the assertion language A is an extension of the host expression language e defined in Figure 3.2. In essence, the assertion language is equivalent to the expression language e, but complemented with some additional functions not part of the language. This has the advantage that the engineer is typically already familiar with the host language. The functions available are summarised together with short descriptions of their meaning in Table 4.1.

The formal grammar of the assertion language is given in Figure 4.2. According to the grammar, an assertion is a series of conjunctions. The

Table 4.1: Description of specification constructs

Construct	Description
$\bullet(c)$	The number of tokens consumed on c before the current contract window.
rd(c)	Total number of tokens consumed on channel c.
tot(c)	Total number of tokens produced on channel c.
urd(c)	Number of unread tokens on c, i.e. **tot**(c) − **rd**(c).
rd•(c)	Number of tokens consumed on c during the current contract window, i.e. **rd**$(c) - \bullet(c)$.
tot•(c)	Number of tokens produced during the current contract window plus previously unconsumed tokens, i.e. **tot**$(c) - \bullet(c)$.
next(c)	The next token to be consumed on c.
prev(c)	The latest token that was consumed on c.
last(c)	The latest token that was produced on c.
tokens(c, e)	Predicate expressing that the number of unread tokens on c is equal to e.

construct **tokens** can only appear as an independent operand in the conjunction. This limitation helps the verifier in keeping track of for which channels **tokens** has been used. Although, according to the grammar, an assertion can be any expression expressible in the expression language, a well-formed assertion is always of Boolean type, i.e. a predicate. On the contrary, only assertions can contain specification constructs such as $\bullet(c)$, **rd**(c) and **tot**(c).

4.2 Contracts for dataflow actors

In addition to specifying the behaviour of dataflow networks, the contract format introduced in Definition 2 can also be used to specify behaviour of actors.

In a CAL-style language, actor actions are often fired in a predefined order. However, this order is not always made explicit in actor code. Consider the actor *Split16* in Figure 4.3. The functionality of this actor is that it receives 16 tokens on the inport and forwards the first token to the outport *out1* and the remaining 15 tokens to the outport *out2*. A natural way to express this as actor actions is to have a separate action *first* for the first token and another action *rest* for the additional tokens. The actor state, which determines which action is to be executed next, is maintained as a variable *count*. While not immediately apparent from the action implementations, *Split16* has a repeating behaviour where a firing of *first* is always followed by 15 firings of the action *rest*.

```
actor Split16 int in ⟹ int out1, int out2:

  int count := 0;

  contract in:16 ⟹ out1:1, out2:15
    ensures out1[•] = in[•]
    ensures ∀ int i · 0 ≤ i < 15 ⟹ out2[•+i] = in[•+i+1]
  end

  contract invariant count = 0

  first: action in:[ x ] ⟹ out1:[ x ]
    guard count = 0
    do
      count := count + 1;
  end

  rest: action in:[ x ] ⟹ out2:[ x ]
    guard count ≠ 0
    do
      count := count + 1
      if (count = 16)
        count := 0;
      end
    end
  end
end
```

Figure 4.3: Example of an actor with a contract allowing static scheduling.

The contract format defined in Definition 2 can be used to describe repeating behaviours like the one of *Split16*. Consider the contract of *Split16*, defining the amount of input and output tokens as well the relationship between input tokens and output tokens. Essentially, the contract defines a composite action consisting of 16 actor firings. When verifying networks containing actors with contract specifications, the behaviour of the actors can be abstracted by assuming that they behave according to their contract specification. Hence the specification and verification approach is hierarchical and modular. This simplifies the reasoning and makes the verification approach scalable.

4.3 Contracts for Simulink models

The model of computation in Simulink differs from asynchronous dataflow in that actors are triggered by a clock and updated at sampling times. To be intuitive, also contracts for Simulink models should then state properties about signals at certain sampling times. Furthermore, as contracts describe multi-rate subsystems, different sampling periods also have to be taken into account. The proposed contract format for Simulink models (Paper I) is given in Figure 4.4. The contract format is syntactically similar to the one defined for dataflow programs in Definition 2. Syntactically, the largest difference is that the Simulink contracts also declare types for inports and outports. As Simulink is implicitly typed, these explicit type annotations in contracts aid in ensuring type correctness.

In Figure 4.4 an example contract for the subsystem illustrated in Figure 2.1 is given. The precondition states that the input of type *double* should have either value 0 or 1. The postcondition then states that the output should be the decimal value calculated based on the four last inputs. Here the construct **delay**(s, i) is used to refer to the previous value on a signal s, where i is the initial value of that signal, i.e. $s(0)$. The invariant in the contract relates the delays to block memories in the Simulink model. Here the syntax B_X is used to refer to the memory X of block B.

4.4 Mapping Simulink contracts to asynchronous dataflow contracts

As discussed in Section 2.2.5, an extensive subset of discrete-time Simulink can be mapped to static dataflow (SDF). In this section it is outlined on a high level how the Simulink contracts can be mapped into the contract format defined in Definition 2 in a way such that a Simulink subsystem period is mapped to a contract window. Based on this mapping, a unified specification and verification framework for asynchronous dataflow languages and synchronous languages similar to Simulink is achieved.

Consider a Simulink contract C for Simulink subsystem with an inport x and an outport y. Assume further that the subsystem sampling period is T_{sub} and the subsystem repeating period is P. When translated to an SDF representation, the SDF network will consume or produce P/T_{sub} tokens on each inport and outport. It is then possible to translate a port variable p in the Simulink contract according to the

```
contract
  inports (id: type)*
  outports (id: type)*
  parameters (id: type)*
  requires A_P
  ensures A_Q
  invariant A_I
end
```

```
contract
  inports x: double
  outports y: double
  parameters
    B0: double
    B1: double
    B2: double
    B3: double
  ensures
    y =
      B0 * x +
      B1 * delay(x,0) +
      B2 * delay(delay(x,0),0) +
      B3 * delay(delay(delay(x,0),0),0)
  invariant
    delay(x,0) = D1_X ∧
    delay(delay(x,0),0) = D2_X ∧
    delay(delay(delay(x,0),0),0) = D3_X
end
```

Figure 4.4: The proposed Simulink contract format and an example contract for the FIR filter in Figure 2.1.

following:

$$p = \mathsf{sdf}(p)[\bullet - d]$$

where d is the number of delays. This means that, if p is not referenced inside a **delay** construct, then there is no delay and $d = 0$. If p is nested inside n **delay** constructs, then the delay d is n.

However, the upsampling of inports and outports to the subsystem rate in the SDF representation means that the precondition P and postcondition Q have to be quantified over the period. This means that a precondition $A(x)$ takes the following form:

$$\forall k \cdot 0 \le k < P/T_{sub} \implies A(\mathsf{sdf}(x)[\bullet + k])$$

while a postcondition $A(x, y)$ takes the following form:

$$\forall k \cdot 0 \le k < P/T_{sub} \implies A(\mathsf{sdf}(x)[\bullet - d], \mathsf{sdf}(y)[\bullet + k])$$

Considering this mapping more closely reveals some limitations in expressiveness of the Simulink contract format. It is, for instance, not possible to refer to individual sampling points of the input inside a period in the contracts, which the more general contract format for asynchronous dataflow allows using the $\bullet$ operator.

