Fachgebiet Programmierung eingebetteter Systeme
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

A Framework for the Automatic Verification of Discrete-Time MATLAB Simulink Models using Boogie

vorgelegt von

Robert Reicherdt,

Master of Science in Computer Science

geb. in Erlabrunn

von der Fakultät IV – Elektrotechnik und Informatik

der Technischen Universität Berlin

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 10. Juli 2015

Berlin, 2015

Abstract

MATLAB/Simulink is a widely used industrial tool for the development of embedded systems, especially for the development of embedded controller software in automotive industries. Since such embedded systems are often deployed in safety critical areas where an error may lead to severe injuries and even to death of persons, comprehensive and complete quality assurance measures are required for ensuring their correctness in all possible cases. Still, incomplete techniques like testing are favored over safe formal techniques in practice. Although there exist some formal verification approaches for MATLAB/Simulink models that can guarantee correctness, they are either poorly automated or suffer from scalability issues.

To overcome this problem, we present an approach for a highly automated verification framework for MATLAB/Simulink models that enables the formal verification of discrete-time controller models. Our main idea is to use inductive verification techniques in combination with an automatic extraction of verification goals for a number of important run-time error classes to provide an automatic verification flow. Furthermore, as automatic model reduction technique, we present a slicing approach for MATLAB/Simulink. With that, we increase the scalability by dividing a possibly complex verification task into a number of less complex subtasks.

To enable the automatic verification of MATLAB/Simulink models, we present a formal semantics for discrete-time models based on a mapping of the informally defined sequential simulation semantic into the formally well defined intermediate verification language Boogie2. Together with automatically generated invariants and verification goals that are automatically weaved into the formal model, this mapping enables the verification of the models using the Boogie verification framework and inductive verification techniques. To achieve a high degree of automation, we also support inductive verification over more than one simulation step (k-induction), which allows for weaker invariants that can be generated automatically at the price of decreased scalability. To overcome scalability issues for k-induction, we use our novel slicing technique for MATLAB/Simulink models to automatically reduce a model to those blocks that are relevant for a (possible) error at a particular block. Furthermore, we propose a process for the efficient use of our verification and slicing techniques.

To show the practical applicability of our framework, we have implemented our approach as the *MeMo* tool suite and applied our verification process to two industrial case studies. With that, we demonstrate the performance and the capability to automatically verify a given model for the absence of important run-time errors with our verification framework for discrete-time MATLAB/Simulink models.

Zusammenfassung

MATLAB/Simulink ist ein weit verbreitetes Werkzeug für die Entwicklung von eingebetteten Systemen, welches vor allem in der Automobilindustrie zur Entwicklung von Steuerungssystemen benutzt wird. Da solche Systeme häufig in sicherheitskritischen Umgebungen eingesetzt werden, wo eine Fehlfunktion zu schweren Verletzungen und Todesfällen führen kann, werden umfangreiche und vollständige Qualitätssicherungsmaßnahmen benötigt um die Korrektheit der Systeme für alle möglichen Ausführungen sicher zu stellen. Trotzdem werden unvollständige Techniken wie Testen in der Praxis gegenüber den sicheren formalen Methoden bevorzugt. Obwohl es einige formale Verifikationstechniken für MATLAB/Simulink gibt, sind diese entweder kaum automatisiert oder skalieren schlecht.

Um dieses Problem zu lösen stellen wir in dieser Arbeit einen Ansatz für eine hochautomatisierte Verifikationsumgebung für MATLAB/Simulink Modelle vor, mit der zeitdiskrete Modelle für Steuerungen formal verifiziert werden können. Die Kernidee in unserem Ansatz ist hierbei eine Kombination aus induktiven Verifikationstechniken und dem automatischen Extrahieren von Verifikationszielen für bestimmte Laufzeitfehler verwenden, um einen automatisierten Verifikationsfluss zu erreichen. Außerdem stellen wir einen Ansatz für das Slicing von MATLAB/Simulink Modellen vor, der als automatisches Verfahren zur Reduzierung der Modellkomplexität verwendet wird. Mit diesem erreichen wir eine bessere Skalierbarkeit, da wir eine komplexe Verifikationsaufgabe in eine Anzahl von weniger komplexen Teilaufgaben aufspalten können.

Um die automatische Verifikation von MATLAB/Simulink Modellen zu ermöglichen, stellen wir eine formale Semantik für zeitdiskrete Modelle vor, die auf einer Abbildung der informellen, sequenziellen Simulationssemantik in die formale Verifikationszwischensprache Boogie2 basiert. Zusammen mit automatisch erzeugten und in das formale Modell eingewobenen Invarianten und Verifikationszielen, erlaubt dies die Verifikation der Modelle mit dem Boogie-Framework und induktiven Verifikationstechniken. Um einen hohen Grad der Automatisierung zu erreichen, unterstützen wir auch induktive Verifikationstechniken über mehr als einen Simulationsschritt (k-Induktion), was zwar die Verwendung von schwächeren, automatisch generierten Invarianten erlaubt, aber gleichzeitig die Skalierbarkeit reduziert. Um dem entgegenzuwirken, nutzen wir unser neuartiges Slicing-Verfahren für MATLAB/Simulink Modelle um diese automatisch auf genau die Blöcke zu reduzieren, die für einen (möglichen) Fehler an einem bestimmten Block relevant sind. Darüber hinaus schlagen wir einen Prozess für die effiziente Benutzung unserer Verifikations- und Slicing-Techniken vor.

Um die praktische Anwendbarkeit unserer Verifikationsumgebung zu zeigen, haben wir diese in unserem *MeMo*-Werkzeug implementiert und unseren Prozess auf zwei industrielle Fallbeispiele angewendet. Damit haben wir die Performanz und die Fähigkeit, die Abwesenheit von wichtigen Laufzeitfehlern in einem gegebenen Modell automatisch zu verifizieren, nachgewiesen.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet ”Programmierung eingebetteter Systeme”. Daher möchte ich mich an dieser Stelle herzlich für die Betreuung meiner Arbeit durch Prof. Dr. Sabine Glesner bedanken. Sie hat mir die Möglichkeit gegeben in einem praxisnahen Bereich zu promovieren und mir den Freiraum gelassen, eigene Schwerpunkte zu setzen und hat durch ihre Bereitschaft zur Diskussion und ihre wertvollen Anmerkungen zum Gelingen dieser Arbeit beigetragen. Weiterhin möchte ich Herrn Prof. Dr. Giese und Herrn Prof. Dr. Kao für die Durchsicht und Begutachtung meiner Arbeit danken. Nicht zuletzt gilt mein Dank unseren Industriepartnern, der Berner und Mattner Systemtechnik GmbH und der Model Engineering Solutions GmbH, die mir durch Bereitstellung der Anwendungsfälle und Fallstudien geholfen haben, Kriterien und Anknüpfungspunkte für meine Arbeit abzuleiten.

Natürlich möchte ich mich auch bei den Kolleginnen und Kollegen in unserem Fachgebiet für die angenehme Atmosphäre und ihre Anmerkungen zu meiner Arbeit bedanken. Besonders bedanken möchte ich mich hierbei bei Dr. Paula Herber und Dr. Thomas Göthel. Beide haben mich nicht nur während meiner Promotionszeit durch ihre Bereitschaft zu wissenschaftlichen Diskussionen unterstützt, sondern auch während der Fertigstellung durch wiederholtes Korrekturlesen und hilfreiches Feedback maßgeblich zum Gelingen dieser Arbeit beigetragen haben.

Weiterhin möchte ich meiner Familie und meinen Freunden danken für ihre Unterstützung während der ganzen Zeit. Mein größter Dank gilt hierbei meiner Freundin Kerstin Knebel, die mir in der heißen Phase den Rücken freigehalten und mich immer unterstützt hat. Natürlich möchte ich auch meinen Eltern, Sabine und Lutz Reicherdt danken, die mich auch vor und während meines Studiums sowie meiner Promotionszeit unterstützt haben. Schließlich möchte ich auch meinem Sohn Nils danken, der mich zum einen durch seine anstehende Geburt zu Höchstleistungen angespornt hat, mir aber gleichzeitig genug Zeit gegeben hat, die Arbeit meinen Ansprüchen genügend fertigzustellen.

Robert Reicherdt, Dezember 2015

Contents

1 Introduction **13**

1.1 Problem . . . 13
1.2 Objectives . . . 14
1.3 Proposed Solution . . . 15
1.4 Motivation . . . 16
1.5 Main Contributions . . . 17
1.6 Context of this Work . . . 18
1.7 Overview of this Thesis . . . 18

2 Background **21**

2.1 Model Driven Development and Engineering . . . 21
2.1.1 Model Driven Engineering . . . 21
2.1.2 Model Driven Development . . . 23
2.1.3 Summary . . . 24
2.2 MATLAB/Simulink . . . 25
2.2.1 Simulink . . . 26
2.2.2 Model Elements and Simulation Mechanics . . . 27
2.2.3 Simulation . . . 30
2.2.4 Summary . . . 32
2.3 Program Verification . . . 33
2.3.1 Verification of the Correctness of Programs . . . 33
2.3.2 Automatic and Inductive Verification with Weakest Preconditions . . . 38
2.3.3 Verification with Automatic Theorem Proving . . . 45
2.3.4 Summary . . . 47
2.4 The Boogie Verification Framework . . . 47
2.4.1 Boogie2 . . . 48
2.4.2 The Boogie Verifier . . . 52
2.4.3 Summary . . . 53

2.5 Slicing . . . 53
2.5.1 The Program Dependence Graph . . . 54
2.5.2 Static Slicing with Program Dependence Graphs . . . 56
2.6 Summary . . . 57

3 Related Work 59
3.1 Verification of Simulink Models . . . 59
3.1.1 Verification of Hybrid Simulink Models . . . 59
3.1.2 Verification of Discrete Simulink Models . . . 60
3.2 Slicing of Simulink Models . . . 62
3.2.1 Slicing of State-based Models . . . 63
3.2.2 Slicing of Synchronous Models . . . 65
3.3 Summary . . . 65

4 Automatic Verification Approach 67
4.1 Verification Idea . . . 68
4.2 Verification Framework Architecture . . . 69
4.3 Verification Process . . . 71
4.4 Summary . . . 72

5 Slicing of Simulink Models 73
5.1 Assumptions . . . 74
5.2 Dependence Analysis . . . 75
5.2.1 Data Flow and Data Dependence . . . 75
5.2.2 Control Flow in Simulink . . . 76
5.2.3 Conditional Execution Contexts and Control Dependence 79
5.2.4 Calculation of Control Dependence . . . 81
5.2.5 Summary . . . 86
5.3 Slicing of Simulink Models . . . 86
5.3.1 Building the Dependence Graph . . . 86
5.3.2 Computing the Simulink Slice . . . 87
5.3.3 Example . . . 88
5.4 Increasing the Precision of Slices . . . 90
5.4.1 Routing-aware Dependence Analysis . . . 91
5.4.2 Routing-aware Dependence Graph . . . 95
5.5 Summary . . . 97

6 Transformation of Discrete-Time Simulink Models to Boogie 99
6.1 Overview . . . 99
6.2 Assumptions . . . 101

6.3 A Formal Model for Simulink . . . 103
6.3.1 Modeling Blocks . . . 105
6.3.2 Modeling Signals . . . 107
6.3.3 Block States and Intermediate Results . . . 107
6.3.4 Initialization . . . 107
6.3.5 Modeling Control Flow . . . 108
6.3.6 Data Types . . . 108
6.4 Calculation of the Control Flow Graph . . . 109
6.4.1 How Simulink calculates the Sorted Order . . . 110
6.4.2 Calculating the Sorted Order . . . 110
6.5 Mapping of Simulink Blocks to Boogie2 . . . 112
6.5.1 Sources Block Set . . . 112
6.5.2 Math Operations Block Set . . . 114
6.5.3 Logic and Bit Operations . . . 117
6.5.4 Ports & Subsystems Block Set . . . 118
6.5.5 Signal Routing Block Set . . . 122
6.5.6 Discrete Block Set . . . 125
6.5.7 User-defined Functions Block Set . . . 127
6.5.8 Sinks Block Set . . . 129
6.5.9 Implicit Casts and Mechanisms for Multiple Block Types 130
6.6 Summary . . . 132

7 Automatic Inductive Verification of Simulink Models 133
7.1 Automatic Generation of Formal Specifications . . . 133
7.1.1 Inductive Invariant Checking . . . 134
7.1.2 k-Induction . . . 136
7.2 Slicing for k-Induction . . . 138
7.3 Automatic Generation of Verification Goals . . . 139
7.3.1 Encoding Over- and Underflow Checks . . . 141
7.3.2 Encoding Division-by-zero Checks . . . 141
7.3.3 Encoding Range Violation Checks . . . 142
7.3.4 Other Verification Goals . . . 143
7.4 Loop Invariants . . . 143
7.4.1 Automatically Generated Invariants . . . 144
7.4.2 Manually Specified Invariants . . . 145
7.5 Summary . . . 146

8 Implementation 147
8.1 The MeMo Framework . . . 147
8.2 Parsing Engine . . . 148
8.2.1 Parsing of Model and Inlining of References . . . 149

8.2.2 Enhancement with Run-time Information 149
8.2.3 Preprocessing 150
8.2.4 Persisting the Parsed Model 151
8.3 Slicing Engine 151
8.4 Translation Engine 151
8.4.1 BlockTranslatorFactory 152
8.4.2 ArithmeticsMapping 152
8.4.3 BoogieModel 152
8.5 Verification Engine 153
8.6 Graphical User Interface of the MeMo Tool 153
8.7 Summary 154

9 Evaluation 155
9.1 Case Studies 156
9.1.1 TestModel Design 156
9.1.2 Odometer 157
9.1.3 Distance Warning (DistWarn) 158
9.1.4 Additional Models for Slicing Evaluation 159
9.2 Slicing Evaluation 161
9.2.1 Evaluation of Our Basic Slicing Technique 162
9.2.2 Evaluation of the Routing-aware Slicing Technique . . . 163
9.3 Verification Evaluation 164
9.3.1 TestModel 165
9.3.2 Odometer 168
9.3.3 DistWarn 169
9.4 Summary 171

10 Conclusion and Future Work 173
10.1 Results 173
10.2 Discussion 175
10.3 Outlook 177

List of Figures 181

List of Algorithms 183

List of Tables 185

List of Acronyms 187

Bibliography 189

1 Introduction

Nowadays, *Model Driven Development (MDD)* is a widely used software development technique, in particular for embedded systems. By enabling the specification of (software) systems using domain specific languages, both domain specialists and software engineers can work on the same model of a system. This reduces the development effort and avoids misunderstanding between these two groups. Furthermore, MDD enables the iterative refinement of models down to the implementation level while the models are executable (e. g., by simulation) in each design stage. This enables testing of the models in the early development stages where removing errors is much cheaper than in late stages of the development process. However, it is practically impossible to perform exhaustive testing for real world systems and, hence, testing is always incomplete. Moreover, embedded systems are often deployed in safety critical areas like in automotive or avionics industries, where an error may lead to severe injuries and even to death of persons. As a consequence, comprehensive and *complete* quality assurance measures are required for ensuring the correctness of such systems. Formal verification techniques are complete since they show the correctness of a model for all possible inputs and all possible scenarios. Hence, formal verification techniques that are applicable on models, in particular in early stages of the process, offer a great opportunity to increase the benefit of MDD, namely the early detection of errors, even more.

1.1 Problem

In this thesis, we address the problem of automatically verifying discrete-time Matlab/Simulink models that are used for the specification and development of embedded controllers. Matlab/Simulink [Mat14a] is a de-facto standard for the development of embedded systems, especially in the automotive industry. Over the last decades, more and more functionality, e. g., assistant systems, were added to cars. Also, functionality previously implemented in hardware has been moved to the software level. Both led to an increase in the complexity of embedded systems and the controller software. Since this software is developed using models, the size of such models have increased accordingly. Existing automatic formal verification approaches like model checking suffer of the state space explosion problem. In contrast, existing verification approaches that scale well with large state spaces are often less automated or rely heavily on user-given annotations.

1.2 Objectives

In this thesis, we aim at providing an automated formal verification framework for discrete time MATLAB/Simulink models. Since the simulation semantics of MATLAB/Simulink is only informally defined in [Mat14b], a formal representation for MATLAB/Simulink models is required. This formal representation enables the use of formal verification methods to verify properties of the model, e. g., the absence of certain error classes like *run-time errors*. Both, the transformation as well as the verification of certain properties should be done automatically. Overall, we require our framework to fulfill the following criteria:

- **Automation** To reach a high degree of automation, the transformation of MATLAB/Simulink models into the formal specification has to be done fully automatically. This means that the translation should not require any user supplied annotations.
- **Coverage** Most of the frequently used blocks from the MATLAB/Simulink standard block library have to be supported by the transformation and verification. This mainly comprises blocks from the following block sets: *Discrete*, *Logic and Bit Operations*, *Math Operations*, *Ports & Subsystems*, *Signal Routing*, *Sources* and *Sinks*. The set of blocks should be extensible such that translation rules for further blocks can be easily added in the future.
- **Verification Goals** Verification goals have to be extracted automatically from the model for certain important error classes. Examples are *division-by-zero*, *range violations* and *over-* and *underflows*.
- **Scalability** Translation and verification should be possible on a standard computer system within time bounds acceptable for practical use. Hence, the verification technique that is used shall scale with possibly unbounded state spaces and large models.
- **Comprehensibility** The framework has to preserve the structural information of the original MATLAB/Simulink model. This is crucial to present verification results (like counterexamples) in a human-readable way.
- **Practicability** The framework should be applicable to real world industrial models.

1.3 Proposed Solution

To achieve the objectives defined in the previous section, we propose a novel automatic verification framework for MATLAB/Simulink models. The key idea of our framework is a combination of *inductive invariant verification* using *automatic theorem proving*, more precisely the *Boogie verification framework*[BCD+06], and *slicing* to automatically verify properties of a given model.

In contrast to other automatic verification approaches like model checking or abstract interpretation, inductive verification techniques are less prone to the state space explosion problem since they do not explore state space from an initial state. Instead, the verification is done for a finite number of state transitions from an arbitrary start state. Moreover, the Boogie approach is based on *proof by contradiction* exploiting the power of modern, automatic theorem provers in finding a satisfying assignment. Hence, this technique is less prone to scalability issues and well suited to deal with large and even unbounded state spaces. Furthermore, we use automatic model reduction techniques to reduce the complexity and, hence, the state space of a model to achieve a higher scalability. To this end, we propose a *static slicing* method to reduce the model for a given point of interest, namely a block, and run the verification on the resulting slice.

A prerequisite for the application of formal verification and, hence, for automatic theorem proving, is the existence of a formal specification. Since the semantics of MATLAB/Simulink models as given in [Mat14b] is informal, we propose a transformation from MATLAB/Simulink into the *Boogie verification language (Boogie2)*. Boogie2 is semantically well defined and also serves as the input language to the Boogie verification framework. We use the sequential simulation semantics of MATLAB/Simulink according to a fixed-step discrete solver for our transformation, which matches the sequential semantics of the Boogie verification language. Due to the limitations of Boogie and the underlying theorem prover, some arithmetic operations as well as floating point numbers have to be over-approximated. However, the Boogie verification language is expressive enough to represent most of the block behavior and arithmetics in MATLAB/Simulink.

For both, the transformation as well as the slicing, it is necessary to calculate the data and control dependences in a MATLAB/Simulink model. For the transformation, these dependences are required to calculate the execution order of the blocks within a model according to the simulation semantics. For slicing, these dependences are used to determine the blocks that are part of a slice of a model. Hence, we propose a dependence analysis for MATLAB/Simulink models

To achieve a high degree of automation, both the transformation and the verification of the MATLAB/Simulink models have to be fully automated. To meet this requirement, we introduce an automatic transformation engine that automatically calculates the block order of a MATLAB/Simulink model and translates the model block-wise into the Boogie verification language. For the most common block types and their common parameter combinations, translation templates and rules are specified. In addition to the translation of blocks, we automatically create verification goals for the error classes *division-by-zero*, *range violations* and *over-* and *underflows* depending on the type of the blocks. This enables the automatic verification for the absence of errors from these error classes using the Boogie verification framework. The translation is extensible to further properties.

In summary, we propose a verification method for MATLAB/Simulink which is both: *highly scalable* due to the use of inductive verification techniques and automatic model reduction techniques, and *highly automated* due to the automatic transformation of MATLAB/Simulink models into a formal model in the Boogie verification language and the automatic generation of verification goals. To show the practical applicability, we have developed our *MeMo tool suite*[HWS+11]. Besides parsing and slicing of the models, it also provides a fully automatic *push-button* verification of MATLAB/Simulink models. We evaluate our framework and our tool on a number of case studies supplied by our industrial partners.

1.4 Motivation

The world is smart nowadays. There are smart phones, smart homes, smart watches, smart refrigerators, smart TVs and many more. All these devices are smart due to small integrated computing units and optionally, sensors, actuators and communication interfaces. They contain *embedded systems.*

Embedded systems are everywhere. Besides TVs, mobile phones or refrigerators, where they are used to increase our quality of living, they are also part of cars, trains, aircrafts, traffic management systems and other critical infrastructure. But, in contrast to a TV, a malfunction in a car, a plane or a traffic light control system may lead to costly damages, severe injuries and even to the loss of human lives. The latter class of embedded systems are *safety critical* systems. To ensure the correct functionality of these systems, comprehensive quality assurance methods have to be applied. In practice, this is mostly done by following a well-defined development process and guidelines and by testing.

A widely spread technique for the development of embedded systems, especially in automotive and avionic industries, is *model driven development.* In this technique, models are used as artifacts during all refinement phases of the development process and often, code is generated from the models by automatic code generators. These models are specified in a Domain-specific Modeling Language (DSML) that is tailored to the application domain and to a specific design phase. Often, these models are executable and enable the simulation of the models, which in turn, renders the testing of the models in all stages of the design process possible. With that, errors can be detected in the early design phases, which is usually less costly than in the late phases of the development.

MATLAB/*Simulink* is an integrated development environment widely used in industry for the development of embedded systems. Almost every big automaker and their external suppliers are using MATLAB/Simulink[1]. It provides the graphical DSML *Simulink*, which is tailored to the development of dynamical systems. The DSML is sufficient to model multiple steps of the design process. Simulation and testing of Simulink models is possible for every stage in the development process.

However, to show the absence of errors using testing, *exhaustive testing* (for all possible inputs) would be necessary. Thus, testing is incomplete. In contrast, formal verification techniques are well suited to *guarantee* the absence of errors. But this guarantee comes for a price: Formal verification techniques require a formal representation of the system and they either require a high degree of manual effort and scale well, or they are automated but are subject to scalability issues. This trade-off makes formal verification less attractive for practical use.

In this thesis, we present a formal verification framework that provides an automatic and scalable verification technique for MATLAB/Simulink models and enables the detection of errors in Simulink models, even in the early phases of the design process.

1.5 Main Contributions

The main contributions of this thesis are:

- A **dependence analysis** and a **slicing approach** for MATLAB/Simulink models. This approach enables the static forward and backward slicing of MATLAB/Simulink models for a point of interest in the model, the *slicing criterion*. The approach uses a block as slicing criterion and the calculation of the slices is conservative and sound. Hence, it can be used as an automatic model reduction technique to split the verification problem into smaller sub-problems. We have published this approach in [RG12]

- A **formal semantics** for a subset of Simulink defined by a transformation into the Boogie verification language. This enables the use of the Boogie verification framework for the verification of MATLAB/Simulink models. The transformation is fully automatic and additionally to the semantically equivalent Boogie program, verification goals for common run-time errors are generated during the translation for all block types prone to these specific classes of errors. We have published this approach in [RG14a].

- An **automatic verification framework** for MATLAB/Simulink with tool support that shows the practical applicability of our approach and is integrated in our *MeMo tool suite*[HWS+11, RG14b].

[1]http://www.mathworks.de/automotive/userstories.html

- A demonstration that our framework can be **applied to** automatically verify a number of **industrial case studies**. These case studies are an *odometer* and a *distance warning* system.

1.6 Context of this Work

This work has taken place in context of the MeMo[2] project [HWS+11], which was funded by the *Investitionsbank Berlin* within the subsidy program for research, innovation and technology (Pro FIT). The project was carried out by the chair of *Software Engineering for Embedded Systems* and two industrial partners: *Berner & Mattner Systemtechnik GmbH* and *Model Engineering Solutions GmbH*. Within this project we have developed novel methods and analyses to ensure and asses the quality of Matlab/Simulink models. Both companies are operating in the automotive area and offer consulting and quality assurance services to automakers and external suppliers. The key idea of the MeMo project was to provide both, analytical as well as constructive methods to ensure the quality of Matlab/Simulink models. As a prerequisite, we have developed a parsing framework and a generic intermediate representation that serves as a basis for all further analyses. To improve the scalability of the analyses, we have developed a novel slicing approach [RG12] for Matlab/Simulink models. As analytical method, we have developed a method for the automatic detection of certain error classes: a verification framework for Matlab/Simulink using the Boogie verification framework [RG14a]. As constructive method, a quality model and metrics have been developed [HLW12] for Matlab/Simulink.

1.7 Overview of this Thesis

This thesis is structured as follows: In Chapter 2, we provide an overview of the background of this thesis. We start with an introduction of the general concepts of Model Driven Development and give a brief introduction into the Matlab/Simulink tool suite and modeling notation. Then, we introduce the weakest precondition predicate transformer that, together with inductive techniques and automatic theorem proving, is used for the automatic verification of programs. Furthermore, we describe the Boogie verification framework that provides a formal language and a tool that we use in our approach. Finally, we briefly introduce program slicing. In Chapter 3, we discuss the related work of this thesis. In Chapter 4, we present our approach for the automatic formal verification of discrete-time Matlab/Simulink models that is based on an automatic transformation of Matlab/Simulink models into the Boogie2 verification language and uses slicing as an automatic model reduction technique to improve the scalability. In Chapter 5, we present our slicing approach for Matlab/Simulink models based on dependence graphs constructed

[2]Methods of Model Quality

using our dependence analysis. In Chapter 6, we present our transformation of discrete-time Matlab/Simulink models into our formal model in the Boogie2 verification language where we also use the the dependence analysis to calculate the control flow graph for a model. In Chapter 7, we present our automatic verification approach based on inductive techniques and verification goals and invariants that we automatically generated during the translation of models. In Chapter 8 we give a brief overview of our implementation of the framework before we present our experimental results in Chapter 9 where we have applied our framework to some case studies. In Chapter 10, we conclude this thesis and discuss further directions for future work.

2 Background

In this chapter, we give background information regarding the languages and techniques used in this thesis and briefly introduce the tools used in our automatic verification framework for discrete-time MATLAB/Simulink models. First, we briefly introduce the paradigm of Model Driven Development in Section 2.1. Then we briefly describe the MATLAB/Simulink tool chain and the Simulink graphical modeling notation as an industrially used Model Driven Development approach in Section 2.2. Furthermore in Section 2.3, we give an overview to program verification with predicate transformers and introduce efficient techniques for the verification of programs using the weakest precondition predicate transformer and automatic theorem proving. Subsequently, we introduce the verification tool and language used in our approach. We give an introduction to the Boogie verification language and Boogie tool in Section 2.4. Finally, we give a brief introduction into static slicing techniques in Section 2.5.

2.1 Model Driven Development and Engineering

In software development, models are used for various purposes. Models provide an abstraction of the reality with respect to some specific properties of interest. They are well suited to support communication between domain and software specialists or to enable the (partial) specification of a system or its behavior. In software engineering, models are traditionally used to describe certain features of a system like the architecture, the behavior, the components or the interfaces. However, models are not a driving element in the traditional software development process.

2.1.1 Model Driven Engineering

In Model Driven Engineering (MDE) [Ken02, Sch06], models are used as a central element in the development process. MDE is a generalization of Model Driven Architecture (MDA)[MM$^+$01]. MDA was introduced by the Object

Management Group (OMG)[1]. With MDA, the OMG defined an approach for the platform-independent specification of IT-systems based on the OMG-ecosystem of modeling languages (e. g., Unified Modeling Language (UML),) and middle-ware solutions. MDA separates models into three levels of abstraction:

- **Computation Independent Model (CIM)** The CIM is the layer for the domain practitioner and does not contain any information about the underlying IT-system.
- **Platform Independent Model (PIM)** The PIM is the layer that includes information about the underlying IT-system. However, the model is general enough to be applicable to any possible IT-architecture.
- **Platform Specific Model (PSM)** The PSM is the layer where specific information about a particular IT-architecture is included and used.

Furthermore, transformations are defined for refinement and abstraction between the layers (e. g., $PIM \to PSM$, $PIM \to PIM$, etc.).

More generally as proposed by Kent [Ken02], MDE-approaches have to specify

(1) modeling languages and models,

(2) transformations between models (and languages) and

(3) a process that supports and coordinates the construction as well as the evolution of the models.

(4) Furthermore, an adequate tooling needs to be present for a specific MDE approach

The tools have to support the correct syntactical use of the modeling languages, the transformation between models, the process in general and model-based testing, e. g., using the model to specify the inputs to the system. Finally, the tools have to support the working with instances of models, e. g., by simulation.

Domain Specific Modeling Languages

Since different application domains require models and processes that are especially tailored to the specific constraints and characteristics of the domain, Domain-specific Modeling Languages (DSMLs) are used to model the systems. DSMLs are either textual or graphical languages that, in contrast to *General Purpose Languages*, offer specialized features for their particular domain, but are of limited use for modeling systems outside the domain.

[1]http://www.omg.org

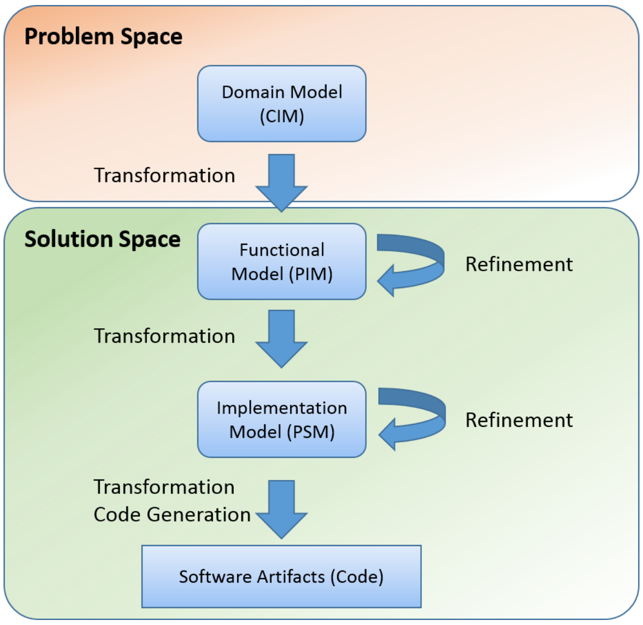


Figure 2.1: The MDD Process

2.1.2 Model Driven Development

Model Driven Development (MDD) is a software development technique where models are the central artifacts in the software development process. The key idea of MDD is a process where, starting from an abstract model, model transformations are iteratively applied to the model. Finally, the implementation is generated from the models. In [BCW12], MDA is classified as a particular instance of MDD (for the OMG-ecosystem). MDD is a subset of MDE since MDE enables more general engineering than only for software.

The MDD Process

The generic process of MDD is depicted in Figure 2.1. Here, we differentiate between the *problem space* where a specific problem exists in its natural domain, and the solution space where the problem is mapped to a computational solution, e. g., a software system. In the problem space, the problem is modeled in its particular domain. The resulting model (CIM) is a physical or mathematical model of the system, e. g., a differential equation, usually designed by a domain specialist. Then, the model is transformed into a functional model (PIM) where a particular (computational) approach is chosen to solve the problem, e. g., an approximation technique for the differential equation. With this transformation, the solution space is entered. The PIM then may undergo a number of refinement (e. g., refinement of the approximation technique) and abstraction (e. g., different views of the system) steps. However, at this point the model does not contain any implementation details and the model (and the modeling language used) is focused on functional aspects of the system. Subsequently, the model is transformed into an implementation model (PSM) where platform and implementation specific details are set. Again, on the PSM-layer, refinement and abstraction steps are possible to model different aspects of the system. Finally, the implementation model is transformed into source code and other artifacts. Transformation steps are either performed manually or automatically depending on the tool support. In practice, automatic code generators are widely used for the last transformation step.

Besides the process that describes the iterative refinement of models from CIM to PSM, MDD also enables quality assurance techniques (especially testing) in the early development stages.

Simulation, Testing and Verification

Besides syntactical checks, editing support and the automation of transformations between or within the modeling languages, MDD tools need to be able to execute the models. Selic [Sel03] names the executability of models as an important factor for the success of MDD since it enables understanding the system by experimentation. Furthermore, the tools need to provide a run-time environment where the simulation of the models can be started, stopped or resumed and steered with input sequences to the models.

The ability to simulate models with specific inputs or input sequences results in an important benefit of MDD: *The models are testable.* The correct functionality of a system can be tested at early stages in the development process. Moreover, the test cases (e. g., input sequences for the models) may be reused in later refinement stages. If the simulation environment supports the integration of (legacy) code, it can be reused even for testing the final software artifacts. Many testing techniques transferred from programing languages can be applied to the model level. Tests and test sequences showing the *correct functionality* of the model can be derived from the requirements specification. The input vectors can be tested systematically using techniques like *equivalence partitioning* and *boundary value analysis.* Furthermore, the models can be tested for run-time errors and domain specific errors. Even *agile* development techniques like *test-driven modeling* [Zha04] are possible where tests are created prior to the actual modeling of the system.