Considering the contract for the FIR filter as an example, it can be mapped to the following actor contract and actor invariants, assuming that $P/T_{sub} = 4$:

```
contract x:4 ⟹ y:4
  ensures ∀ int k · 0 ≤ k ∧ k < 4 ⟹
    y[•+k] = B0 ∗ x[•] + B1 ∗ x[•−1] + B2 ∗ x[•−2] + B3 ∗ x[•−3]
end

invariant tokens(D1_X, 1) ∧ tokens(D2_X, 1) ∧ tokens(D3_X, 1)
invariant D1_X[0] = 0 ∧ D2_X[0] = 0 ∧ D3_X[0] = 0
invariant x[•−1] = D1_X[•] ∧ x[•−2] = D2_X[•] ∧ x[•−3] = D3_X[•]
```

The first and second invariants derives from the block memories of the Simulink *Unit Delay* blocks. The third invariant is derived from the invariant given in the Simulink contract.

Chapter 5

Verification

One of the main targets of this thesis work is to enable automated verification of dataflow programs with respect to contracts. Automated here means that the verification is done automatically only guided by user-provided invariant annotations. The purpose of this chapter is to outline how the programming languages and contracts described in the previous chapters can be converted into efficient input for an automatic verifier.

The proposed verification approach is based on translating the dataflow programs and contracts into input for a well-tested and efficient verification tool. The Boogie [5] verifier was chosen for this purpose, because of its expressive input language and tight integration with the state-of-the-art SMT solver Z3 [21]. The verification problem then essentially becomes converting dataflow programs and accompanying contracts into Boogie input that can be efficiently verified. The input language for Boogie is a sequential programming language, which makes it straightforward to encode many of the host language constructs. However, the matrix functions and operators need special treatment, as there is no native matrix support in Boogie. In addition to this, the main challenge regarding this encoding consists in describing actor networks and their specifications in terms of sequential programming language constructs. In this chapter the mapping of dataflow programs into verifiable sequential program routines is discussed on a mainly informal level, deferring most of the formal details to the research papers.

5.1 From dataflow to sequential program

For restricted models of computation like static dataflow and synchronous languages, well-studied algorithms for conversion of dataflow diagrams to sequential programs exist [53, 52, 74, 17]. These algorithms consist in scheduling actor actions into a static sequence of firings of actor actions. However, for more general models of computation like DPN, this approach is not possible in the general case, because scheduling is a run-time problem and no static schedule can be obtained. This essentially follows from the fact that an actor can be enabled or disabled based on the values of incoming tokens, which are not known at compile-time.

The purpose of dataflow scheduling is to find a sequence of actor firings to allow the network to return to a state corresponding to the one before executing the sequence. This means that, e.g. the number of tokens on channels are the same after executing a schedule. The purpose of these schedules are that they can be repeated indefinitely to execute the network. To concretise the discussion, the example network in Figure 4.1 can again be considered. This network has two actors *Del* and *Sub*. When the network receives one input token both actions become enabled. This means that the network has two possible schedules [*del*, *sub*] and [*sub*, *del*], which both are functionally equivalent. Executing only one of either of the actors would not return the number of tokens on the channels to the original state and are hence not valid schedules.

To extend the idea of sequential schedules to MoCs which are not statically schedulable, we can consider expressing the program as a loop in which each iteration non-deterministically fires an enabled action:

```
Receive input tokens
while (enabled actors)
  Non-deterministically execute an enabled actor action        (5.1)
end
Send output tokens
```

Executing a dataflow diagram then consists in repeated executions of the above program. Since it has been established that, given some restrictions, all schedules are functionally equivalent [54, 51] we can use the above scheme as a basis to verify asynchronous dataflow networks without computing a static schedule, given that we can check for the required restrictions.

If we again consider the example network N in Figure 4.1, it can be expressed according to the scheme above as follows:

```
$\textit{Receive 1 token on x}$
$\textbf{while}$ $((\textit{del enabled}) \lor (\textit{sub enabled}))$
  $\textbf{do}$
    $(\textit{del.a enabled}) \rightarrow \textit{execute del.a}$
    $\mid (\textit{sub.a enabled}) \rightarrow \textit{execute sub.a}$
  $\textbf{end}$
$\textbf{end}$
$\textit{Output 1 token on y}$
```

where the construct $G_1 \rightarrow S_1 \mid G_2 \rightarrow S_2$ denotes non-deterministic choice, meaning that any S_i where G_i is true can be executed. Considering this pseudocode it becomes apparent that there needs to be a method to decide the appropriate number of input tokens. The number of input tokens should allow N to fire in such a way that the network channels are returned to an initial state where no components in N are fireable without receiving additional network input. As discussed earlier, the contract format defined in Definition 2 makes explicit the number of input and output tokens.

It is worth noting that the same strategy can be used to encode not only networks, but also basic actors. In this case the actions executed in the *while* loop are those defined in the actor itself.

5.2 Boogie encoding of actors and networks

Based on the mapping from dataflow to sequential programs introduced in the previous section, it is possible to verify dataflow components using traditional program verification techniques. Consider again a contract C:

```
$C$: contract $x\colon C_x \Longrightarrow y\colon C_y$
  guard $C_{grd}$
  requires $C_{pre}$
  ensures $C_{post}$
end
```

Further assume that there are contract invariants C_{cinv} and action invariants C_{ainv}. On a very high level, to verify a network N with respect to C, the following proof obligations have to be discharged:

1. Executing the initialisation actions of the components in N establishes the C_{cinv} and C_{ainv}.

2. Receiving C_x input tokens on channel x satisfying C_{grd} and C_{pre} preserves the action invariants.

3. Executing the components of N (i.e. the *while* loop) preserves C_{ainv}.

4. Assuming C_{ainv} holds and the loop condition is falsified (i.e. the loop has terminated), there is C_y output tokens on channel y, and C_{cinv}, C_{ainv}, and C_{post} hold.

If a network has more than one contract, the steps 2 and 4 above need to be proven separately for each contract.

However, to efficiently encode the proof obligations in practice, there are many issues that need to be addressed. For instance, dataflow constructs like channels need to be efficiently encoded in the Boogie language.

To encode network channels, an encoding based on Boogie's polymorphic maps was chosen. The encoding is based on the following global map variables:

$$\mathcal{I}\colon \mathbf{ch} \to \mathbf{int} \quad \mathcal{R}\colon \mathbf{ch} \to \mathbf{int} \quad \mathcal{C}\colon \mathbf{ch} \to \mathbf{int} \quad \mathcal{M}\colon (\mathbf{ch}\langle\beta\rangle, \mathbf{int}) \to \beta$$

Here $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{C}$ are maps from channels to integers. The integer $\mathcal{I}[c]$ is the number of tokens that had been consumed on the channel c before the current contract window. The integer $\mathcal{R}[c]$ is the total number of consumed tokens on c, while $\mathcal{C}[c]$ is the total number of tokens produced on c. The variable $\mathcal{M}$ contains the actual channel tokens. Note that $\mathcal{M}$ is a polymorphic map where β is a type variable for the channel contents. The value $\mathcal{M}[c,i]$ denotes the i:th token produced on channel c.