However, testing mostly covers the possible inputs for a model only partially and, hence, it is inherently incomplete. To show the absence of run-time errors or safety properties, complete techniques like formal verification are needed. However, due to the fact that different MDD approaches consider different modeling languages with different execution semantics, formal verification approaches have to be especially tailored to particular modeling languages.

2.1.3 Summary

In this section, we have briefly introduced Model Driven Development as a subset of Model Driven Engineering. Furthermore, we have presented the

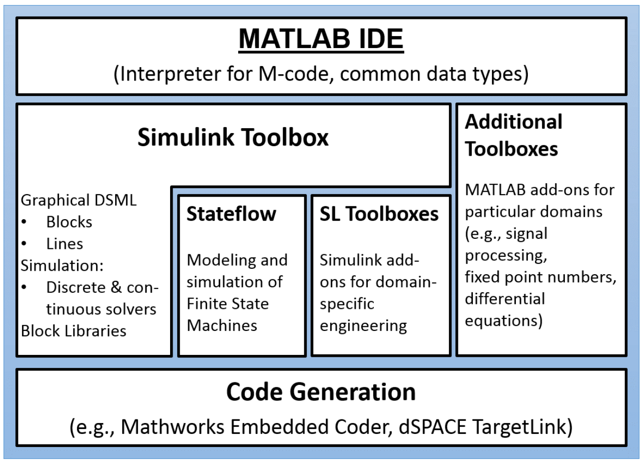

MATLAB IDE
(Interpreter for M-code, common data types)
Simulink Toolbox
Graphical DSML
• Blocks
• Lines
Simulation:
• Discrete & continuous solvers
Block Libraries
Stateflow
Modeling and simulation of Finite State Machines
SL Toolboxes
Simulink add-ons for domain-specific engineering
Additional Toolboxes
MATLAB add-ons for particular domains (e.g., signal processing, fixed point numbers, differential equations)
Code Generation
(e.g., Mathworks Embedded Coder, dSPACE TargetLink)

Figure 2.2: The MATLAB Architecture

generic process of MDD and the different levels (and views) of abstraction. Finally, we have presented the benefits of MDD especially for quality assurance in the early stages of the development process and motivated the need for verification techniques tailored to the specific modeling languages. In the next section, we introduce the MDD tool MATLAB/Simulink. It provides both, the graphical modeling notation *Simulink* and the MATLAB/Simulink tool that enables the editing and simulation of Simulink models.

2.2 MATLAB/Simulink

MATLAB by Mathworks[2] is a tool that enables numerical computing and visualization of mathematical problems. The core features of the MATLAB tool are matrix manipulations, numerical simulations, plotting of functions and data, programming and development of algorithms and even the development of user interfaces. The name MATLAB originates from the first: *MATrix LABoratory*. For the development of algorithms and user interfaces, a programming language MATLAB Code (M-code) is provided and interpreted by the MATLAB environment. However, MATLAB also offers interfaces or native support for other programming languages like *C/C++* or *Java*.

The functionality of MATLAB can be extended by add-ons, which are called *toolboxes*. Toolboxes exist for various purposes (e. g., image processing, statistics, finance, biology, control systems) and offer functions, user-interface elements and data types for the particular domain. Figure 2.2 shows the architecture of the MATLAB environment. These toolboxes either extend the functionality of the MATLAB tool or further extend a certain toolbox. For example, the *Simulink* toolbox extends MATLAB by editing and simulation facilities while the *Stateflow* toolbox extends Simulink. Finally, there exists a number of toolboxes for automatic code generation. In the following sec-

[2]http://www.mathworks.de/products/matlab/

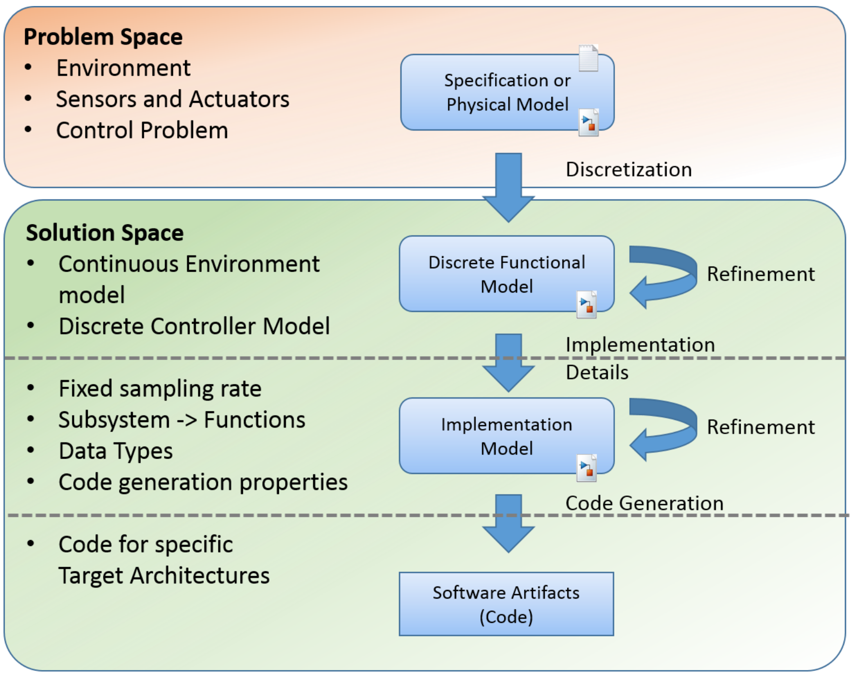


Figure 2.3: The MDD Process in Simulink

tion we introduce the Simulink toolbox and notation which provides an MDD environment for the domain of embedded controllers.

2.2.1 Simulink

Matlab/Simulink [Mata, Matb] is a particular toolbox that enables the graphical modeling and simulation of synchronous reactive embedded systems. The Simulink toolbox provides the graphical DSML Simulink, an editing framework for the Simulink models and a simulation engine that is able to simulate the model with sampling or different numeric techniques.

The DSML Simulink is a data flow oriented block diagram notation which consists of blocks and lines. Blocks represent either some kind of functionality, like mathematical or logical functions, or are used for structuring the model. Lines are used to represent signal flow between blocks.

Matlab/Simulink provides a block library of about 300 predefined blocks. Further toolboxes for Matlab/Simulink extend these libraries by adding blocks for particular domains, e. g., the *Aerospace Blockset* or the *SimHydraulics* toolboxes. Moreover, the *Stateflow* toolbox adds a whole event- and state-based modeling language to Matlab/Simulink, which enables the modeling of complex decision logic in Simulink models using finite state machines.

Matlab/Simulink can be used to model both, *discrete* embedded controllers as well as *continuous* physical environments the controller is deployed on. It enables the model driven development for the software of embedded controllers. Figure 2.3 depicts the MDD process for Matlab/Simulink. Analogously to the generic MDD process, the control problem is specified in its particular domain. To this end, MATLAB/Simulink offers a number of blocks representing common (continuous) functions from control engineering and blocks for modeling (or approximate) the continuous time environment and sensors. Then, the model is refined using discretized blocks representing the discrete version of the functions from control engineering. Note that MATLAB/Simulink is always able to simulate continuous and discrete parts of the model together. With subsequent refinement steps, discrete sampling rates are defined for the controller and more implementation details are added to the controller. Finally, code is generated for the refined model.

In the next two sections we give a brief introduction to the model elements and the simulation of models in Simulink. Note that a more detailed discussion about block semantics and simulation semantics is given in Chapter 6.

2.2.2 Model Elements and Simulation Mechanics

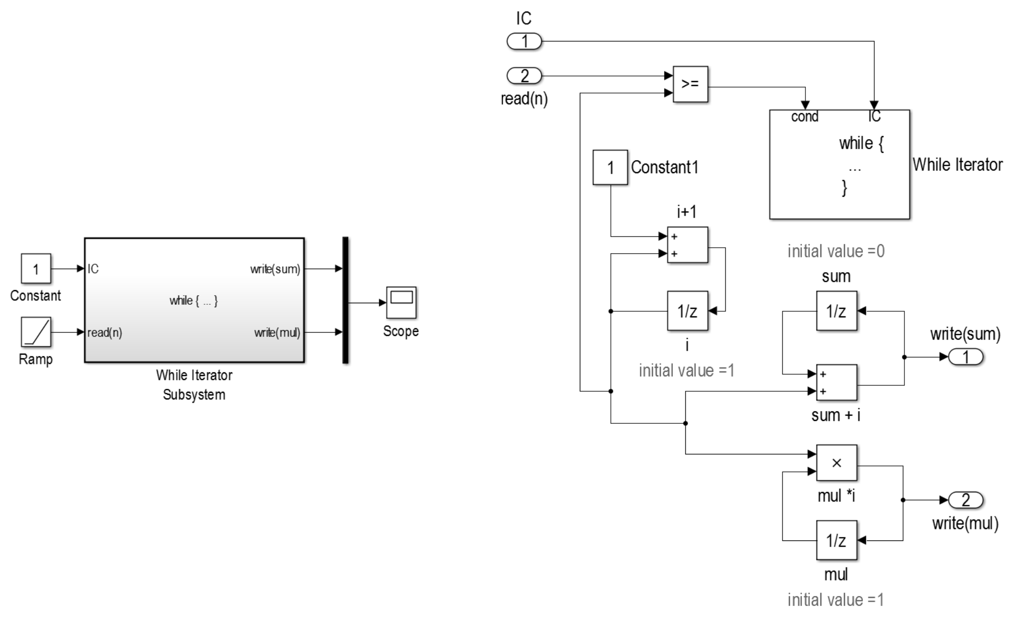
In this section, we briefly introduce the elements of a MATLAB/Simulink model. Furthermore we describe concepts and mechanics used for the simulation of the models by the MATLAB/Simulink tool.

Figure 2.4 depicts a Simulink model that calculates the product and sum of the natural numbers up to a bound n, which in this example is given by a `Ramp` function that increases n in every simulation step. The sum and product are calculated whitin the `WhileIterator` subsystem in every simulation step and displayed using a `Scope` block. We use this model to explain the elements and concepts in the following sections.

We use a `WhileIterator` subsystem to calculate the sum and the product of the first n positive integers up to a given n. A `WhileIterator` subsystem executes its contents as long as the signal supplied to the *cond* port is greater than zero. A `WhileIterator` requires an *initial condition* (*IC*) which is used to determine if the subsystem is executed at least once. This *IC* has to be supplied from outside of the `WhileIterator` subsystem. Therefore we have embedded the `WhileIterator` subsystem in an environment (Figure 2.4a) where n is provided by a `Ramp` function which increases n by 1 in each simulation step and *IC* is a `Constant`. Figure 2.4b depicts the contents of the `WhileIterator` subsystem. We have named the blocks corresponding to the statements of the example program. The initialization of the variables is done using the initial values of the `UnitDelay` ($\frac{1}{z}$) blocks. A `UnitDelay` block holds the value of a signal for a simulation (or loop execution) step. We have named the blocks within the model corresponding to variables and statements that would have been used in an imperative program.

Blocks

In Simulink, the behavior and appearance of block is defined by its type and its block parameters. The parameters consist of a set of mutual parameters,



(a) Top Level of the Model **(b)** Contents of the Subsystem

Figure 2.4: A Example Simulink Model

e. g., color, size, position, data type, which can be found in every block, and a number of parameters that are specific to the block type. While the basic behavior is given by the type, the block parameters influence the behavior of a certain block instance. Furthermore, blocks may have an internal state that changes during the simulation. Blocks can be classified with respect to the following properties:

- **Functional vs. Virtual** A block either models some functionality or is used for structuring the model. Blocks that do not model functionality are called *virtual* blocks. Virtual blocks normally do not influence the behavior of the model. However, some special parameter configurations may cause a virtual block to become *non-virtual*. We refer to non-virtual blocks also as *functional* blocks. In Figure 2.4a, the black bar is a `Mux` (multiplexer) block that combines the two input signals into one signal. This block and the port blocks with the rounded corners are only used for signal routing. They are virtual blocks. All other blocks influence the behavior of the model and, hence, are functional blocks.
- **Direct-feedthrough vs. Non-direct-feedthrough** In Simulink, the blocks are classified the way their outputs are calculated. If the output of a block is directly dependent on one of its input, it is a *direct-feedthrough* block. If the output is only dependent on the state, this means the inputs are only used to calculate the next state, it is called a *non-direct-feedthrough* block. Note that this property is also affected by certain parameter configurations. In Figure 2.4b, the blocks labeled “`i+1`”, “`sum+i`” and “`mul*i`” are *direct-feedthrough* blocks since they di-

rectly output the sum or product of the values supplied to their inputs. The `UnitDelay` blocks (`1/z`) store the value supplied to their input for one simulation step and output the value of the step before. Hence, they are *non-direct-feedthrough* blocks.

- **Composite vs. Basic** The block libraries do not only provide blocks for domain specific tasks but also a number of *composite* blocks. These blocks provide a (parameterized) combination of other blocks to realize commonly used tasks. *Basic* blocks are not composed from other blocks. In Figure 2.4a, the `Ramp` block is a composite block since it is composed from other basic blocks. The `Sum` block labeled with “`i+1`” is a basic block.

Hierarchy

Matlab/Simulink enables hierarchical structuring of the model with `Subsystem` blocks. Subsystems can contain other blocks and even further subsystems. However, on the lowest level of this nested structure of subsystems there are only basic blocks. This means that composite blocks are realized by hierarchical subsystems with parameters used by underlying basic blocks. Subsystem are either *atomic* or virtual. Virtual subsystems are practically invisible to the Simulink scheduler and do not influence the behavior of the model. However, they are still useful to visually structure the model. Atomic subsystems do impact the behavior since they require all contained blocks to be executed in an atomic operation. The `WhileIterator` subsystem in Figure 2.4a is an atomic subsystem, which is indicated by the thick borders.

Masks

Subsystems can be annotated with masks. A mask is specified in the block parameters and can be used to change the visual appearance of subsystems and to specify additional parameters. Masks are used for composite blocks where some functionality is encapsulated in a subsystem, e. g., a block from a library, to configure the behavior. For example, the `Ramp` block in Figure 2.4a uses a mask to specify the slope, the start time and the initial value of the function implemented by this composite block.

Lines and Signal Flow

Signal flow is modeled using lines in Simulink. Lines connect the outputs of a block with the inputs of other blocks. However, a line does not necessarily model one signal. Instead, lines may model a scalar signal, a vector of signals, a matrix signal and even signals with multiple hierarchy levels (nested signals). The signals carried by a line are inferred by the Matlab/Simulink tool at run-time. In Figure 2.4a, the line connecting the `Mux` block with the `Scope` block is actually carrying two signals.

To model the signal flow into subsystems, the inputs and outputs of the subsystem blocks are mapped to (`Inport` and `Outport`) port blocks within the subsystem. `Inport` and `Outport` blocks usually have rounded corners and a number that indicates the corresponding input of the subsystem in top-down direction. For example, the input `read(n)` of the subsystem in Figure 2.4a is mapped to the `Inport` block labeled "`read(n)`" in Figure 2.4b. Since "`read(n)`" is the second input of the subsystem, the `Inport` is labeled with the number "2".

Control Flow

Control flow in Simulink models is realized by conditionally executed subsystems. Besides port blocks, these subsystems contain special blocks that trigger the conditional execution. These blocks model either some iterator functionality (like the `While Iterator` block in Figure 2.4b) or provide special port blocks that trigger the execution if a certain condition is satisfied by the connected signal.

Moreover, control flow may be introduced by certain blocks and parameters. We give a detailed description to that mechanisms in Chapter 5 when we introduce control dependence. In the next section, we briefly introduce the general simulation semantics of Simulink models and the concept of execution contexts, which is vital for the understanding of control flow.

2.2.3 Simulation

Even though the data flow oriented notation of Matlab/Simulink is generally concurrent, Matlab/Simulink executes the blocks *sequentially* in a simulation. Hence, the simulation engine schedules the blocks to determine an execution order. To this end, Matlab/Simulink uses so-called Execution Contexts (ECs).

Definition 2.1 (Execution Context)**.** *An execution context is comparable to a sorted list of blocks and child execution contexts that have to be executed as an atomic operation. This means that once an execution context is entered, all blocks and nested execution contexts have to be executed before the execution is allowed to return to the parent execution context. In* Matlab*/Simulink, every atomic subsystem has its own execution context.*

Simulation Phases

Generally, the simulation of a Matlab/Simulink model is performed in two phases: An *initialization* phase and a *simulation* phase. Figure 2.5 briefly depicts these two phases. The simulation engine that performs the either discrete or continuous simulation is called *solver*. Note that the initialization phase is basically the same for all solvers and the simulation phase differs only in one step between some solvers.

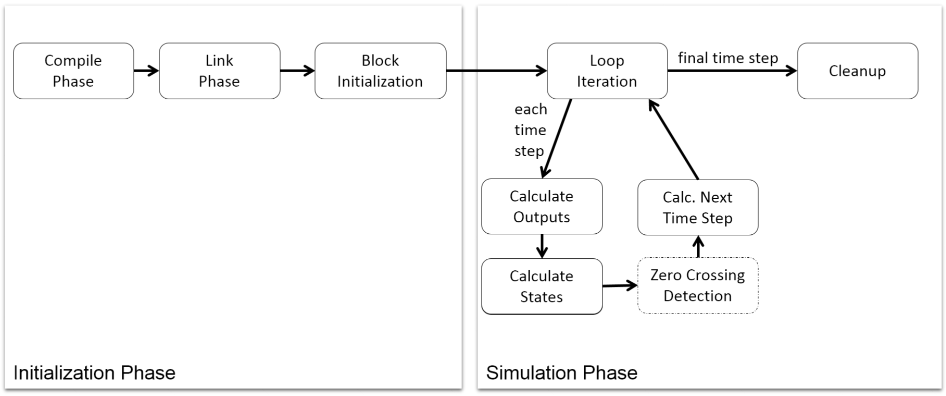


Figure 2.5: The Simulation Phases in Simulink

The *initialization* phase mainly consists of three subphases:

- **Compile Phase** In the *compile* phase, a number of analyses are applied to the model to determine the sorted order of the blocks and information not stored explicitly in the model. Such information are, e. g., data types, signal dimensions or sample times. Furthermore composite blocks from libraries and model references are integrated and the hierarchy is flattened.
- **Link Phase** In the *linking* phase, binaries for external code are linked into stubs in the model. External code may be introduced with so called `S-Function` blocks. These may integrate legacy components but also may be a compiled finite state machine modeled with the Stateflow toolbox.
- **Initialization of Blocks** In the *blocks initialization* phase, data structures are generated for the blocks and their fields are set to initial values according to the block parameters.

Once the initialization phase is finished, the *simulation* phase is started. In this phase, an overall simulation loop executes the blocks in the sorted order for number of time instants up to a specified upper bound. The simulation loop performs the following steps:

- **Calculate Outputs** First, for every block the new output for the current simulation step is calculated. Stateful blocks may use their state for the calculation.
- **Calculate States** Second, new states for the stateful blocks are calculated using the newly calculated outputs of all blocks.
- **Zero Crossing Detection** This step is optional and only relevant for solvers that use a variable step size. It is a technique to determine when a relevant event (*a zero crossing*), e. g., a sign change, occurs within the current time instant and the next time instant using the current step size.

- **Calculate Next Time Steps** Finally, the next time instant is calculated. If a zero crossing has been detected, the step size to calculate the next instant is reduced.

Note that the simulation steps in Simulink are *delta cycles* and the execution of blocks does not consume simulation time. This also holds for iterator systems like the `WhileIterator` subsystem form our example.

Solver

Matlab/Simulink provides various simulation engines to simulate a model. These simulation engines, so-called *solver*, provide a number of techniques for the simulation of discrete and continuous time models. Most of the solvers implement techniques for the numerical solving of ordinary differential equations to simulate continuous time models. To speed up simulation, many solvers use a variable step size to determine the points where the model is sampled. However, it is also possible to simulate the model by sampling the model at a fixed rate, which corresponds to the discrete execution of the controller software on an embedded hardware (fixed-step discrete solver).

Workspace

The Matlab/Simulink tool suite has a global area where variables can be defined and used during the simulation. This area is called *workspace*. The variables can be loaded into the workspace by call-back functions in the model file, by executing scripts written in M-code or by loading the variable from files where they are stored in some binary format. Variables defined in the workspace are often used as parameters to configure a model or to provide input to the model during the simulation. The values of these variables can be extracted from the tool at run-time.

2.2.4 Summary

In this section, we have given an introduction to the Matlab/Simulink environment. First, we have presented the general process for the development of embedded systems with Matlab/Simulink and its relation to the MDD process. Then, we have briefly introduced the model elements and concepts in Simulink. Finally, we have given a short overview of how models are executed by the Matlab/Simulink environment. We give a more thorough description of model elements and simulation semantics in Chapter 5 where we introduce our slicing approach and Chapter 6 where we explain the semantics of model elements by taking the example of our translation into the formal model for the verification of Matlab/Simulink models.

2.3 Program Verification

In this section, we briefly introduce and motivate techniques that we use in our approach, which enable the verification of properties for structured programming languages. The techniques are based on an *axiomatic* approach for specifying and verifying programs with assertions over program states. More precisely, *backward reasoning* with *predicate transformers* is used to transform a program into a set of propositional and *first order logic* formulas, so called *verification conditions* that can be used to verify the program. Finally, automatic theorem proving (*Satisfiability Modulo Theories*) is used to solve the *verification conditions* to verify that the properties are satisfied (or report them being unsatisfiable). For a detailed introduction we refer to [BM07] and [KS08].

2.3.1 Verification of the Correctness of Programs

In program verification, the correctness of a program is shown by verifying that the program satisfies its (formal) specification. The specification of a program formally describes the desired behavior, output or program state depending on the verification technique. Programs can be either be *partially correct* or *totally correct*. *Partial correctness* is the property of programs that ensures that certain (error) states do not occur in any possible execution of a program. However, this does not necessarily mean that a program terminates. *Total correctness* not only ensures partial correctness, furthermore it also ensures that the program terminates.

Various techniques exist to formally verify the correctness of programs. Some techniques, like abstract interpretation, explore the state space of programs using (symbolic) simulation and abstraction techniques. Other techniques use an axiomatic approach where assertions are used to specify preconditions and postconditions for the states before and after the execution of a program (or statement).

In the subsequent sections, we use a *While language* as an example for a higher level imperative programming language. The syntax of this language is defined as

$$\begin{aligned}
\mathrm{S} ::= \;& \mathrm{skip} \mid \mathrm{x{:=}e} \mid \mathrm{S;S} \\
& \mathrm{if\ (b)\ then\ \{S\}\ else\ \{S\}} \mid \mathrm{while\ (b)\ \{S\}} \\
\mathrm{e} ::= \;& \mathrm{e}\ op_n\ \mathrm{e} \mid \mathrm{n} \mid \mathrm{x} \\
\mathrm{b} ::= \;& \mathrm{true} \mid \mathrm{false} \mid \mathrm{b}\ op_b\ \mathrm{b} \mid \mathrm{e}\ op_p\ \mathrm{e}
\end{aligned}$$

where `skip` is the empty instruction, `n` is a numerical literal and `x` is a variable name. Moreover, there are numerical operators op_n, logical operators op_b and comparison operators op_p that can be applied to literals and variables and finally, there is the assignment operator "`:=`" that sets a variable to a value or expression. The semantics for the *if-then-else* and the *while*[3] statement are analogously defined like in other high level languages.

In our verification approach for MATLAB/Simulink models, we use an axiomatic approach based on preconditions and postconditions over the states of the models. Hence, we introduce the foundations for our technique in the next sections.

The Hoare Calculus

An axiomatic technique for the verification of programs is the *(Floyd-)Hoare caluclus*, which was originally defined for flow charts by Floyd [Flo67] and generalized for programs by Hoare [Hoa69]. The basic idea is to specify programs using assertions over the program states before and after the execution of the program. This is done using a *Hoare-triple*

$$\{P\}\ C\ \{Q\}$$

where C is a program, P is the *precondition* and Q is the *postcondition*. The precondition and postcondition are functions that map the state of a program to a truth value. The state of a program is a specific assignment of variables used by the program at a point in the execution. A program is correct if the following holds: If P is satisfied in some state s and C is executed on s and terminates in a final state s', then Q is satisfied in s'. To obtain a proof system from this specification, first a formal syntax for the programming language has to be defined. Second, axioms for the basic instructions of the programming language and inference rules are defined. This results in a deductive proof system where further theorems can be derived using the axioms and inference rules.

As an example for an axiom, we present the *Axiom of Assignment*

$$\{Q[x \leftarrow e]\}\ x := e\ \{Q\}$$

where `x := e` is the assignment of an expression e to the variable x and $Q[x \leftarrow e]$ is derived from Q by substituting all occurrences of x with e. Axioms for the specific instructions of a programming language can be derived from this rule. The `skip` instruction of our while language leads to the derived rule $\{Q\}skip\{Q\}$ since the program state is not affected by this statement.

An inference rule takes the form

$$\frac{X_1\ \wedge\ X_2 \ldots\ \wedge\ X_n}{Z}\ Cond$$

where X_1 to X_n are assertions that form a premise from which, if all, X_1 to X_n, have been proven valid, Z can be concluded to be valid too. Furthermore, a

[3]While is sufficient since `do ... while` and `for` loops can be transformed into a semantically equivalent `while` loop.

side condition *Cond* can be specified that has to be satisfied. As an example of an inference rule, we present the *Rule of Consequence*:

$$\frac{P' \longrightarrow P \ , \ \{P\}\ C\ \{Q\} \ , \ Q \longrightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

This rule describes that a precondition can be *strengthened* and a postcondition can be *weakened*. More precisely, if C is executed in a state that satisfies the precondition P and produces a state where the postcondition Q is satisfied, then, for a precondition P' that implies P but is more restrictive, the postcondition Q is also satisfied. Analogously, if a postcondition Q is satisfied for after the execution of C on a state satisfying P and Q implies a postcondition Q' that is less restrictive, Q' is also satisfied after the execution of C. This rule is basically the key in any Hoare-style proof system since it can be used to glue the single components of the proofs together by enabling the consolidation of preconditions and postconditions.

Furthermore, the Hoare calculus specifies a rule for the verification loops (like *while* statements): The *Rule of Iteration*. The basic idea is the following: If there exists a *loop invariant* that is satisfied before entering the loop and after (but not necessarily during) each execution of the loop, and is also satisfied once the loop is finished. Then, it is suitable to show that the invariant is satisfied by a state before the execution of the loop body and afterwards. The *Rule of Iteration* for a `while` loop in the Hoare calculus is defined as

$$\frac{\{P \wedge I\}C\{I\}}{\{I\}\textbf{while } P \textbf{ do } C\ \{\neg C \wedge I\}},$$

where P is the condition that has to be satisfied for the execution of the loop body C and I is the loop invariant. Note that the rule is only suitable to show partial correctness, since the loop may fail to terminate.

While the Hoare calculus is well suited to show the correctness of programs, it suffers in terms of automation since one is required to specify complete preconditions and postconditions and the sophisticated application of the inference rules to construct a proof. Hence, we use the *weakest precondition* calculus in our approach that enables a higher degree of automation.

Weakest Preconditions

Weakest Precondition (WP) is a predicate transformer introduced by Dijkstra [Dij75] that reduces the verification problem for structured, imperative programs to a problem of reasoning over a set of formulas in First Order Logic (FOL). WP enables a higher degree of automation compared to the Hoare logic since there exist powerful decision procedures for subsets of FOL that are used in automatic theorem provers to solve FOL formulas without the need for human interaction.

A predicate transformer p is a function that maps a formula given in FOL and some statements S to an expression in FOL:

$$p : FOL \times S \longrightarrow FOL$$

The weakest precondition $wp(Q,S)$ is a predicate transformer that maps a postcondition Q expressed in FOL to a precondition. This precondition is the *weakest* precondition such that, if S is executed on some states satisfying this precondition, the postcondition Q is satisfied by the final program state.

Definition 2.2 (Weakest Precondition)**.** *The weakest precondition $wp(Q,S)$ for a postcondition Q and the statements S is defined such that*

- *for a postcondition Q that is satisfied by the final states after the execution of S,*
- *$wp(S,Q)$ returns the least restrictive precondition P that has to be satisfied by the states before the execution of S,*

such that Q is satisfied by the final states.

With the weakest precondition a given Hoare-triple $\{P\}\ S\ \{Q\}$ is provable w. r. t. total correctness if the FOL formula

$$P \longrightarrow wp(S,Q)$$

is *valid*. Such a FOL formula is called Verification Condition (VC).

Similarly to the Hoare calculus, WP is defined on a formal syntax for the programming language and there are rules for the basic instructions of the language. The WP transformer has the following properties by definition:

(1) (*Law of the Excluded Miracle*) For any statement S holds $wp(S, false) = false$.

(2) For any statement S and any postconditions Q and R holds

$$\frac{Q \longrightarrow R}{wp(S,Q) \longrightarrow wp(S,R)}.$$

This property is related to the *Rule of Consequence* of Hoare logic and enables the weakening of postconditions.

(3) (*Conjunction*) For any statement S and any postconditions Q and R holds

$$(wp(S,Q) \wedge wp(S,R)) = wp(S, Q \wedge R)$$

(4) (*Disjunction*) For any *deterministic* statement S and any postconditions Q and R holds

$$(wp(S,Q) \vee wp(S,R)) = wp(S, Q \vee R)$$

For WP, the set of semantic rules for the instructions of our While language is defined as follows:

(1) (*Empty statement*) The rule for the empty statement, e. g., the `skip` statement, is defined as

$$wp(skip, Q) = Q\ ,$$

since the state of the program is not modified.

(2) (*Assignment*) The semantics of an assignment statement `x:=e` is defined as

$$wp(x := e, Q) = Q[x \leftarrow e]$$

where $Q[x \leftarrow e]$ is derived by substituting every occurrence of x in Q with the expression e.

(3) (*Concatenation*) The semantics of the concatenation, e. g., the "`;`" in most programming languages, is defined as

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)).$$

(4) (*Conditional*) The semantics or an instruction for the conditional execution of different branches like an "`if C then` S_1 `else` S_2" is defined as

$$wp(\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2, Q) = (C \longrightarrow wp(S_1, Q)) \land (\neg C \longrightarrow wp(S_2, Q)). \tag{2.1}$$

Together with the properties presented in the subsection before, the application of these rules enables the construction of FOL formulas for any loop-free, structured and imperative program and its verification.

Weakest Precondition for Loops

To verify loops, WP uses a similar approach like the Hoare calculus. Since WP are defined to show total correctness of programs, an additional *loop variant* is required for the verification of loops to show termination. However, there exist application areas where termination is not desired, e. g., reactive systems modeled in MATLAB/Simulink. Hence, we use the Weakest Liberal Precondition (WLP) predicate transformer for partial correctness in our approach. In WLP, the rules are basically the same as in WP except for the loop rule, where the loop variant is removed. This enables the verification of programs for partial correctness.

The rule for the weakest liberal precondition of a *while* loop is defined as

$$\begin{aligned} wlp(\textbf{while } C\ \{I\} \textbf{ do } S, Q) \ = \ & I \ \land\ \forall y, ((C \land I) \longrightarrow wp(S, I))[x \leftarrow y] \\ & \land\ \forall y, (\neg C \land I) \longrightarrow Q))[x \leftarrow y] \end{aligned} \tag{2.2}$$

where $\{I\}$ is the loop invariant, C is the loop predicate and Q is the postcondition. Moreover, x is a set of variables of the program state modified by the statement S and y is a fresh set of variables (with arbitrary assignments). In other words, the loop invariant I must hold initially. Moreover, if the loop invariant I and the loop predicate C hold for an arbitrary program state y, the loop invariant and the loop variant are satisfied before and after the execution of S. Finally, if the loop invariant I is satisfied for an arbitrary state y but the loop predicate evaluates to $false$, the postcondition Q is satisfied by y.

In this section we have presented the WP and WLP predicate transformers. In the next subsection we present how these can be used to verify programs automatically.

2.3.2 Automatic and Inductive Verification with Weakest Preconditions

With the predicate transformers as presented in the last subsection, it is possible to verify programs for partial or total correctness. For this purpose,

(1) verification conditions have to be calculated using WP (or WLP) and

(2) the verification conditions have to be verified by showing that the FOL formulas are valid.