Based on the global map definitions, it becomes possible to define Boogie encodings for the specification constructs introduced in Table 4.1. The definitions are listed in Figure 5.1. In the figure, the notion $\llbracket A \rrbracket$ is used to denote the Boogie encoding of the specification construct A. As an example, consider the encoding of the following precondition in contract C in Figure 4.1:

$$\llbracket \texttt{0} < \bullet\texttt{(x)} \implies \texttt{x[}\bullet - \texttt{1]} \leq \texttt{x[}\bullet\texttt{]} \rrbracket$$

which encodes into the following Boogie expression:

$$0 < \mathcal{I}[x] \implies \mathcal{M}[x, \mathcal{I}[x] - 1] \leq \mathcal{M}[x, \mathcal{I}[x]]$$

$$\begin{array}{lcl@{\qquad}lcl}
[\![c[i]]\!] & = & \mathcal{M}[c, [\![i]\!]] & [\![\mathtt{urd}(c)]\!] & = & \mathcal{C}[c] - \mathcal{R}[c] \\
[\![\bullet(c)]\!] & = & \mathcal{I}[c] & [\![\mathtt{next}(c)]\!] & = & \mathcal{R}[c] \\
[\![\mathtt{rd}(c)]\!] & = & \mathcal{R}[c] & [\![\mathtt{prev}(c)]\!] & = & \mathcal{R}[c] - 1 \\
[\![\mathtt{tot}(c)]\!] & = & \mathcal{C}[c] & [\![\mathtt{last}(c)]\!] & = & \mathcal{C}[c] - 1 \\
[\![\mathtt{rd}\bullet(c)]\!] & = & \mathcal{R}[c] - \mathcal{I}[c] & [\![\mathtt{tokens}(c, e)]\!] & = & \mathcal{C}[c] - \mathcal{R}[c] = [\![e]\!] \\
[\![\mathtt{tot}\bullet(c)]\!] & = & \mathcal{C}[c] - \mathcal{I}[c] & & &
\end{array}$$

Figure 5.1: Encoding of assertion constructs.

Note that standard arithmetic and logical operators have direct correspondences in Boogie. These are hence straight-forward to encode and are not included in Figure 5.1.

One of the main drawbacks with the loop-based encoding approach outlined above is the number of action invariants needed to verify programs of realistic size. In practice the approach requires that, for each network component D reading from a channel a and writing to a channel b, there is invariants expressing the relation between the tokens on a and b. In Chapter 6 it is discussed that, with the help of contracts, it is often possible to compute static schedules for parts of the dataflow networks not adhering to any well-known schedulable model of computation and that this can greatly reduce the number of invariants required.

5.3 Host language encoding

To verify that actor actions satisfy their contracts action bodies need to be encoded in the verifier. Boogie is itself a sequential programming language, which makes many of the constructs straight-forward to encode. However, the host language introduced in Section 3.2 has built-in support for MATLAB-style matrix operators and functions, which are not natively supported by Boogie or SMT solvers in general, and therefore need special treatment. Boogie's map datatype, used in the previous section to encode dataflow channels, can also be used to describe matrices. A matrix is encoded as a map of maps, where sub-maps correspond to matrix rows. However, operators and function on matrices also need to be encoded.

Within the scope of this thesis (Paper II), two different approaches to encoding verification conditions involving matrix operations in a verifier were evaluated. The first approach is based on giving matrix operations preconditions and postconditions in the same manner as in traditional program verification. The second approach uses information about matrix shapes to expand matrix operations.

As an example of the axiomatisation approach, consider the MATLAB addition operator (denoted with the sign +). When both operands are matrices, this operator returns the element-wise sum of the two operands. This behaviour can be described with the following axiom:

$$\forall\ \mathbf{int}\ i, \mathbf{int}\ j \cdot 1 \leq i \leq m \wedge 1 \leq j \leq n \Longrightarrow (a+b)(i,j) = a(i,j) +_s b(i,j)$$

where the operator $+_s$ denotes standard scalar addition. However, the behaviour of the MATLAB addition operator depends on the size of the operand matrices. If either of the operands is a scalar, the resulting matrix is the same size as the non-scalar operand, where each element is summed with the scalar operand. This can be described with the following axiom:

$$\forall\ \mathbf{int}\ i, \mathbf{int}\ j \cdot 1 \leq i \leq m \wedge 1 \leq j \leq n \Longrightarrow (a+b)(i,j) = a(i,j) +_s b$$

Hence, the MATLAB addition operator needs to be described by different axioms depending on the input matrix shapes. Consequently, this means that matrix size need to be known when generating verification conditions. The type system described in Section 3.3 is hence used to infer types, including matrix sizes, before generating verification conditions.

The other encoding approach evaluated within the scope of this thesis is *expansion*. The idea behind expansion is to utilize the fact that matrix shapes are known to expand matrix functions and operators. For a matrix a, the expansion $\llbracket a \rrbracket$ denotes the syntactically expanded matrix:

$$\llbracket a \rrbracket = \begin{bmatrix} \llbracket a(1,1) \rrbracket & \cdots & \llbracket a(1,n) \rrbracket \\ \vdots & \ddots & \vdots \\ \llbracket a(m,1) \rrbracket & \cdots & \llbracket a(m,n) \rrbracket \end{bmatrix}$$

Element-wise operators and functions like the addition operator for two matrices of size $\langle m, n \rangle$ described above is encoded as:

$$\llbracket a+b \rrbracket = \begin{bmatrix} \llbracket a(1,1) \rrbracket +_s \llbracket b(1,1) \rrbracket & \cdots & \llbracket a(1,n) \rrbracket +_s \llbracket b(1,n) \rrbracket \\ \vdots & \ddots & \vdots \\ \llbracket a(m,1) \rrbracket +_s \llbracket b(m,1) \rrbracket & \cdots & \llbracket a(m,n) \rrbracket +_s \llbracket b(m,n) \rrbracket \end{bmatrix}$$

Matrix element accesses are encoded as $\llbracket a \rrbracket(i,j)$. However, if a is an expression and i and j are known variables the element can directly extracted from a. This essentially means that, e.g., $\llbracket (a+b)(2,2) \rrbracket$ can be converted to $\llbracket a(2,2) \rrbracket +_s \llbracket b(2,2) \rrbracket$ directly.