Before we present these automatic techniques, we need to extend the *While* language by a number of statements required to annotate the program for verification:

$$\begin{aligned}
\text{S} ::= \;& \text{skip} \mid \text{x:=e} \mid \text{S;S} \mid \text{l:} \mid \text{goto l} \\
& \text{if (b) then } \{\text{S}\} \text{ else } \{\text{S}\} \mid \text{while (b) } \textbf{invariant} \text{ I } \{\text{S}\} \\
& \textbf{assume} \text{ b} \mid \textbf{assert} \text{ b} \mid \textbf{havoc} \text{ x} \\
\text{e} ::= \;& \text{e } op_a \text{ e} \mid \text{n} \mid \text{x} \mid \text{l} \\
\text{b} ::= \;& \text{true} \mid \text{false} \mid \text{b } op_b \text{ b} \mid \text{e } op_p \text{ e}
\end{aligned}$$

The statements of the language are defined as follows:

Definition 2.3 (Assert Statement)**.** *An assertion is a statement in the program that enables the programmer to provide a formal comment about the program states. Besides the keyword* `assert`*, it contains a predicate over variables of the program state. Assertions are used to specify preconditions and postconditions (e. g., for Hoare logic or predicate transformers). The WP and WLP for the assert statement are defined as*

$$wp(\textbf{assert}\ c, Q) = c \wedge Q \ \textit{and} \tag{2.3}$$
$$wlp(\textbf{assert}\ c, Q) = c \longrightarrow Q. \tag{2.4}$$

Definition 2.4 ((Loop) Invariant Statement)**.** *A loop invariant is an annotation to a loop that describes a predicate over the program states that holds on loop entry, after every execution of the body and after the loop finished. The* *invariant is specified between the loop head and the loop body using the keyword* ***invariant****.*

Definition 2.5 (Assume Statement)**.** *An assumption is a statement in a program that restricts the feasible executions of a subsequent program part to states that do no conflict with the predicate specified in the statement. The WP and WLP for the assume statement are defined as*

$$wp(\textbf{assume } c, Q) = c \longrightarrow Q \textit{ and} \tag{2.5}$$

$$wlp(\textbf{assume } c, Q) = c \longrightarrow Q. \tag{2.6}$$

where c is a predicate over the state of the program and Q is the postcondition. The basic idea of this definition is that if c holds after the `assume` *statement is executed, c also has to hold for the precondition.*

Furthermore, the `havoc` statement assigns an arbitrary value to a variable: $wp(havoc\ x, Q) = \forall z.Q[x \leftarrow z]$. This statement is often used in combination with the `assume` statement to first set a variable free with `havoc` and than to constrain the variable with an assumption. Finally, the language is extended by a `goto l` instruction that jumps to a label "`l:`". The goto statement is used to desugar[4] *if-then-else* and *while* statements in order to generate more compact VCs.

This extended *While* language is suitable to write programs and annotate preconditions, postconditions and loop invariants directly into the program. With WP and WLP such programs can be automatically translated into FOL formulas (VCs). By solving the VCs with automatic tools like theorem provers, the program can be verified automatically. Since the time needed for the tools grows with the size (and structure) of the FOL formulas, compact VCs are desired.

Compact and Efficient Weakest Preconditions

A key problem in generating and verifying verification conditions using WP or WLP is that due to the rule for the conditional instruction (Formula (2.1)) even small programs with control flow result in large formulas. Since for every branch the postcondition is copied, the verification condition increases exponentially in size compared to the actual size of the program.

Figure 2.6a depicts a small example program with two `if`-statements. The first calculates the absolute value for x, the second decrements a counter c. The assertion is the postcondition for this example program where it is required that the absolute value is not negative. Figure 2.6b also depicts the WP for the program. The postcondition is duplicated four times, even for the second `if`-statement even though it is not relevant to (since only c is assigned a new value, but there are not any occurrences of c in the postcondition).

[4]Remove syntactic sugar.

```
if( x ≤ 0) {
 x := -x
}
if(c > 0) {
 c := c-1
}
assert x ≥ 0;
```

(a) Example Program

$$
\begin{aligned}
& wp(if(x \leq 0)\{x := -x\}; if(c > 0)\{c := c-1\}, x \geq 0) \\
\Longleftrightarrow\ & wp(if(x \leq 0)x := -x, (c > 0) \longrightarrow (x \geq 0)) \wedge \\
& wp(if(x \leq 0)x := -x, (c \leq 0) \longrightarrow (x \geq 0)) \\
\Longleftrightarrow\ & ((x \leq 0) \longrightarrow (((c > 0) \longrightarrow (-x \geq 0)) \wedge \\
& ((c \leq 0) \longrightarrow (-x \geq 0)))) \wedge \\
& ((x > 0) \longrightarrow (((c > 0) \longrightarrow (x \geq 0)) \wedge \\
& ((c \leq 0) \longrightarrow (x \geq 0))))
\end{aligned}
$$

(b) Weakest Precondition for the Program

Figure 2.6: Example Program and its Weakest Precondition

Moreover, for every assignment like $x := e$ the expressions e is substituted in every occurrence of x, which may also lead to a large expression since the size of the expression may increase exponentially to the number of assignments (e. g., for a program $x_1 := x_2 + x_2; ...x_n = x_{n-1} + x_{n-1}; assert(x_n > 0);$).

Flanagan and Saxe presented an approach [FS01] to calculate more compact weakest preconditions for programs. Their approach is twofold: First the program is translated into a *passive form*, second, the WP is applied to the program in passive form to calculate the VC.

Definition 2.6 (Passive Form of a Program)**.** *A program is in a passive form, if every assignment $x := e$; is replaced by **assume** $x' = e$; where x' is a fresh variable.*

With a program in passive form, assignments no longer need to be substituted in the postconditions, which leads to a decrease in the size of the formulas. To translate the program in passive form into VCs, Flanagan and Saxe define two predicate transformers, N and W where N describes the states of the program terminating normally and W describes the states where the execution may go wrong. Then the following formula holds for programs in passive form

$$wp(S,Q) = \neg(W(S)) \wedge (N(S) \longrightarrow Q)$$

and the resulting verification condition is quadratic in size to the program since the postcondition Q is not longer duplicated but the VC now contains two formulas per statement. In [Lei05], Leino presented the proof that for programs in passive form holds that

$$\neg W(S) = wp(S, true) \text{ and}$$
$$\neg N(S) = wlp(S, false).$$

This enables the use of the original predicate transformers.

Barnett and Leino presented a technique [BLS05] based on programs in passive form that is able to generate even more compact VCs.

Unstructured Control Flow and Loop Cutting

In their approach [BLS05], Barnett and Leino assume that a (possibly unstructured) program consists of a number of blocks starting with a label followed by a number of statements and ending with a goto-statement:

$$L:\quad \texttt{S; goto } M \tag{2.7}$$

Moreover, the statement `if (C) {A} else {B}` is desugared into:

```
if:   skip; goto then, else;
then: assume C; A; goto end;
else: assume ¬C; B; goto end;
end:  ...
```
(2.8)

To transform a program into the passive form, they calculate the *dynamic assignment form* [Fea91] first. Furthermore, fresh variables are introduced at the end of every branch before a join point (e. g., the `if`- and the `else`-block assign the calculated value for a variable x both to a variable x_{end} which is then used for further calculations).

The key idea of their approach is the introduction of an *auxiliary variable* L_{OK} for each block, which is defined as

$$L_{OK} \equiv wp(S, \bigwedge_{M \in Succ(L)} M_{OK})$$

where $Succ(L)$ is the set of all successor blocks. If no successors exist, then L_{OK} is defined as $L_{OK} = wp(S, true)$. This is called the *block equation* for a block. For example, the block equation for the `else` block from the desugared `if`-instruction (Formula (2.8)) is

$$else_{OK} \equiv (\neg C \longrightarrow wp(B, end_{OK})).$$

The verification condition for a program P then is

$$\left(\bigwedge_{B \in BlockEq(P)} B \right) \longrightarrow Start_{OK} \tag{2.9}$$

where $BlockEq(P)$ is the set of all block equations of a program and $Start_{OK}$ is the auxiliary variable for the first block.

However, this technique is limited to *loop-free passive* programs. Hence, the Control Flow Graph (CFG) is transformed into a loop-free from using *loop cutting*. A loop `while (C) invariant I {A}` is desugared into

```
start:   ...  goto head;
head:   assert I; goto body, end;
body:   assume C; A; goto end, head;                (2.10)
end:    assume ¬C; ...
```

where I is the loop invariant. This representation is still cyclic. In a subsequent transformation step, the back edges are removed. Furthermore, the invariant

(1) is asserted before the loop head is entered and

(2) for an arbitrary assignment to the loop variables assumed to hold for the loop head and

(3) checked at the end of the body.

The loop is transformed in the acyclic representation

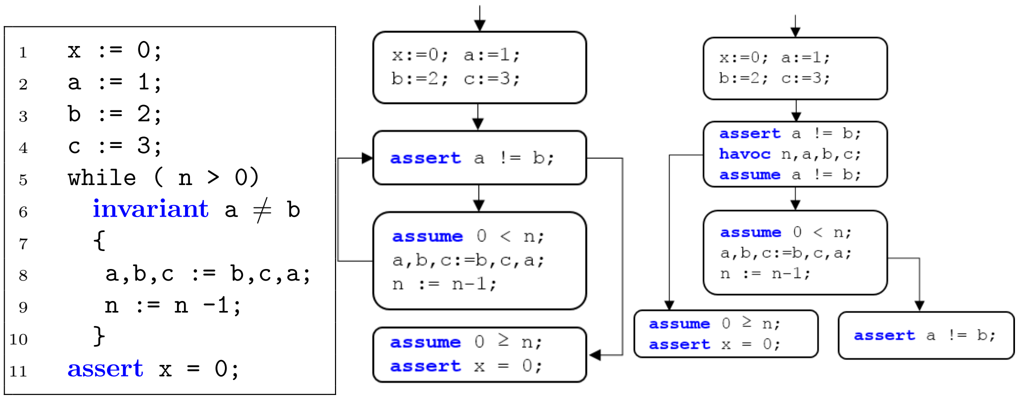
```
start:  ...; assert I; goto head;
head:   havoc V_loop; assume I; goto body, end;
body:   assume C; A; assert I; goto ;                  (2.11)
end:    assume ¬C; ...
```

where V_{loop} are the *loop-bound* variables modified in the body. This representation corresponds to the rule of iteration (see Formula (2.2)). The acyclic form of the loop can now be verified using the aforementioned technique for the generation of verification conditions.

Since programs with structured control are a special case of programs with unstructured control flow, this technique can also be used for the verification of structured programs. Barnett and Leino also point out that the resulting verification conditions are linear in size to programs in passive form.

Example Figure 2.7a depicts a small program in our while language that switches three variables in a loop. For simplicity we assume the instruction "`a,b,c := b,c,a;`" to do the assignment of the variables simultaneously. Figure 2.7b shows the CFG of the program after desugaring has been applied. Every vertex represents a block and every edge represents a `goto`-statement at the end of the block. Finally, Figure 2.7c shows the CFG after all back edges have been cut.

In this subsection we have introduced an algorithm that is able to produce compact verification conditions. This is actually the algorithm used by Boogie, the verification tool we use in our framework. In the next section, we present a verification strategy that increases the degree of automation even further.



```
x := 0;
a := 1;
b := 2;
c := 3;
while ( n > 0)
  invariant a ≠ b
  {
   a,b,c := b,c,a;
   n := n -1;
  }
assert x = 0;
```

x:=0; a:=1; b:=2; c:=3;
assert a != b;
assume 0 < n; a,b,c:=b,c,a; n := n-1;
assume 0 ≥ n; assert x = 0;
x:=0; a:=1; b:=2; c:=3;
assert a != b; havoc n,a,b,c; assume a != b;
assume 0 < n; a,b,c:=b,c,a; n := n-1;
assume 0 ≥ n; assert x = 0;
assert a != b;

(a) Example Program **(b)** Desugared Loop **(c)** Loop Cutting

Figure 2.7: Example Program and CFG with Loop-Cutting

k-Inductive Invariant Verification

k-Inductive invariant verification, which is often referred to as *k-induction*, is a technique originally proposed by Sheeran et al. [SSS00] as a verification technique for finite state transition systems. This technique proposes an inductive proof, where the *base case* as well as the *induction step* each are performed over k transition steps in the system.

Let $I(s)$ and $T(s,s')$ be formulas where $I(s)$ describes the initial states of a system and $T(s,s')$ describes the transition relation from a state s to a state s'. Furthermore, let $P(s)$ be a formula that describes that a property P holds for state s. To prove that the property p holds for the base case, the following formula has to be unsatisfiable:

$$I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_{k-1},s_k) \wedge \neg P(s_1) \wedge \cdots \wedge \neg P(s_k) \tag{2.12}$$

In other words, the property p is not violated in k transition steps of a finite state system. For the induction step, it is necessary to show that a second formula is unsatisfiable where $s_n \ldots s_{n+k+1}$ are consecutive states:

$$P(s_n) \wedge T(s_n,s_{n+1}) \wedge \cdots \wedge P(s_{n+k}) \wedge T(s_{n+k},s_{n+k+1}) \wedge \neg P(s_{n+k+1})$$

In other words, the property p holds for k transition steps and is not violated after $k+1$ steps. From these two formulas one can conclude that the property holds in all states.

This technique can also be applied in program verification. Donaldsen et al. introduced k-induction for WP in [DHKR11]. The key idea of this technique is to treat a loop as a transition relation $L(s,s')$ on the states of the program. Here, s is the program state before the execution of the loop body and s' is the program state afterward. L is constructed using a predicate transformer like WP and the loop invariant I is the property that needs to be satisfied by the program states.

For the program

S_{init}`; while (C) invariant I {` S_{body}`};assert P;`,

we can show by unrolling the loop k times that the base case holds:

```
start:  $S_{init}$; goto $b_1$;
$b_1$:     assume C; assert I; $S_{body}$; goto $b_2$, end;
$b_2$:     assume C; assert I; $S_{body}$; goto $b_2$, end;
...
$b_k$:     assume C; assert I; $S_{body}$; goto end;
end:    assume ¬C; assert P;
```

The invariant is not violated for k consecutive steps starting from an initial state. Furthermore, the post condition P holds if the loop is left after every step. To show the induction step, the loop is unrolled $k+1$ times

```
start:  havoc $V_{loop}$; goto $b_1$;
$b_1$:     assume C; assume I; $S_{body}$; goto $b_2$;
...
$b_k$:     assume C; assume I; $S_{body}$; goto $b_{k+1}$;
$b_{k+1}$:   assume C; assert I; $S_{body}$; goto end;
end:    assume ¬C; assert P;
```

where the loop-bound variables are set to an arbitrary state and the invariant is assumed to hold for the first k steps and asserted for the final $(k+1)$ step.

If the resulting VC for the base case is unsatisfiable, then there exists an error in the program (or specification). If the VC for the induction step is not satisfiable, then there either may be an error in the program or the k is not sufficiently chosen to prove the properties.

In their approach, Donaldsen et al. also present a way to encode the base case and the induction step into one VC, which preserves the information over the rest of the program states and hence enables the use of weaker invariants.

Example The loop invariant $a \neq b$ for Figure 2.7a is too weak to verify the loop with one-induction since it does not contain any information about c. If we perform k-induction with $k = 2$ and assume that the invariant is satisfied for two loop iterations, we can derive that $b \neq c$. If we perform k-induction with $k = 3$, we can derive that $b \neq c \wedge c \neq a$. Since from $b \neq c \wedge c \neq a$ follows that $a \neq b$, k-induction with $k = 3$ is sufficient to verify the loop.

Once the VCs are calculated with WP, WLP and k-induction, they have to be solved to verify the programs. In our approach we use automatic theorem provers to solve the formulas.

2.3.3 Verification with Automatic Theorem Proving

In the previous section, we have introduced efficient techniques to automatically generate FOL formulas from programs and their specification. To verify a program $\{P\}S\{Q\}$, we have to show that the VC

$$P \longrightarrow wp(S,Q)$$

is valid. For automatic verification, this yields two questions. First, how to show that a verification condition is valid and, second, how to minimize the manual effort needed to derive a specification? The first problem can be solved by using automatic theorem proving. However, the second problem requires a trade-off between the degree of automation and the completeness of the verification: *extended static checking.* To describe the automatic theorem proving techniques we use, we first need to introduce theories and decision procedures for FOL.

Theories and Decision Procedures

To perform the verification process automatically, techniques are required that enable the solving of FOL formulas without human interaction. The *decision problem*, i. e., deciding whether a formula is valid, can be answered automatically using *decision procedures* [KS08]. However, decision procedures are only available for first-order-*theories.* These theories are defined for a subset of FOL: *nonlogical symbols* like functions, predicates and function symbols.

Definition 2.7 (Theory)**.** *A first-order theory is defined by*

- *a set of nonlogical symbols, the signature* Σ
- *a set of axioms, which are formulas where every variable is bound to a quantifier and only symbols of* Σ *and logical connectives appear.*

A Σ-formula ϕ that only consists of variables, quantifiers, logical connectives and symbols of Σ is valid for a theory T if every possible assignment to variables of ϕ satisfies all axioms of T and ϕ.

Definition 2.8 (Decision Procedure)**.** *A decision procedure for a theory* T *is a procedure that solves the decision problem for every formula* ϕ *of* T *and,*

- *if it returns "valid" then* ϕ *is valid (*soundness*) and,*
- *for every valid* ϕ*, it returns "valid" and terminates (*completeness*).*

A theory T is *decidable*, iff there exists a decision procedure for it.

Since FOL is undecidable in general, in practice also incomplete procedures or fragments of theories are of interest. A fragment is a syntactically restricted subset of formulas to a theory, e. g., all quantifier-free formulas. Incomplete (decision) procedures may not always terminate but are able to cope with undecidable theories or fragments. In practice, incomplete procedures are also used to solve decidable theories faster. For a detailed introduction to decision procedures, we refer to [KS08].

Important theories are, e. g., *propositional logic*, *equality linear arithmetic*, *bit vectors*, *arrays*, *non-linear arithmetic* and their fragments, e. g., *quantifier free bit vectors* or *linear integer arithmetic.* In practice, often a combination of multiple theories is necessary to solve a VC. For example, to solve the equation from Figure 2.6b a combination of the theories of *linear arithmetic* and *propositional logic* is required. A technique to solve such a VC is to use a solver for *Satisfiability Modulo Theories.*

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the (research) field concerned with the satisfiability of FOL formulas with respect to some background theories. The key idea in SMT is to map a FOL problem with respect to the background theories into a problem of satisfiability for propositional logic, also known as SAT. For SAT, there exist very efficient decision procedures (e. g., the widely used Davis-Putnam-Logemann-Loveland (DPLL) framework [DLL62]).

There are two major approaches used for SMT-solvers: The *eager* and the *lazy* approach. While eager approaches directly encode the FOL formula into a propositional formula to invoke a decision procedure for SAT (and to benefit of the efficiency of modern SAT-solvers), lazy approaches incrementally construct the propositional formula by adding clauses derived using specialized theory solvers and repeatedly checking for satisfiability. For a detailed introduction to SMT, we refer to [BSST09].

Nowadays, many SMT-solvers support various theories and fragments (up to 22) [BDdM$^+$13] and are used in multiple application areas, e. g., verification or *extended static checking.* Even though many of the theories are not decidable, solvers are often capable of finding a solution.

Checking Verification Conditions with SMT

Since SMT-solvers are more suited to find a satisfying assignment than showing that a formula is satisfiable for all possible assignments, the VC has to be adjusted. Instead of proving that the VC is valid, the strategy is to show that the formula is unsatisfiable if the property is violated. To this end, the properties of interest (usually the assertions) are negated in the formula like in (2.12). Moreover, if the SMT-solver is able to find a satisfiable assignment for the VC with negated assertions, this assignment is actually a *counterexample* describing an error state of the program. This counterexample can be used to determine the source of the error. A technique, where SMT is used in that way to find certain errors ins programs, is Extended Static Checking.

Extended Static Checking

Extended Static Checking (ESC) is a technique introduced by Detlefs et al. for Modular-3 (ESC/Modular-3) [Det96, DLNS98] and has later been adopted and improved for Java (ESC/Java) by Flanagan et al. [FLL+02]. The aim of ESC is twofold: Provide a static checker that gives mathematically sound guarantees for certain properties, more precisely the absence of specific errors, but needs only little manual effort in specification (and in the actual proving of the properties) compared to formal verification. The key idea of ESC is to obtain the mathematical guarantees by translating the program into VC using predicate transformers and to use automatic theorem provers to get a counterexample or to verify the absence of errors. ESC focuses on the error classes *division-by-zero*, dereferencing of null pointers, invalid casts or race conditions. Thus, ESC is only sound with respect to the error classes, but not to the correctness of the program since if ESC does not report an error, the program is not necessarily correct. However, in practice ESC has been successfully used to verify real world applications [LS09].

The techniques for the generation of efficient WP presented in the previous sections originally were introduced in the ESC-context. Furthermore, the Boogie verification framework used in our approach has been developed as an intermediate framework for a number of ESC-tools. We give a detailed introduction to Boogie in Section 2.4.

2.3.4 Summary

In this section, we have introduced the weakest precondition predicate transformer, which is used for the automatic verification of programs. We have presented techniques for the generation of efficient and compact verification conditions in FOL. Furthermore, we have presented inductive techniques for the verification of loops with loop invariants. We have presented k-inductive invariant verification (k-induction), which provides a higher degree of automation since it allows for the use of weaker loop invariants. Finally, we have given a brief introduction to automatic theorem proving, more precisely SMT-solving, and how it can be used to verify programs. In the next section, we give an introduction in the Boogie verification framework.

2.4 The Boogie Verification Framework

Boogie[BCD+06] is a verification framework developed at Microsoft Research. It was originally used for ESC as presented by Detlefs et al. in [Det96, DLNS98]. Boogie is used as intermediate framework for the verification of various programming languages, such as *VCC*[CDH+09] and *Havoc*[BHL+10] for *C*, *Spec#*[BLS05] for *C#*.

The Boogie verification framework consists of the intermediate verification language *Boogie2* [Lei08] (formerly BoogiePL) and the Boogie tool[5], which is used to verify programs written in Boogie2. The Boogie tool uses an automatic theorem prover (usually a SMT-solver) to verify programs. In this section, we briefly introduce the Boogie2 language and the Boogie verification tool.

2.4.1 Boogie2

In this section, we briefly introduce the Boogie2 language following Leino in [Lei08]. We will introduce the most important constructs relevant to our translation. For a detailed description of all language features (e. g., the full grammar) we refer to [Lei08].

Boogie2 is an imperative, typed and procedural language. It provides a number of primitive types and some basic *arithmetical*, *relational* and *propositional* operations for these types. Furthermore, it enables the use of *symbolic types*, *maps*, *constants* and *uninterpreted functions*. Constants and uninterpreted functions can be constrained using *axioms*.

The Boogie2 language provides a general intermediate verification language which is independent from the tooling. Hence, nearly every language element may be annotated by attributes that are specific to the desired verification tool to use additional tool-specific features. For reasons of simplicity, we omit these attributes in the subsequent grammar descriptions. Furthermore, an apostrophe in a grammar expression like term'^{*} means that if the term occurs more than one time, it is separated by a colon.

A Boogie2 program has the following syntax:

$$\begin{array}{lcl} \mathit{Program} & ::= & \mathit{Decl}^* \\ \mathit{Decl} & ::= & \mathit{TypeDecl} | \mathit{ConstantDecl} | \mathit{FunctionDecl} | \mathit{AxiomDecl} \\ & & | \mathit{VarDecl} | \mathit{ProcedureDecl} | \mathit{ImplementationDecl} \end{array}$$

Types

Types in Boogie2 are defined using the following grammar:

$$\mathit{TypeDecl} \quad ::= \quad \textbf{type finite}?\ \mathit{Id}\ \mathit{Id}^*$$

The keyword *finite* indicates that the type has a finite number of individuals. Furthermore, possibly polymorphic map types are defined with

$$\mathit{MapType} \quad ::= \quad [\mathit{Type}_x'^{+}]\mathit{Type}_y$$

where $\mathit{Type}_{\{x,y\}}$ is primitive or an already declared type. However, even though user defined types and maps are useful, e. g., to model a heap, we do not

[5]http://boogie.codeplex.com

need these currently in our formal specification for Matlab/Simulink models. Hence, we focus on the primitive types in Boogie2.

Table 2.1 depicts the available primitive types in Boogie2 together with the available operations. The "<:" operator is a relational operator for the

Table 2.1: Primitive Types in Boogie2

bool	Boolean values	$\wedge, \vee, \neg, \Longrightarrow, \Longleftrightarrow, =, \neq$
int	Mathematical Integers	$+, -, *, /, <, \leq, >, \geq, =, \neq, <:$
real	Rational Numbers	$+, -, *, /, <, \leq, >, \geq, =, \neq, <:$
m**bv**n	Bit-vectors (of size n with value m)	$++, [n:m]$

partial ordering. The $[n:m]$ operator is the extract operator on bit-vectors where a range of bits are extracted from n to m resulting in a bit-vector of the size $m-n$. However, Boogie2 does not support conversion between *int* and *real* types and no further bit vector operations. Nevertheless, it is possible to enhance the built-in arithmetic using (uninterpreted) functions.

Variables

The syntax for variable declarations in Boogie2 is defined as

$$\begin{aligned} VarDecl &::= \textbf{var}\ IdsTypeWhere'^{+}; \\ IdsTypeWhere &::= Id : Type^{+}?\ WhereClause?; \end{aligned}$$

where the *WhereClause* specifies a predicate that has to be fulfilled when the variable is assigned to an arbitrary value, e. g., using the *havoc* (see Section 2.3.2) command. Note that the *WhereClause* may be violated due to assignments. An assignment to a variable is done using the assignment operator ":=". Variables declared globally outside of procedures are visible for all scopes. Variables declared in procedures are only visible within the procedure body.

Functions, Constants and Axioms

Beside mutable variables, Boogie2 also provides constants. Constants are declared using the following grammar:

$$ConstantDecl \ ::= \ \textbf{const unique}?\ Id'^{+} : Type\ OrderSpec?$$

The keyword *unique* indicates that the value of the constant is unique with respect to other constants of the same type in the program. The *OrderSpec* is used to define a partial order for the type of the constant. We refer to [Lei08] for a detailed description.

(Uninterpeted) functions are declared using the syntax:

$$\begin{array}{lcl} \mathit{FunctionDecl} & ::= & \textbf{function}\ \mathit{Attribute}?\ \mathit{Id}\ \mathit{FSig}; \\ & & \mid \textbf{function}\ \mathit{Attribute}?\ \mathit{Id}\ \mathit{FSig}\{\mathit{Expr}\}; \\ \mathit{FSig} & ::= & (\mathit{IdType}'^{*})\ \textbf{returns}\ (\mathit{IdType}) \\ \mathit{IdType} & ::= & \mathit{Id}\ {:}?\ \mathit{Type} \end{array}$$

A function always has a return type and it may have a list of arguments. While it is sufficient to only define the types of the argument, it is also possible to use additional variable identifiers. These identifiers are necessary if the function is defined with an additional expression that describes how the return value is calculated. However it is also possible to describe the behavior of a function using axioms.

In our translation, we make use of two attributes for functions. The first is the {*:bvbuiltin* ***fun_name***} attribute that maps the Boogie2 function to the *fun_name*-function of the underlying theorem prover. This is useful to enhance the arithmetics of Boogie2. The second attribute is the {*:inline*} attribute that causes a translation of the function into a macro instead of a function. Macros may lead to a performance gain for the verification.

Besides of functions, axioms are also used to set a constant to a specific value. The syntax for axioms is defined as

$$\mathit{AxiomDecl} \ ::= \ \textbf{axiom}\ \mathit{Expr};$$

where *Expr* has to evaluate to a Boolean value. Note that inconsistent axioms can lead to incorrect verification results.

Assumptions, Assertions and Quantifiers

Boogie2 is an intermediate verification language. Hence, it provides commands to constrain the possible execution paths (*assumptions*) and to prove properties (*assertions*). The syntax for these statements is defined as follows:

$$\begin{array}{lcl} \mathit{Stmt} & ::= & \textbf{assert}\ \mathit{Expr}; \\ & & \mid \textbf{assume}\ \mathit{Expr}; \\ & & \dots \end{array}$$

Note that *Expr* has to evaluate to a Boolean value. The semantics of these two statements corresponds to the definitions in Section 2.3.1.

Assertions, assumptions and axioms may also contain quantifiers. The syntax of quantified expressions is defined as

$$\begin{array}{lcl} \mathit{QExpr} & ::= & \textbf{forall}\ \mathit{IdsType}^{+}\ ::\ \mathit{TrigAttr}\ \mathit{Expr} \\ & & \mid \textbf{exists}\ \mathit{IdsType}^{+}\ ::\ \mathit{TrigAttr}\ \mathit{Expr} \end{array}$$

where the quantifier is applied to a number of variables or constants of certain types. Note that it is possible to define a number of *triggers* for a quantifier. A trigger is an expression specifying a pattern, when the quantifier shall be instantiated by the underlying theorem prover. Hence, triggers may be crucial for the performance of the verification.

Procedures

The syntax for a procedure is defined as

$$\begin{array}{lll} \mathit{ProcedureDecl} & ::= & \textbf{procedure}\ \mathit{Id}\ \mathit{PSig}\ ;\ \mathit{Spec}^*; \\ & & \mid \textbf{procedure}\ \mathit{Id}\ \mathit{PSig}\ \mathit{Spec}^*\ \mathit{Body} \\ \mathit{PSig} & ::= & (\mathit{IdsTypeWhere}^*)\ \mathit{OutParameters}? \\ \mathit{OutParameters} & ::= & \textbf{returns}(\mathit{IdsTypeWhere}) \end{array}$$

where the first line only declares the interface of procedure while with the second line, we can also specify the implementation body. Besides an identifier, an optional set of inputs parameters and an optional output value, a procedure declaration may also contain a specification (*Spec*). A specification consists of zero or more preconditions (*requires*), postconditions (*ensures*) or frame conditions (*modifies*). Since these are basically syntactic sugar for assumptions and assertions, they also require a (possibly quantified) expression that evaluates to a Boolean value.

A procedure may also have none or more than one implementation. The signature of the implementation must repeat the declaration except for the specification, which is only attached to the declaration. An implementation for a procedure is defined using the keyword *implementation* in a similar syntax like the declaration. However, the implementation body can also be directly added to the declaration.

The body of a procedure consists of a list of local variable declarations and a list of statements. These statements are either assignments, assertions, assumptions, goto statements or labels. Furthermore, there are statements for procedure calls and control structures, which are only syntactic sugar.

Blocks and Loops

Besides *if-else* statements and while loops, Boogie2 also enables the modeling of programs with unstructured control flow using *goto*-and *break*-commands. Loops and control flow structures are desugared into *goto*-blocks (see Section 2.3.2).

Example Listing 2.1 depicts a small example program in Boogie2. It consists of one procedure *foo* that takes one parameter x, compares x against a constant c and returns either 42 if x equals the constant c or returns 24 otherwise.

```
function { :bvbuiltin "to_int" }
   real_to_int(x1: real) returns (int);
const c: real;
axiom c == 42.0;
procedure foo (x: int) returns (int)
 ensures bar >= 0;
{
  var bar: int;
block1:
  goto if, then;
if:
  assume to_int(c) == x;
  bar := 42;
  goto end:
then:
  assume !(to_int(c) == x);
  bar := 24;
  goto end:
end:
  return bar;
}
```

Listing 2.1: A Boogie2 Example

In this example, we use the {*:bvbuiltin*} parameter to connect the Boogie2 function *real_to_int* with the *to_int*-function of the theorem prover. The constant c of type real is set to 42.0 using an axiom. The procedure *foo* has a specification attached to the declaration consisting of a postcondition that the return value is always positive. The body of the procedure is directly attached to the declaration. It consists of a declaration of the local variable *bar* and four *goto*-blocks. Moreover, these blocks represent a desugared if statement (see Section 2.3.2, Formula (2.7)).

2.4.2 The Boogie Verifier

While there exist interfaces to other SMT-solvers, the Boogie verification framework usually uses the automatic theorem prover *Z3* [dMB08] developed at Microsoft Research[6]. To verify a Boogie2 program, it is translated into FOL formulas and passed to the SMT-solver. Instead of proving that a formula ϕ is valid for all possible variable assignments, the Boogie tool shows that $\neg\phi$ is not satisfiable. If there exists a satisfying assignment for $\neg\phi$, this variable assignment corresponds to a counterexample.

To verify a Boogie2 program, the Boogie tool performs a number of analysis, rewriting and transformation steps prior to the actual generation of the FOL formulas. The most important steps are:

- **Desugaring & Loop Detection** One of the first processing steps is a rewriting of the Boogie2 specification into a loop free, desugared form. Since the *goto*-Statement may introduce cycles and unstructured control

[6]http://z3.codeplex.com

flow, a loop detection is performed on the CFG and, if necessary the CFG is transformed into a reducible CFG. Furthermore, *if-then-else* statements, *while* loops and procedure calls are desugared into *goto*-blocks, assertions and assumptions (see Section 2.3.2, Formula (2.7) and Formula (2.8)).

- **Abstract Interpretation** If the flag is not disabled, abstract interpretation [CC77] is performed to derive invariants for loops for ranges of variables. However, the domains supported by abstract interpretation in the current version of the Boogie tool is limited to intervals. Moreover, constraints given by assertions are not considered by the analysis.
- **Passive Form** Once the acyclic and reducible CFG is calculated, the program is translated into its passive form (see Section 2.3.2).
- **Verification Condition Generation** Finally, the program is translated into a FOL verification condition using the weakest precondition predicate transformer and the technique by Barnett and Leino described in Section 2.3.2. The resulting VC corresponds to Formula (2.9). Boogie encodes the formulas using the *SMTLIB 2.0* format [BST10]. However, it also uses some of *Z3*-specific commands such as *labels* that are used to map the parts of the formula to their corresponding assertion and *goto*-blocks. With these labels, a counterexample (satisfying assignment) can be easily associated to a specific part of the Boogie2 program.
- **Counterexample Generation** The Boogie verification tool is designed to find all possible counterexamples up to an user specified maximal number of errors to be reported. Hence, every time a violated assertion is found, (1) the counterexample is reported, (2) it is assumed to be not violated, and (3) the verification is continued until all counterexamples are found ($\neg\phi$ is unsatisfiable) or the user defined bound is reached. Then the counterexamples found are reported to the user.