Evaluation (Paper II) showed that the expansion approach yields efficient verification conditions as long as matrices are of relatively small size, which in practice is common in embedded control applications. The main drawback of the axiomatisation approach concerns so-called collapsing functions, e.g. for obtaining the sum over a vector, which typically needs recursive axioms. This is something not handled very efficiently by off-the-shelf SMT solvers. This problem does not arise with the expansion approach, as axioms for matrix functions and operators are not needed.

Chapter 6

Scheduling

One of the main drawbacks with dynamic dataflow programs, compared to restricted models of computation like static dataflow, is that scheduling of actions is a runtime problem. This incurs an execution overhead, decreasing the performance of the implemented program. Different approaches to alleviate this problem, by enabling as many scheduling decisions as possible to be made at compile-time, have been proposed [7, 14, 30]. In this chapter, it is discussed how contracts can aid in finding static schedules for dataflow programs which cannot be classified to belong to any well-established model of computation. Consequently, dataflow contracts are not only useful for specifying and verifying correct behaviour, but can also be utilised to improve runtime performance. Furthermore, it is also shown that the obtained schedules can be utilised in the verification process and that this significantly reduces the need for user-provided invariant annotations.

6.1 Contract-based scheduling

In the scheduling approach presented here, the target is to obtain a static schedule for each contract of the scheduled component. Contracts are used to specify the number of input tokens consumed and output tokens produced during one iteration of the schedule. A valid schedule is a sequence of firings, i.e. executions of actions, that returns the component to a state satisfying the contract invariants.

The contract-based scheduling approach is based on the observation that contracts naturally make explicit information that can be used to guide the search for repeating schedules. Consider again the example actor in Figure 4.3 and its contract. The contract describes the repeating behaviour of this actor, which is one firing of the action *first* followed by 15 firings of the action *rest*. Automatically finding these schedules is not a trivial problem and has been subject to extensive research. For instance, Ersfolk et al. [28, 14] used state-space analysis with the SPIN model checker to search for repeating schedules of dataflow networks. Their work does, however, not utilize extra user-provided information given in the form of contracts in this approach.

Based on the information provided in the contract, the schedule search for the example in Figure 4.3 can be narrowed to search for schedules that consume 16 integer input tokens. The search is performed by translating actors and networks to the input language of the SPIN model checker, named Promela, essentially reusing the work of Ersfolk et al. [28, 14]. Promela is designed to describe concurrent processes and hence includes built-in constructs such as communication channels, which are useful to describe actor networks. To utilize our contracts in the schedule search, the SMT solver Z3 is first used to generate an arbitrary instance of input satisfying the contract, which is then used in the generated Promela program. SPIN is then used to find a state where all input tokens have been consumed and the component has returned to a state satisfying the contract invariants. The actual schedule can then be obtained as an execution trace of a counter-example from the model checker.

It should be noted that this state space analysis only ensures that the obtained schedule is valid for the specific instance of input generated by the SMT solver. Further measures are needed to ensure that the schedule is valid for any input allowed by the contract.

6.2 Ensuring correctness

A schedule obtained through state space analysis can be utilised to ensure that a component is correct with respect to its contract. However, it also needs to be ensured that schedule is valid for all inputs allowed by the contract. Both correctness with respect to the contract as well as schedule validity can be verified by generating a Boogie program based on the schedule, including assertions ensuring that each step in the schedule is firable for any valid input. For the example in Figure 4.3, a Boogie program like the one outlined Figure 6.1 is obtained. Verifying the program ensures that:

1. The obtained schedule is valid for all inputs allowed by the contract. This means that the component is deadlock-free and that no preconditions of the subcomponents are violated.

2. The dataflow component is correct with respect to its contract. This follows from the fact that, for DPN, all valid schedules are functionally equivalent, as proved by Lee et al. [51, 54].

```
$\textbf{assume}\ \llbracket C_{cinv} \rrbracket$;
$\mathcal{C}[in] := \mathcal{C}[in] + 16$;
$\textbf{assume}\ \llbracket C_{grd} \rrbracket \wedge \llbracket C_{pre} \rrbracket$
$\textbf{assert}\ 1 \leq \mathcal{C}[in] - \mathcal{R}[in] \wedge \llbracket \texttt{count = 0} \rrbracket$;
fire(first);
$\textbf{assert}\ 1 \leq \mathcal{C}[in] - \mathcal{R}[in] \wedge \llbracket \texttt{count} \neq \texttt{0} \rrbracket$;
fire(rest);
$\textbf{assert}\ 1 \leq \mathcal{C}[in] - \mathcal{R}[in] \wedge \llbracket \texttt{count} \neq \texttt{0} \rrbracket$;
fire(rest);
...
$\textbf{assert}\ \mathcal{C}[out1] - \mathcal{R}[out1] = 1 \wedge \mathcal{C}[out2] - \mathcal{R}[out2] = 15$
$\textbf{assert}\ \llbracket C_{post} \rrbracket \wedge \llbracket C_{cinv} \rrbracket$;
```

Figure 6.1: High-level overview of the Boogie program obtained by scheduling the actor in Figure 4.3 according to its contract.

In Figure 6.1, fire is used to denote firing of an action or sub-component. In practice fire consists e.g. of updating the global state variables $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{M}$, asserting sub-componenent preconditions, and assuming sub-component postconditions.

One of the main advantages with utilising contracts and schedules in the correctness proof, compared to the approach in Chapter 5, is that the number of invariants the user needs to provide is greatly reduced. Scheduling essentially unrolls the loop in (5.1), which means that most often no invariants are needed to express the relationship between components in the network. In practice, only contract invariants, expressing the state between contract windows, are then needed. On the other hand, the main drawback with the schedule-based verification approach is that schedules with a very large number of firings also converts into large Boogie programs, which consequently may be challenging or slow to verify. In Table 6.1 a comparison of number of invariants and elapsed verification time for a number of example programs with and without scheduling is presented (Paper IV). It can be noted that scheduling significantly reduces the number of needed invariants, but the verification time also increased in most cases. This is due to the larger Boogie programs generated.

Table 6.1: The number of user-provided invariants needed as well as verification time for a number of different dataflow networks with and without contract-based scheduling (Paper IV)

	Unscheduled		**Scheduled**	
Network	**Invariants**	**Time (s)**	**Invariants**	**Time (s)**
SumNet	3	1.7	3	5.1
DataDependent	6	1.9	0	7.4
IIR	2	1.9	2	5.3
FIR	4	2.4	4	5.8
LMS	9	14.6	9	12.1
ZigBee	28	6.8	2	250.7
Addressing	11	3.4	1	12.3
DCRInvpred	25	38.2	1	18.2
DCSplit	6	11.8	1	8.6
Algo_IS	8	3.4	2	72.0
Algo_IAP	5	2.7	1	22.3
Dequant	3	2.2	1	11.1
Algo_IDCT2D	2	3.6	0	13.4
DCReconstruction	13	4.6	0	40.6
Texture	29	5.9	0	103.9

6.3 Improving runtime performance using contracts

Schedules obtained by scheduling can also be used to generate executable code for the network. As the schedule defines in which order actions will be fired, there is no need to use processor time to evaluate action guards for the scheduled actions at runtime. For dataflow networks of realistic size, this kind of optimisations can have significant impact on execution time [7, 14, 30]. Hence, contracts are not only useful to state and verify functional properties, but they can also improve runtime performance.

Going further, contracts can, in fact, enable compile-time scheduling decisions that would be extremely challenging or even impossible to deduce solely based on the network and actor implementations. As an example, consider the actor *DupNonNeg* and the network *N* in Figure 6.2. The actor *DupNonNeg* writes non-negative outputs twice to the output channel and negative input once. Assume that it is known that the input to *N* is always a negative number followed by a non-negative number. This can be expressed as a contract precondition, as done for the contract *C* of *N*. Based on *C*, the proposed contract-based scheduling

```
actor DupNonNeg int in1, int in2 ⟹ int out:
  a1: action in:[ x ] ⟹ out:[ x, x ] guard 0 ≤ x end
  a2: action in:[ x ] ⟹ out:[ x ] guard x < 0 end
end

network N int in ⟹ int out:
  C: contract in:2 ⟹ out:3
    requires in[•] < 0 ∧ 0 ≤ in[•+1]
  end


  entities
    a = DupNonNeg();
  end

  structure
    a: in ⟶ a.in;
    b: a.out ⟶ out;
  end
end
```

Figure 6.2: Properties provided in contracts can be used to do scheduling decisions.

approach can deduce a static schedule for N consisting of one firing of *a2* followed by one firing of *a1*. Without the extra information provided in the contract C, it would not be possible to obtain this static schedule for the network N.

Furthermore, state-space analysis can be used not only to find an arbitrary valid schedule, but a schedule optimal with respect to a provided cost function. This is possible by instructing the SPIN model checker to search for schedules in a Branch-and-Bound fashion [70]. Hence, it is possible to, for instance, find a schedule optimised for minimal channel buffer sizes.

Within this thesis (Paper IV), the runtime performance improvements for an MPEG-4 decoder obtained by scheduling according to two different cost functions were evaluated. The programs were evaluated on two different platforms; a laptop with an Intel i5 processor and an Odroid platform with an ARMv7 processor. Overall, improvements in throughput of 70 to 95 %, compared to making all scheduling decisions at runtime, were observed. It was also observed that, depending on the platform, it varied which scheduling cost function produced the most efficient code.

```
actor Sign int in ⟹ int out:
  contract in:1 ⟹ out:1 end
  pos:  action in:[ x ] ⟹ out:[ 1 ] guard x > 0 end
  zero: action in:[ x ] ⟹ out:[ 0 ] guard x = 0 end
  neg:  action in:[ x ] ⟹ out:[ −1 ] guard x < 0 end
end
```

Figure 6.3: Although an actor satisifes its contract, it is not necessarily possible to generate a static schedule based on the contract.

6.4 Limitations

The scheduling approach described above has some limitations. A contract guard and precondition is not necessarily strict enough to yield a schedule valid for any allowed input. Consider, for instance, the simple actor in Figure 6.3 implementing a sign function. It is apparent that the actor satisfies the contract, which only states that the actor consumes 1 token and outputs 1 token. However, it is not possible to obtain a static schedule valid for all inputs based on this contract, as any of the three actions can fire on inputs allowed by the contract. Using the proposed contract-based scheduling approach, state-space analysis would return a schedule for one of the actions, depending on the input instance generated by the SMT solver, but the Boogie correctness check would fail, since the obtained schedule is not correct for all allowed inputs. In cases like this, a separate contract has to be supplied for all three cases. However, the presented verification approach does not require that the entire network is statically schedulable. This means that the approach with unknown schedule described in Chapter 5 can be applied to select components of the network, while the schedule-based approach is used to verify other components.

Another limitation regarding the schedule-based approach concerns networks with feedback loops. The hierarchical scheduling performed based on contracts is not always possible for such networks. However, also in these cases the approach in Chapter 5 can be used as a fall back.

Chapter 7

Related work

The focus of this thesis is on contract-based specification and verification of dataflow programs. The thesis is based on, and relates to, research from several different research topics. In this chapter, the thesis is reviewed in relation to other work. The chapter is split into different sections roughly based on research topic.

7.1 Verification

The use of contracts has a long history. The foundations go back to the works of Dijkstra [24], Hoare [41] and Jones [47], while contract-based design as a term was introduced by Meyer [65, 43]. Contract-based static verification has been researched and verifiers implemented for a large amount of widely used general-purpose programming languages, for instance, Java [15, 18], C# [6] and C [49] and .NET [29].

Dafny [55] is a rich programming language that is designed specifically with contracts and verification of functional correctness in mind. The language Chalice [56, 57] and the verifier with the same name support verification of multi-threaded programs. Chalice also supports communication over channels that can be verified to be deadlock free [58].

Several verification backends that aim to ease the transformation from programs with specifications (e.g. contracts) to input for an SMT solver exist, for instance Boogie [5], Why3 [31] and Viper [67]. In this thesis work, Boogie [5] has been used as a backend because it is a mature tool supporting an expressive sequential programming language as input. This makes it convenient to encode the considered host language. Boogie also has tight integration with the SMT solver Z3 [21], which means that it typically generates very efficient verification conditions. Also verifiers for other languages mentioned above, e.g. Dafny and Chalice, are based on translating the programs and contracts to a Boogie representation.

Verification of dataflow programs has also been studied extensively before, especially in the synchronous setting. In [62] Maraninchi and

Morel presented contracts for Lustre. Based on the connection between Simulink and Lustre [75] this could be applied also on Simulink models. Hagen and Tinelli have used k-induction with Lustre [36, 35]. A more recent contract language for synchronous reactive systems is CoCoSpec [19]. CoCoSpec is mode-aware, which, in essence, is similar to annotating components with multiple contracts in the approach of this thesis. Dragomir, Preoteasa and Tripakis have developed a compositional semantics and analysis framework for reactive systems based on predicate transformers [25]. They have also developed a type inference technique for Simulink diagrams in Isabelle [69]. In recent work [73] by Sun et al. a contract-based semantics and refinement for Simulink models supporting both continuous and discrete-time Simulink was developed. They do, however, not support multi-rate models and do not discuss automatic verification. Limiting the scope to discrete-time Simulink enables verifying Simulink models through translation to SDF, as done in this thesis. However, none of the approaches mentioned above support asynchronous dataflow in the same framework.

Static analysis of dynamic dataflow networks has also been studied to some extent before. In [44] an approach to modular analysis of Dataflow Process Networks based on Interface Automata is presented. In this work, dataflow processes are associated interface automata describing the interface and environmental assumptions. An extension to this work, Counting Interface Automata [76], has also been researched. Counting Interface Automata can capture temporal and quantitative aspects of actor interfaces. However, neither of these approaches consider contract-based verification.