2.4.3 Summary

In this section, we have presented the Boogie verification framework. First, we have introduced the core elements of the intermediate verification language Boogie2. Subsequently, we have given a brief overview to the core mechanics of the Boogie verification tool and how this mechanics are related to the techniques presented in Section 2.3.2. However, we give a more thorough description of some of the language elements and the Boogie tool in Chapter 6 and Chapter 8.

2.5 Slicing

Program slicing is a technique for extracting those parts from a program that either affect or are affected by a point of interest in the program. A point of interest in a program is usually a line number (or a statement) and a set of variables.

The extracted statements, which are a subset of all statements in the program, are called a *slice*. The point in the program and the variables of interest are usually referred to as a *slicing criterion*. A detailed overview about program slicing is given by Tip in [Tip95] and Silva in [Sil12]. In general, slices can be classified with respect to the following properties:

- **Forward vs. Backward** *Backward slices* contain the statements that influence the program point given by the *slicing criterion*. *Forward slices* contain the statements that are influenced by the *slicing criterion* in the further execution of the program.
- **Static vs. Dynamic** A slice can either contain all possible execution paths from or to the slicing criterion, which is called a *static* slice, or only contain those execution paths that are possible for a given set of inputs to the program, which is called a *dynamic* slice.
- **Executable vs. Non-Executable** A slice can either be an *executable* subset of the program or a *non-executable* subset. Depending on the purpose of the slicing approach, non-executable slices can also be useful, e. g., program comprehension.
- **Syntax Preservation** A slice can preserve the syntax of the program, e. g., preserve control flow structures, or do not preserve the syntax, which not necessarily renders the slice un-executable.
- **Intraprocedural vs. Interprocedural** Slices can be either calculated within a procedure (*intraprocedural*) or also with respect to calls to other procedures (*interprocedural*).

Slicing is used for various purposes ([Tip95, Sil12]). While originally developed for debugging, it was also used for program comprehension, software maintenance, for testing or reverse engineering. Forward slicing is also used to analyze the impact of changes in programs. Finally, static, executable and syntax preserving slices can also be used as automatic reduction techniques prior to static analysis since it captures all possible execution paths, and are hence a sound over-approximation for the subset of statements under consideration. In the next section, we introduce a widely used technique for the calculation of program slices, which is based on *Program Dependence Graphs*.

2.5.1 The Program Dependence Graph

The original slicing approach[Wei81] was based on solving data flow equations. However, most slicing algorithms are using Program Dependence Graphs (PDGs) that represent the program and calculate the slices with graph traversal techniques. PDGs were first introduced by Ferrante et al. [FOW84] as follows:

Definition 2.9 (Program Dependence Graph (PDG))**.** *A PDG is a directed, rooted graph $G(V,A)$ where*

(i) the set V that contains nodes representing the statements (from the CFG) of a program and region nodes, and

(ii) the set A of ordered pairs of $V \times V$ that represent the dependence edges.

A special region node is the entry *node that cannot be the target of any dependence edge.*

Dependence Analysis

To build the dependence graph for a program, a *dependence analysis* has to be performed first. In the original approach, the set of dependence relations consisted of two types of dependences: *data* and *control* dependence. Both are defined in terms of the control flow graph of a program. Later, further dependence relations have been added, e. g., to be able to calculate interprocedural slices.

Data dependence (also known as *flow* dependence or *def-use-chains*) is defined as:

Definition 2.10 (Data Dependence)**.** *A node j is data dependent on a node i if there exists a variable x with*

(i) x is defined at i

(ii) x is referenced at j

(iii) there exists a path from i to j where x is not redefined at any node in that path.

Control dependence between nodes in the CFG is usually defined in terms of post-dominance. A node i is post-dominated by a node j if all paths from i to the ***exit*** node pass through j.

Definition 2.11 (Control Dependence)**.** *A node j is control dependent on a node i if*

(i) there exists a path from i to j such that j post-dominates every node in the path excluding i and j, and

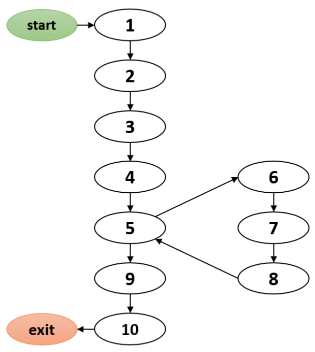
(ii) i is not post-dominated by j.

In other words, a node j is control dependent on a node i if i has at least two outgoing edges and j is not in all of the paths to the ***exit*** node starting from these edges. Control dependence can be determined by calculating the post-dominator tree for a CFG.

Example In Figure 2.8, we depict a small example program to demonstrate slicing and the corresponding CFG. The program (Figure 2.8a) calculates

```
1  read(n);
2  i = 1;
3  sum = 0;
4  mul = 1;
5  while( i ≤ n)
6    sum = sum + i;
7    mul = mul * i;
8    i = i + 1;
9  write(sum);
10 write(mul);
```

(a) A Example Program



(b) CFG

Figure 2.8: A Example Program and its CFG

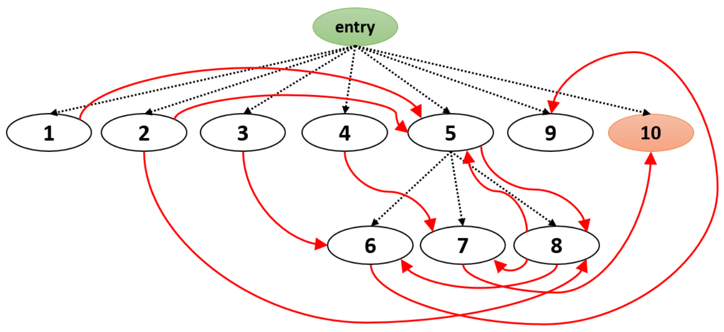
the sum as well as the product of the first n positive integers. In the CFG (Figure 2.8b), the node 5 post-dominates node 3. Since in the path from 3 to 5 every node including 3 is post-dominated by 5, 5 is not control dependent on 3. In contrast, node 7 post-dominates all nodes on the path to node 5 except for node 5. Hence, node 7 is control dependent on node 5 ($5 \rightarrow_{cd} 7$). In Figure 2.8b node 5 is data dependent on node 2 ($2 \rightarrow_{dd} 5$) since node 5 reads the value of the variable i, which is defined in node 2.

With the dependences calculated for every statement in th program, we are now able to construct the dependence graph. In the next section we show, how slicing is performed using dependence graphs.

2.5.2 Static Slicing with Program Dependence Graphs

Once the dependence analysis is finished, the dependence graph can be constructed using the calculated dependences. This is done using the nodes from the CFG and adding the data and control dependence edges. Additionally, every node that is not control dependent on a node from the CFG is set to be control dependent on the *entry* node.

Ottenstein and Ottenstein [OO84] presented an approach to perform slicing based on PDGs. With PDGs, slicing can be mapped to a reachability problem on the graph. In this approach, the slicing criterion is not any more a line in the program and a set of variables of interest. Instead, the slicing criterion is a node in the graph. This means, all variables defined or used by the statement the node is representing, are the variables of interest. To calculate the slice, a reachability analysis has to be performed either following the edges in backward direction to calculate the backward slice or in forward direction to calculate the forward slice. The slice consists of all nodes that are reachable starting from the slicing criterion.



(a) PDG for Fig. 2.8a

```
1   read(n);
2   i = 1;
3
4   mul = 1;
5   while( i ≤ n)
6
7     mul = mul * i;
8     i = i + 1;
9
10  write(mul);
```

(b) Slice for Line 10

Figure 2.9: PDG and Slice for Line 10 for Fig. 2.8

The slices obtained by [OO84] are static syntax preserving intraprocedural forward and backward slices. Later Horwitz et al. extended this approach [HRB90] to also be able to calculate interprocedural slices by enhancing the PDG by additional region nodes and dependence edges for procedure calls, and by adjusting and optimizing the dependence analysis and graph traversal algorithms. In the last decades, further slicing algorithms have been presented for various programming languages and concepts, e. g., functional languages, object oriented languages, concurrent programs and even modeling notations. For more information on these approaches, we refer to [Tip95, Sil12].

Example In Figure 2.9, we depict the PDG and the backward slice for the statement in Line 10 of the example program. In the PDG (Figure 2.9a), control dependence edges are drawn with a black dotted line. Data dependence edges are drawn with a solid red line. The slicing criterion (node 10) is marked in light-red color. The backward slice is calculated by following the edges backward starting at node 10 until every node that is reachable is added to the slice. Starting from node 10 the nodes 3, 6 and 9 are not reachable using a backward traversal of the graph. Hence, they do not belong to the slice and the corresponding statements of the program are removed. The resulting static backward slice is depicted in Figure 2.9b.

In this section, we have briefly introduced program slicing. Furthermore, we have presented the program dependence graph and its use in static program slicing.

2.6 Summary

In this chapter, we have introduced the relevant background for our verification approach. To this end, we have given an introduction into the general paradigm of Model Driven Development (MDD) and its realization with the Matlab/Simulink tool suite and the Simulink modeling notation. For the latter, we have presented the core modeling and simulation concepts. Then, we have introduced program verification, in particular automatic techniques like the weakest precondition predicate transformer, inductive invariant verification and k-induction, and automatic theorem proving. Subsequently, we have given an introduction into the Boogie program verification framework and its formal intermediate verification language Boogie2. Finally, we have introduced static program slicing based on dependence graphs.

3 Related Work

In this thesis, we present a framework for the formal verification of discrete-time Simulink models. It is based on a transformation of the models into an intermediate verification language to define a formal semantics for the informally defined Simulink models. Furthermore, we propose the use of slicing to reduce the complexity of the verification problem and present a slicing approach for Simulink models. The following discussion of related work is structured along these core aspects of our approach. First, we start with a discussion of verification approaches that have been published for Matlab/Simulink. Then, we discuss related work for the slicing technique presented in this thesis.

3.1 Verification of Simulink Models

In the last decade, a number of verification approaches has been published for Matlab/Simulink. A major challenge for all verification approaches is that the semantics of Matlab/Simulink models is only specified informally in the documentation of the Simulink tool. To obtain a formal semantics for Simulink models, most approaches utilize a transformation of the Matlab/Simulink models into a formally well defined representation. To this end, the models are either directly translated into the input language of a verification tool or into programming languages or other modeling notations with mature verification backends. In the following, we differentiate these approaches whether they aim for the verification of hybrid models that contain both, discrete and continuous time model elements or for purely discrete models.

3.1.1 Verification of Hybrid Simulink Models

Silva and Krogh presented an approach [SK00] for the verification of Simulink/Stateflow diagrams that is based on a transformation into hybrid automata. Originally, the approach was limited to the class of *threshold-event-driven hybrid systems* and required a specific way of modeling. Later on, Agrawal et al. [ASK04] presented a graph transformation to translate a subset of Simulink/Stateflow models into hybrid automata. This translation enables the use of verification tools for hybrid automata such as CheckMate [SK00], HyTech [HHWT97] or SpaceEx [FLGD$^+$11]. However, while this technique is well suited to show that a modeled system is correct w. r. t. its desired functionality (trajectory), it is not suited to detect possible run-time errors in the model like our error classes of interest. Tiwari presented in [Tiw02] a method to translate Simulink/Stateflow into the input language of the *SAL* [Sha00] model checker. To this end, the Stateflow state machines are first translated into *pushdown automata* which then are translated to SAL. While this approach generally supports variable-step simulation and inductive invariants, both, the translation and the specification of verification goals and invariants have to be done manually. Zou et al. [ZZW$^+$13] presented an approach for the verification of hybrid Simulink models using *Hybrid Hoare Logic (HHL)*. This approach is based on an automatic transformation of the models into the *Hybrid CSP (HCSP)* calculus. The resulting *HCSP* model is verified against a HHL specification with interactive and automatic theorem proving. However, the generation of the *HHL* specification and necessary refinements have to be specified manually. Bouissou and Chapoutot [BC12] presented an operational semantics for Simulink models based on the simulation semantics. Therefore, they defined a translation of Simulink blocks into equations and defined semantic rules. Furthermore, they formalized the *zero crossing detection* for some variable step solvers. However, it requires backtracking steps to determine the point where the zero crossing occurred to calculate the new variable time step. Hence, this semantics is rather suited for symbolic simulation and abstract interpretation. It has not been used for verification purposes so far.

3.1.2 Verification of Discrete Simulink Models

Caspi, Tripakis, Scaife et al. [CCM$^+$03, TSCC05, SSC$^+$04] presented an approach for the translation of discrete-time MATLAB/Simulink models into the synchronous data flow language *Lustre*. To this end, they translate the Simulink blocks to statements or modules in *Lustre*. To obtain data types and sample times, they have presented algorithms to calculate inferred sample times and data types. To verify the models, they have originally used the *Lesar* model checker [RHR91]. Note that the translation enables the use of other verification backends for Lustre like *NBac* [Jea03] and *Gloups* [CDC00]. However, due to the type inference algorithm, the translation is restricted to a limited set of models and the verification goals need to be specified for each verification backend manually.

Meenakshi et al. [MBR06] presented an approach for the verification of discrete-time MATLAB/Simulink models using the model checker *NUSMV* [CCGR00]. It is based on a direct transformation of the models into the input language of *NUSMV* where every block is represented by a module. Although they provide a set of templates, the verification goals have to be specified manually. Furthermore, it is not described how they deal with inferred data types. Arthan et al. [ACOS00] proposed an approach (*ClawZ*) to translate Simulink models into *Z* [WD96] to obtain a specification that can be used for the verification of generated *Ada* code. Based on this technique Cavalcanti et al. [CCO05] presented a technique to translate discrete-time Simulink models into the *Circus* language, which is a combination of *Z* and *CSP*. Furthermore, Cavalcanti, Zeyda and Marriot [ZC09, MZC12] enhanced the technique by further Simulink constructs and presented a tool chain for the automatic transformation of Simulink models to *Circus*. Finally, Cavalcanti et al. [CMW13] presented a timing model for Simulink models using a timed extension of *Circus*, *CircusTime*. These models can be verified by a translation to *CSP* using the *FDR2* model checker. However, in [CMW13] they state that they do not support the verification of run-time errors in Simulink models.

Ryabtsev and Strichman [RS09] presented a method to show that a Matlab/Simulink model is correctly translated by an automatic code generator using the theorem prover *YICES*. They translate both the model and the generated code into the input language of *Yices* and perform induction to show output equivalence of the Matlab/Simulink model and the generated C-code. However, since the arithmetic functions are abstracted with uninterpreted functions, this representation is not suitable to verify the models for the absence of run-time errors. Roy and Shankar [RS10] presented an approach to automatically check the types in a Simulink model. To this end, they transform the model into the input language of *Yices* and use k-induction for the verification. However, they require the manual annotation of types and invariants for a model and use abstractions for mathematical operations.

In [HRB13], we have presented a verification approach based on a translation into the input language of the *UCLID* verifier. However, *UCLID* supports less theories and theory fragments than the *Z3* theorem prover. Hence, we were required to use more over-approximations of block behavior in that transformation such that it is less precise than our approach. Furthermore, the schedule for the blocks is calculated using synchronous data flow instead of the actual scheduling rules of Simulink and control flow and conditional execution contexts are not mentioned in this translation. Boström, Wiik et al. [Bos11, BHH$^+$14, WB14] presented an approach for the contract-based verification of Matlab/Simulink models. It is based on a manual specification of contracts for atomic subsystems and a subsequent translation of the models into verification conditions for the Z3. As in [HRB13], synchronous data flow is used, too. Furthermore, the user needs to specify the type information manually in the contracts. Hence, considerably small atomic subsystems require large effort in contract specification.

The *Simulink Design Verifier (SLDV)* is an additional toolbox for the Matlab/Simulink tool suite. According to a report by MathWorks [Mat07], it uses abstract interpretation and model checking techniques to automatically verify models for for the absence of division-by-zero, dead code and overflows. However, these techniques are subject to scalability issues especially for models with large or unbounded state spaces.

Furthermore, the *SCADE Design Verifier*[1] has been used for the verification of MATLAB/Simulink models, e. g., by Joshi and Heimdahl [JH05]. These approaches are based on a translation[2] of Simulink models into the modeling notation of the *SCADE Suite*. The *SCADE Design Verifier* provides further back-ends [MWC10, HJWW09] like model checkers (e. g., *NuSMV*, *SAL* and *Prover*) and theorem provers (e. g., *PVS* and *ACL*), which can be used for the verification of translated Simulink models. However, it is not specified in the papers, how the translation is done and thus it is not clear whether the simulation semantics of Simulink is modeled or not.