The work in this thesis extends on a line of research by Boström et al. [9, 10, 11, 12] on contract-based design of Simulink models. The work in this thesis extends on the above works by generalizing it to consider, for instance, multi-rate Simulink models, matrix data types, as well as asynchronous and dynamic dataflow models of computation. The approach is also evaluated on example programs to a much greater extent in this thesis.

The actor model introduced by Hewitt, Bishop and Steiger [40] should also be mentioned in this context. On a high level, actors are concurrent objects that communicate with each other by message passing. Under this broad definition also the dataflow actor networks considered in this thesis could be considered special cases of the actor model. Implementations of the actor model are found in many modern programming languages and software libraries, such as Erlang [4], Scala [37], and Ray [66]. Several frameworks for analysis of actor programs exists. Rebeca [71], for instance, provides a formalism and model checking technique for actor networks. Recent work by Lohstroh et al. [60] introduces an interesting model called reactors, which combines concepts from the actor model with concepts from dataflow and synchronous models of computation to achieve a deterministic and timed model of computation. Closely related to the actor model is also the concept of active (or asynchronous) objects [20]. There exists a large amount of research [2, 45, 50] on specification and verification of such programs. In contrast to the dataflow networks considered in this thesis, the actor model is inherently non-deterministic, which makes it more challenging to verify correctness properties. Dataflow actor networks are also static, while actors in the above models can be dynamically created. Restricting the approach to deterministic and static networks significantly simplifies reasoning and enables proving stronger properties fully automatically.

7.2 Type systems

In this thesis, a matrix shape aware type system is proposed to allow compile-time type checking and type inference of a subset of the Embedded MATLAB language. Similar type systems for MATLAB have been studied before, but for the purpose of program optimisation [3, 22, 46]. The main target of these works is to achieve performance gains by using inferred matrix shapes to pre-allocate memory and avoiding runtime bounds checks. In these works, failed static type inference means fall back to runtime inference, while the verification approach proposed in this thesis requires that all matrix shapes are known at compile-time. Also MATLAB performs static type and shape inference e.g. when generating code for embedded platforms. It seems that forward-propagation of type information is applied there.

There are some programming languages, e.g. FiSH [42], which have a type system that has been designed with the explicit goal to be aware of data shapes. FiSH allows the user to provide type annotations containing matrix and vector shapes. A dependent type system for ML [87, 88] has also been investigated, which uses constraint solving for type inference. The index language used for array shapes can be arbitrarily complex and is ultimately restricted only by the abilities of the chosen constraint solver. While this approach is applied to the functional language ML, it is potentially more general than the one proposed in this thesis.

7.3 Scheduling

The verification approaches discussed in this thesis are to a large extent based on mapping, i.e. scheduling, the dataflow networks to sequential programs. Lee et al. [51, 54] established that, given some restrictions such as mutual exclusiveness of firing rules, sequential schedules for dataflow process networks are functionally equivalent. This is a central result, which the verification approaches presented in this thesis build upon.

For static dataflow (SDF), there exists well-established scheduling algorithms [52, 53]. Using model checking to find schedules exhibiting certain properties has also been done before [34, 59]. The approach proposed in this thesis enables this for a broader class of dataflow programs than SDF.

Boutellier et al. [14] presented a method for merging actors into composite actors. To do this, they use model checking to find static schedules for parts of a network. The actions of the obtained composite actors are similar to those obtained based on contracts in this work. Essentially, they try to automatically deduce similar properties that are made explicit in contracts in this work. The translation from dataflow networks to SPIN used in this thesis work is based on their work.

A classification method for dataflow actors using satisfiability and abstract interpretation has been developed by Wipliez and Raulet [86]. They also use the Z3 SMT solver as a backend. Using the SMT solver, they detect, e.g., a class of actors that they call *time-dependent*, which can react to absence of tokens on channels. Their classification can, e.g., ensure actor determinism, which is also achieved by generating verification conditions checking for mutual exclusiveness action guards in this work.

There are also approaches that more explicitly describe dynamic behaviour. Siyoum et al. [72] presented an approach based on scenarios, where dynamic dataflow networks are implemented according to a Disciplined Dataflow Network MoC. The dynamic behaviour is captured by special actors, which have a purpose similar to that of contracts in this work. However, these actors are part of the actual programs and not implementation independent like the contracts proposed in this thesis work.

Composition of dataflow actors does not preserve rate consistency and deadlock freedom. Tripakis et al. [74] proposed DSSF (Deterministic SDF with Shared FIFOs), a profile-based methodology to handle actor composition in hierarchical dataflow networks. Falk et al. [30] proposed a rule-based quasi-static scheduling approach with similar targets. In their work, static actors are composed to form composite actors for which the quasi-static schedules guarantee global deadlock freedom. In this thesis work, deadlock freedom is guaranteed by verification with respect to contracts.

The dependencies between actors in a dataflow program can also be analysed from execution traces of the program. Canale et al. [16] presented an approach which translates execution trace graphs of a dataflow program to Petri Nets which are used for e.g. buffer dimensioning. A similar approach could potentially be used as an alternative to optimizing the schedules by model checking as done in this thesis work.

Scheduling and code generation methods specifically aimed at synchronous, rather than asynchronous, languages have also been researched to a great extent. Modular code generation methods for synchronous languages similar to Lustre have been proposed [8, 61]. In [61], code generation for a fairly general block diagram notation is presented. The blocks can be triggered by different means such as clocks or other signals. Diagrams where the time is given by a period-phase pair as in Simulink are discussed in detail. The code generation methods in [8, 61] could be considered as an alternative to scheduling Simulink models through translation to SDF in this thesis work. However, using SDF as an intermediate approach enables a unified framework which handles also more general asynchronous MoCs.

Chapter 8

Conclusions

This thesis has presented a contract-based approach to specification, verification, as well as scheduling of dataflow programs. The thesis is based on four peer-reviewed research papers, which together form a unified specification and verification framework for synchronous as well as asynchronous dataflow programs. In this chapter, the main contributions of the thesis are summarised, followed by a discussion on possible directions for future work.

8.1 Summary

In the introduction of this thesis, in Section 1.2, two research questions were stated. One of the questions concerned finding specification constructs suitable for dataflow programs. The thesis proposed two contract-based specification formats, for Simulink and asynchronous dataflow programs, respectively. Both contract formats consist of precondition and postcondition assertions expressed in languages that are similar to the expressions of the host languages used to implement the actors. This allows engineers to write contract conditions in a language already familiar to them. Evaluation showed that the contract formats also are expressive enough to describe interesting properties of dataflow programs. The relationship between the proposed contract formats were reviewed and it was outlined how Simulink contracts can be mapped to contracts for asynchronous dataflow.

The second research question concerned how to efficiently encode specification constructs and dataflow programs for automatic verification. The thesis proposed verification approaches that are based on translating dataflow networks to sequential programs, which enable verification using well-established techniques and tools developed for sequential program verification. The Boogie program verifier was found to be a suitable verification backend. It is an efficient verification tool, which input language is an expressive sequential programming language. The polymorphic map support of Boogie offered a convenient and efficient way to encode both matrix data types as well as dataflow communication buffers. Using a verifier for sequential programs also provides a convenient way to handle verification of actors implemented in sequantial code within the same framework.

In the proposed verification approach, Simulink models are handled by translation to an intermediate representation in the asynchronous static dataflow (SDF) model of computation. This enables handling both Simulink as well as more general, asynchronous, models of computation in the same framework.

A type system and method for compile-time inference of matrix types and shapes for a subset of Embedded MATLAB was also proposed as part of this thesis. This allows verifying dataflow networks and actors which have MATLAB-style native support for matrix data types. Two approaches to encoding matrix operators and functions were investigated. An approach, where the inferred matrix shapes are utilised to expand the matrix functions and operators, was found to be efficient for typical embedded applications, where matrix sizes are relatively small.

Furthermore, it was also shown that providing dataflow networks with the proposed contracts does not only enable automatic verification of the stated properties, but can also improve runtime performance. This is based on the observation that the proposed contracts often make explicit properties implicitly assumed by the developer. By utilising this information, a larger amount of scheduling decision can be done at compile-time, hence avoiding costly scheduling decisions at runtime. Additionally, it was also found that obtained schedules can be utilized in the verification, with the advantage that, in many cases, significantly fewer invariants are needed to enable automatic verification.

8.2 Future work

There are several potential directions for future research. Feedback loops in dataflow networks should be better supported in the contract-based verification and scheduling approach. This could potentially be solved by introducing state at the level of contracts in conjunction with finer-grained scheduling.

It would also be beneficial to extend the number of MATLAB matrix constructs supported. For instance, Embedded MATLAB supports choosing any sub-matrix from a given matrix, which is currently not supported in this framework. In addition, abstract interpretation [38, 33] could possibly be applied to infer properties regarding recursively defined functions, which could enable more efficient verification conditions.

Another interesting future direction would be to investigate networks where actors could be dynamically created, e.g. as response to input values. This would bring the model of computation closer to the more general actor model [40, 39, 1] and would, for instance, introduce challenges regarding non-determinism not present in strictly static networks.

During recent years, several dataflow-based frameworks aimed at implementing data and machine learning pipelines have emerged. Widely used frameworks are, for instance, Kubeflow Pipelines[1], Flyte[2] and Prefect[3]. It would be interesting to investigate the relationship between these frameworks and the dataflow models of computation considered in this thesis. Potentially, the contract-based framework proposed in this thesis could be used as a basis to develop a contract-based framework for data and machine learning pipelines.

[1]https://www.kubeflow.org/docs/components/pipelines/

[2]https://flyte.org/

[3]https://www.prefect.io/

Bibliography

[1] G. A. Agha. *Actors: A model of concurrent computation in distributed systems.* PhD thesis, University of Michigan, USA, 1985.

[2] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Sci. Comput. Program.*, 77(12):1289–1309, 2012.

[3] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. *SIGPLAN Not.*, 37(5):294–303, 2002.

[4] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG.* Prentice Hall, 2 edition, 1996.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer Berlin Heidelberg, 2006.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[7] B. Bhattacharya and S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for dsp systems. In *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668)*, pages 84–89, 2000.

[8] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2008*, page 121–130. ACM, 2008.

[9] P. Boström. Contract-based verification of Simulink models. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering, ICFEM 2011*, volume 6991 of *LNCS*, pages 291–306. Springer Berlin Heidelberg, 2011.

[10] P. Boström, M. Linjama, L. Morel, L. Siivonen, and M. Waldén. Design and validation of digital controllers for hydraulics systems. In *The 10th Scandinavian International Conference on Fluid Power*, 2007.

[11] P. Boström, L. Morel, and M. Waldén. Stepwise development of simulink models using the refinement calculus framework. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Theoretical Aspects of Computing, ICTAC 2007*, volume 4711 of *LNCS*, pages 79–93. Springer Berlin Heidelberg, 2007.

[12] P. Boström. *Formal Design and Verification of Systems Using Domain-Specific Languages.* PhD thesis, Turku Centre for Computer Science, 2008.

[13] P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, 15(4):1141–1161, 2016.

[14] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Trans. Signal Process.*, 63(10):2496–2508, 2015.

[15] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[16] M. Canale, S. C. Brunet, E. Bezati, M. Mattavelli, and J. W. Janneck. Dataflow programs analysis and optimization using model predictive control techniques - two examples of bounded buffer scheduling: Deadlock avoidance and deadlock recovery strategies. *J. Signal Process. Syst.*, 84(3):371–381, 2016.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188. ACM, 1987.

[18] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, FMCO 2005*, volume 4111 of *LNCS*, pages 342–363. Springer Berlin Heidelberg, 2006.

[19] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. CoCoSpec: A mode-aware contract language for reactive systems. In R. De Nicola and E. Kühn, editors, *Software Engineering and Formal Methods, SEFM 2016*, volume 9763 of *LNCS*, pages 347–366. Springer International Publishing, 2016.

[20] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.

[21] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer Berlin Heidelberg, 2008.

[22] L. De Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[23] J. B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, pages 362–376. Springer Berlin Heidelberg, 1974.

[24] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, 1976.

[25] I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional semantics and analysis of hierarchical block diagrams. In D. Bošnački and A. Wijs, editors, *Model Checking Software, SPIN 2016*, volume 9641 of *LNCS*, pages 38–56. Springer International Publishing, 2016.

[26] J. Eker. and J. W. Janneck. CAL language report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, 2003.

[27] J. Ersfolk, P. Boström, V. Timonen, J. Westerholm, J. Wiik, O. Karhu, M. Linjama, and M. Waldén. Optimal digital valve control using embedded GPU. In J. Uusi-Heikkilä and M. Linjama, editors, *Proceedings of the 8th Workshop on Digital Fluid Power.* Tampere University of Technology, 2016.

[28] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli. Scheduling of dynamic dataflow programs based on state space analysis. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1661–1664, 2012.

[29] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software, FoVeOOS 2010*, volume 6528 of *LNCS*, pages 10–30. Springer Berlin Heidelberg, 2011.

[30] J. Falk, C. Zebelein, C. Haubelt, and J. Teich. A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):74:1–74:31, 2013.

[31] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems, ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer Berlin Heidelberg, 2013.

[32] R. W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[33] P.-L. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods, NFM 2013*, volume 7871 of *LNCS*, pages 139–154. Springer Berlin Heidelberg, 2013.

[34] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings. 42nd Design Automation Conference, 2005*, pages 819–824. ACM, 2005.

[35] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–9, 2008.

[36] G. E. Hagen. *Verifying Safety Properties of Lustre Programs: An SMT-based Approach.* PhD thesis, University of Iowa, 2008.

[37] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[38] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov. Invariant and type inference for matrices. In G. Barthe and M. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2010*, volume 5944 of *LNCS*, pages 163–179. Springer Berlin Heidelberg, 2010.

[39] C. Hewitt. Viewing control structures as patterns of passing messages. *Artifical Intelligence*, 8(3):323–364, 1977.

[40] C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. William Kaufmann, 1973.

[41] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[42] C. B. Jay and P. A. Steckler. The functional imperative: Shape! In C. Hankin, editor, *Programming Languages and Systems, ESOP 1998*, volume 1381 of *LNCS*, pages 139–153. Springer Berlin Heidelberg, 1998.

[43] J.-M. Jazequel and B. Meyer. Design by contract: The lessons of Ariane. *IEEE Comput.*, 30(1):129–130, 1997.

[44] Y. Jin, R. Esser, C. Lakos, and J. W. Janneck. Modular analysis of dataflow process networks. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering, FASE 2003*, volume 2621 of *LNCS*, pages 184–199. Springer Berlin Heidelberg, 2003.

[45] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects, FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer Berlin Heidelberg, 2012.

[46] P. G. Joisha and P. Banerjee. An algebraic array shape inference system for MATLAB. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, 2006.

[47] C. B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University, 1981.

[48] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974*, pages 471–475. North-Holland, 1974.

[49] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Form. Asp. Comp.*, 27(3):573–609, 2015.

[50] I. W. Kurnia and A. Poetzsch-Heffter. Verification of open concurrent object systems. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects, FMCO 2012*, volume 7866 of *LNCS*, pages 83–118. Springer Berlin Heidelberg, 2013.

[51] E. A. Lee. A denotational semantics for dataflow with firing. Technical Report Technical Memorandum UCB/ERL M97/3, Electronics Research Laboratory, Berkeley, 1997.

[52] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, C-36(1):24–35, 1987.

[53] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[54] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5):773–801, 1995.

[55] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2010*, volume 6355 of *LNCS*, pages 348–370. Springer Berlin Heidelberg, 2010.

[56] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *Programming Languages and Systems, ESOP 2009*, volume 5502 of *LNCS*, pages 378–393. Springer Berlin Heidelberg, 2009.

[57] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri,

editors, *Foundations of Security Analysis and Design V, FOSAD 2007-2009*, volume 5702 of *LNCS*, pages 195–222. Springer Berlin Heidelberg, 2009.

[58] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *Programming Languages and Systems, ESOP 2010*, volume 6012 of *LNCS*, pages 407–426. Springer Berlin Heidelberg, 2010.

[59] W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009*, pages 61–70. ACM, 2009.

[60] M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillón, E. A. Lee, and A. L. Sangiovanni-Vincentelli. Reactors: A deterministic model for composable reactive systems. In R. D. Chamberlain, M. E. Grimheden, and W. Taha, editors, *Cyber Physical Systems, Model-Based Design, CyPhy 2019, and WESE 2019*, volume 11971 of *LNCS*, pages 59–85. Springer, 2019.

[61] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 78–89. ACM, 2009.

[62] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th Euromicro Conference, 2004*, pages 48–55, Aug 2004.

[63] M. Mattavelli, I. Amer, and M. Raulet. The reconfigurable video coding standard. *IEEE Signal Process. Mag.*, 27(3):159–167, 2010.

[64] B. Meyer. Design by contract. Technical Report Technical Report TR-EI-12/CO, University of California at Berkeley, 1986.

[65] B. Meyer. Applying 'design by contract'. *IEEE Comput.*, 25(10):40–51, 1992.

[66] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In A. C. Arpaci-Dusseau and G. Voelker, editors, *13th USENIX Conference on Operating Systems Design and Implementation, OSDI 2018*, pages 561–577. USENIX Association, 2018.

[67] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2016*, volume 9583 of *LNCS*, pages 41–62. Springer Berlin Heidelberg, 2016.

[68] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Hamburg, 1962.

[69] V. Preoteasa, I. Dragomir, and S. Tripakis. Type inference of Simulink hierarchical block diagrams in Isabelle. In A. Bouajjani and A. Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems, FORTE 2017*, volume 10321 of *LNCS*, pages 194–209. Springer International Publishing, 2017.

[70] T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In *Model Checking Software, SPIN 2003*, volume 2648 of *LNCS*, pages 1–17. Springer-Verlag, 2003.

[71] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundam. Informaticae*, 63(4):385–410, 2004.

[72] F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal. Automated extraction of scenario sequences from disciplined dataflow networks. In *2013 ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2013*, pages 47–56, 2013.

[73] Q. Sun, W. Zhang, C. Wang, and Z. Liu. A contract-based semantics and refinement for Simulink. In W. Dong and J.-P. Talpin, editors, *Dependable Software Engineering. Theories, Tools, and Applications, SETTA 2022*, volume 13649 of *LNCS*, pages 134–148. Springer Nature Switzerland, 2022.

[74] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):83:1–83:26, 2013.

[75] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, 2005.

[76] E. Wandeler, J. W. Janneck, E. A. Lee, and L. Thiele. Counting interface automata and their application in static analysis of actor models. In *Third IEEE International Conference on Software Engineering and Formal Methods, SEFM 2005*, pages 106–115. IEEE, 2005.

[77] P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-dynamic dataflow. In *4th Euromicro Workshop on Parallel and Distributed Processing*, pages 319–326, 1996.

[78] J. Wiik. Contract-based verification of multi-rate Simulink models. Master's thesis, Åbo Akademi University, 2012.

[79] J. Wiik and P. Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. In S. Merz and J. Pang, editors, *Formal Methods and Software Engineering, ICFEM 2014*, volume 8829 of *LNCS*, pages 396–412. Springer International Publishing, 2014.

[80] J. Wiik and P. Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. Technical Report 1107, TUCS, 2014.

[81] J. Wiik and P. Boström. Contract-based verification of MATLAB-style matrix programs. *Formal Aspects of Computing*, 28(1):79–107, 2016.

[82] J. Wiik and P. Boström. Specification and automated verification of dynamic dataflow networks. Technical Report 1170, TUCS, 2016.

[83] J. Wiik and P. Boström. Specification and automated verification of dynamic dataflow networks. In A. Cimatti and M. Sirjani, editors, *Software Engineering and Formal Methods, SEFM 2017*, volume 10469 of *LNCS*, pages 136–151. Springer International Publishing, 2017.

[84] J. Wiik and P. Boström. Contract-based specification and verification of dataflow programs. In L. Aceto and A. Ingolfsdottir, editors, *Proceedings of 27th Nordic Workshop on Programming Theory, NWPT 2015.* Reykjavik University, 2015.

[85] J. Wiik, J. Ersfolk, and M. Waldén. A contract-based approach to scheduling and verification of dynamic dataflow networks. In P. Derler and S. Gao, editors, *16th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2018*, pages 1–10. IEEE, 2018.

[86] M. Wipliez and M. Raulet. Classification of dataflow actors with satisfiability and abstract interpretation. *International Journal of Embedded and Real-Time Communication Systems*, 3(1):49–69, 2012.

[87] H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[88] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999*, pages 214–227. ACM, 1999.

ISBN 978-952-12-4340-0