Except for the *SLDV*, all these approaches are based on the assumption that the semantics of Simulink is synchronous. Instead, the actual simulation semantics of Simulink is sequential. There exist mechanisms that introduce the conditional execution of blocks based on this sequential semantics. This means, the synchronous semantics may introduce errors that cannot occur in the sequential semantics. With our approach, we provide a more precise formalization of the MATLAB/Simulink semantics w. r. t. control flow. Furthermore, many of these approaches do not target run-time errors and provide only a low degree of automation since they require user specified annotations and further user interaction. Highly automated approaches, like the *SLDV*, use verification techniques that are subject to scalability issues.

3.2 Slicing of Simulink Models

Model slicing has been a research topic in the last decades. To this end, slicing is not only used to support maintenance and the comprehension of models but also to reduce the complexity for verification. To the best of our knowledge, no slicing approach for Simulink models has been published before our approach [RG12]. There was some unpublished work by Krogh [Kro11] in 2000 where they have developed a slicing tool *sliceMDL* for Simulink for an industrial sponsor. The *sliceMDL* tool uses the signal flow between blocks to slice a model but does not consider execution contexts or control dependence.

However, there exists a number of slicing approaches for other graphical notations. An overview of slicing techniques for various kinds of models (mainly UML dialects) has been presented by Singh and Arora [SA13]. In general, modeling notations can be distinguished whether they model architectural or behavioral aspects of systems. Depending on the desired aspect the techniques utilize different dependence relations, e. g., information flow [Zha98] and inheritance dependence [WY04] for architectural models, or data and control dependence for behavioral models (e. g., [KSTV03, ACH+09, AGH+09, WDQ02]). The slices are either calculated using dependence graphs [KSTV03, ACH+09, AGH+09], model traversal algorithms [GR02, ABC+11], work-lists [WDQ02, VLH07], or predicate transformers [Lan09].

[1]http://www.esterel-technologies.com/products/scade-suite/verify/design-verifier/
[2]http://www.esterel-technologies.com/products/scade-suite/gateways/scade-suite-gateway-simulink/

In the following, we focus our discussion on related approaches for behavioral models of (reactive) systems. They can basically be differentiated into slicing of state-based and synchronous modeling notations for reactive systems.

3.2.1 Slicing of State-based Models

Many approaches have been presented for the slicing of state-based models. State-based models are state machine and automaton dialects. A comprehensive overview about slicing techniques for state-based models is given in [ACH+13]. In the following, we distinguish these approaches whether they use dependence graphs or not.

Techniques not Using Dependence Graphs

Ganapathy and Ramesh [GR02] presented a slicing approach for the state machine dialect *Argos*. *Argos* is a graphical notation that enables the modeling of hierarchical state machines. To slice *Argos* models, they construct a graph for the *Argos* model and enhance the graph by edges representing dependences introduced by hierarchy (*hierarchy edges*) and by the use of variables in transition guards (*trigger edges*). The slice is calculated with a traversal of the graph to the state specified is the slicing criterion, namely a signal of interest. Then, *trigger edges* are traversed to identify concurrent states that may affect the slicing criterion.

Wang et al. [WDQ02] presented an approach for slicing *UML Statecharts* based on a transformation to *extended hierarchical automata*. In this approach they have defined five different dependence relations that comprise data and control dependences within (hierarchical and sequential) and between (parallel) states. To calculate the state, they have presented a work-list algorithm that traverses the dependence relations starting from a transition or state used as slicing criterion. This technique was later improved by van Langenhove and Hoogewijs [VLH07], who have increased the precision of the parallel dependences by considering the execution order of the states. Both approaches remove states and (unconnected) transitions from the model. Lano [Lan09] presented an approach that introduces various techniques for the slicing of *UML Statecharts*. The general slicing algorithm that aims to remove states and transitions w. r. t. a slicing criterion is based on *path-predicates*. To this end, for each possible path to a state of interest in the model, a predicate is calculated using a weakest precondition predicate transformer. States that are not contained in any path are removed from the slice. Furthermore, Lano describes a number of cases where transitions and states can be removed or merged depending on the path predicates and a technique to remove those parts of the transition labels that are unrelated to the variables of interest. The resulting slices are *amorphous* since they may contain states and transitions that are not present in the original model. Due to merging, additional paths may be introduced to the slice that were not possible in the original model. Hence, the resulting slices are not suitable for the verification of certain properties (e. g., reaching of invalid states).

Androutsopoulos et al. [ABC+11] presented an approach for *amorphous* slicing of Extended Finite State Machines (EFSMs). To this end, models are reduced according to input variables provided from the environment. In a first step, all states and transitions that are not triggered by environment signals are removed. In a sequence of subsequent steps, transitions that cannot be triggered anymore are removed and states are merged together.

Techniques Using Dependence Graphs

Korel et al. [KSTV03] first presented an approach for the slicing of state machines using a dependence graph. In this approach, they adopt the notion of control dependence for imperative programming languages based on post-dominance for EFSMs. To this end, they calculate the control dependence for EFSMs based on the post-dominance relation between states. Hence, they require the models to have at least one exit state to be able to calculate control dependence. Later on, Androutsopoulos et al. [ACH+09, AGH+09] improved this technique by introducing the notion of *non-termination insensitive control dependence*. With this relation, they are not only able to slice models without exist states, but also to slice through cycles in the paths of EFSMs. In both approaches, data dependence is calculated from the variable manipulations in transition labels. Fox and Luangsodsai [FL06, LF10] presented an approach for slicing of statecharts. To this end, they introduce the notion of *And-Or-Dependence Graphs* to calculate the slices. The graph is constructed using special *And*-nodes to conjoin the control dependences between states and the elements of the transition labels. Data dependence exists between elements of the transition labels. In the above presented techniques, the slices are calculated using a reachability analysis starting from the nodes representing the slicing criterion.

Although the above presented slicing techniques are used to obtain slices for graphical modeling notations, they are not applicable to obtain slices for MATLAB/Simulink models. Many of the presented dependence relations are tailored to the special features of the models. Furthermore, in state-based models control flow is modeled explicitly and data flow is implicitly given by transition and state labels. This is the exact opposite for Simulink, where data flow is modeled explicitly and control flow may be given implicitly by the simulation semantics. Furthermore, due to the sequential simulation semantics, concurrency is not relevant for Simulink models. However, Korel, Androutsopoulos et al. [ACH+09, AGH+09, KSTV03] show that it is possible to lift slicing techniques for imperative programs to the model level by defining data and control dependence accordingly to the particular modeling notation and execution semantics.

3.2.2 Slicing of Synchronous Models

Clarke et al. [CFR+99] presented an approach for slicing of *VHDL*[3] models. This approach is based on a mapping of the *VHDL* models into a programming language. More precisely, they calculate a CFG for the model and use it as input to a slicing tool for imperative programming languages. However, it is not clear how they deal with concurrency in *VHDL* when calculating the CFG. Kulkarni and Ramesh [KR03] presented a technique for slicing the synchronous programming language *Esterel*. To this end, they calculate a Synchronous Threaded Program Dependence Graph (STPDG), which extends the *threaded dependence graph* introduced by Krinke [Kri98] for the slicing of concurrent programs. They define two new types of (control) dependence: *time dependence* to model control flow introduced by `pause` statements and *interference control dependence* for exception handling. Furthermore, they present an algorithm that calculates slices using the STPDG. However, due to the fact that interference data and control dependences are not transitive, a reachability analysis is not suitable to calculate the slice. Instead, they use a work-list algorithm that iterates over the dependence edges and only add nodes for interference edges to the slice if a corresponding path is possible in the CFG (i. e., if there exists a *trace witness*). Together with Kamat [RKK04], they have later extended this technique to *VHDL* by constructing STPDGs for *VHDL* specifications.

The main challenge for the above presented slicing approaches is the concurrency between threads or processes in the models. To deal with concurrency, non-transitive interference dependences are required to cope with possible interleavings. While in synchronous modeling languages like *Esterel* or *VHDL* there exists concurrency, Simulink schedules all blocks to be executed sequentially. Hence, we can ignore interference dependences for Simulink. However, Kulkarni and Ramesh [KR03] define their new control dependence relations w. r. t. the execution semantics of *Esterel*, which is similar to our slicing approach for Simulink, where we also derive control dependences from the simulation semantics.

3.3 Summary

In this chapter, we have reviewed the related work for our verification and slicing approach. There has been a considerable amount of work on the verification of hybrid and discrete MATLAB/Simulink models. However, most approaches presented for the verification do not use the simulation semantics of Simulink. Instead they give the models a synchronous semantics that is incomplete w. r. t. some control flow mechanisms based on the sequential simulation semantics, which may result in false negatives[4]. Some approaches require manual annotations and only few approaches target run-time errors. Furthermore, some

[3]Very High Speed Integrated Circuit Hardware Description Language
[4]An example is given in Section 6.1.

approaches use model checking techniques that are subject to the state space explosion problem for models with large state spaces. Only few approaches perform a type inference ([CCM⁺03, TSCC05, SSC⁺04, RS10]) or extract inferred information from MATLAB ([ZC09, MZC12, RS09] and [HRB13]). Some of these approaches abstract arithmetic operations ([RS09, RS10] and [HRB13]) which reduces the precision in detecting run-time errors and only [HRB13] and [RS10] generate (some) verification goals for such errors automatically. However, due to the use of a synchronous semantics and the abstractions they are less precise than our approach. Except for [HRB13] that is also based on our *MeMo* tool, none of the presented approaches supports[5] *model referencing*, *library blocks* and *configuration files*, which are very common for industrial Matlab/Simulink designs. To the best of our knowledge, no approach that uses the sequential simulation semantics, is highly automated, uses scalable verification techniques, and targets run-time errors has been published, yet.

The slicing of graphical modeling notations, in particular for reactive systems, has been a research topic in the last decades. There exist many approaches for the slicing of architectural and behavioral modeling languages. The approaches for the slicing of models presented in this chapter show that it is necessary to tailor the dependence analysis for each different modeling notation to the particular syntax and execution semantics. Note that some of the presented approaches [WDQ02, Lan09, VLH07] use slicing as a model reduction technique prior to verification. To the best of our knowledge, our approach is the first ever published for the slicing of Matlab/Simulink models. Furthermore, since our dependence analysis also includes control dependence w. r. t. the simulation semantics, it is more precise than the unpublished work of Krogh [Kro11]. However, this also implies that our approach is the first approach published that combines slicing and verification for Matlab/Simulink models.

[5]Some approaches directly translate composite blocks.

4 Automatic Verification Approach

With MATLAB/Simulink as a de-facto standard[1] in the development of embedded and safety critical controller software, there is a need for comprehensive quality assurance techniques like formal verification. However, the increasing complexity of MATLAB/Simulink models demands techniques that scale well. Furthermore, to be applicable in practice, these techniques need to be not only highly automated but also need to cover an adequate subset of the Simulink language, and to produce comprehensible results.

In this chapter, we present our general approach for the verification of discrete-time MATLAB/Simulink models to tackle these requirements. Our approach is based on a fully automatic translation of the MATLAB/Simulink models into the Boogie programming language. A preliminary version of this translation approach has been published in [RG14a]. To provide additional automation, we also automatically generate proof obligations during the translation to verify the absence of important run-time errors. So far, we consider the following error classes: *overflows*, *underflows*, *division-by-zero* and *range violations*. However, the verification approach is not limited to these error classes.

To tackle the issue of scalability, we use inductive verification techniques to prove the absence of these errors. Furthermore, we present a slicing approach for Simulink to reduce the size of the models for a specific point of interest (e. g., a potential error) and hence the verification effort.

In the following sections, we introduce our approach in detail. We first present the general idea of our framework, which is a translation of a given *informal Simulink model* into a *formal Boogie2 specification*. Afterwards, we present the core components of our framework and describe the process for the verification of Simulink models using our framework.

[1]http://de.mathworks.com/company/user_stories/product.html

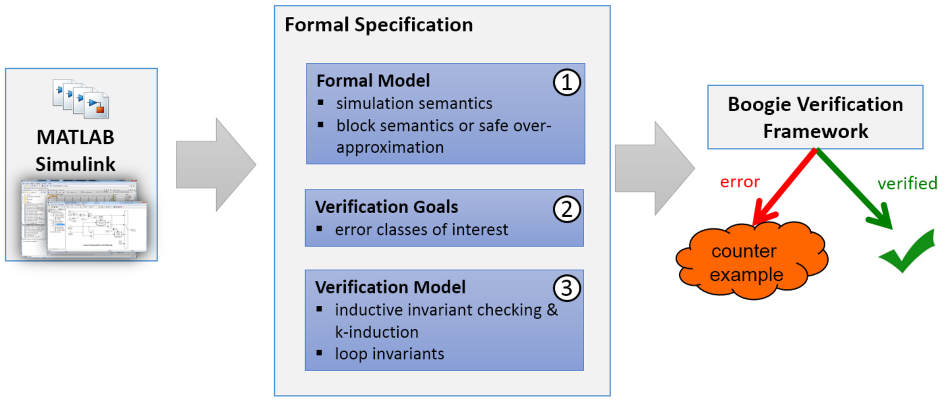


Figure 4.1: The General Verification Idea

4.1 Verification Idea

The aim of our framework is to provide an automated verification environment for Matlab/Simulink models that is scalable and applicable in practice. To achieve scalability, we propose the use of inductive techniques. To achieve practical applicability, we aim for a verification flow that is mostly automatic and requires low user interaction. Furthermore, we use an underlying verification layer that has proven to be useful in practice.

However, formal verification techniques require a formal model for the system to be verified. Hence, the main challenge for our approach is the fact that neither for the graphical notation nor for the actual simulation of Matlab/Simulink models, a formal semantics has been published by *The MathWorks*.

To overcome this problem, we propose a translation of a subset of Simulink models into the Boogie2 intermediate verification language, which serves us for both: as a formal model and as input for the Boogie verification tool. The general idea of the translation is depicted in Figure 4.1. In addition to (1) the formal specification for Matlab/Simulink models, the translation also extracts (2) verification goals for the previously mentioned error classes and (3) additional information like loop invariants from the original model. Hereby, the additional information from (2) and (3) is used to achieve a higher degree of automation since the user does not need to specify verification goals and those invariants manually.

Formal Model While the notation of Matlab/Simulink models is generally concurrent, the simulation of Matlab/Simulink models is done sequentially, just as the code that is generated for controllers. During a simulation, the Matlab/Simulink environment calculates an execution order for the blocks, initializes the blocks and executes the model for a given number of time steps. Hence, in our automatic translation, we first have to calculate the block order to construct a control flow graph *(CFG)*, which then can be translated into a Boogie program. Once the CFG is calculated, we translate the model according to the CFG block by block. For each supported block type, we define translation rules specifying the translation according to the block semantics, the relevant parameters and the signals connected to the inputs.

Verification Goals Depending on the block type, verification goals are created for each block. For each block where overflows, underflows, division-by-zero and range violations can occur, the necessary checks are produced according to the parameters of the concrete blocks.

Verification Model Besides the parameters necessary for the translation of the blocks, we additionally extract information from the model that helps with the verification, e. g., saturation limits, data types, and lower and upper bounds for signals. This information is used to automatically create loop invariants in the specification. With our approach, we support two verification methods: *inductive invariant checking* and *k-induction*. The first is naturally supported by Boogie, for the second we create a special Boogie2 specification for a desired k.

Finally, the formal specification for the Matlab/Simulink model is passed to the Boogie tool which either returns a counterexample if the model contains an error or returns that the model is verified. However, the counterexamples reported may be false negatives. These may occur due to abstraction, due to missing invariants or due to invariants that are too weak to verify the absence of errors. To deal with the latter, our framework makes use of k-induction as presented in Section 2.3.2. However, the size of the verification conditions increases with the size of k. To overcome this problem, we apply slicing techniques to the model to generate smaller verification conditions.

With this approach we are able to achieve a high degree of automation since the transformation, the generation of the verification goals and the verification techniques are mostly automatic. Moreover, we can achieve scalability due to the verification techniques used and the additional model reduction techniques.

4.2 Verification Framework Architecture

In this section, we present the general architecture of our verification framework. Figure 4.2 depicts the components and the interaction between them within our framework. There are three main components: the *transformation engine*, the *verification engine* and the *slicing engine*.

Transformation Engine This is the core component of the framework. It takes a Matlab/Simulink model and transforms it into the formal model. Hereby it not only translates the model, it also generates verification goals and invariants as mentioned in the previous section. Besides transforming a full model, it is also able to transform partial models generated by the slicing

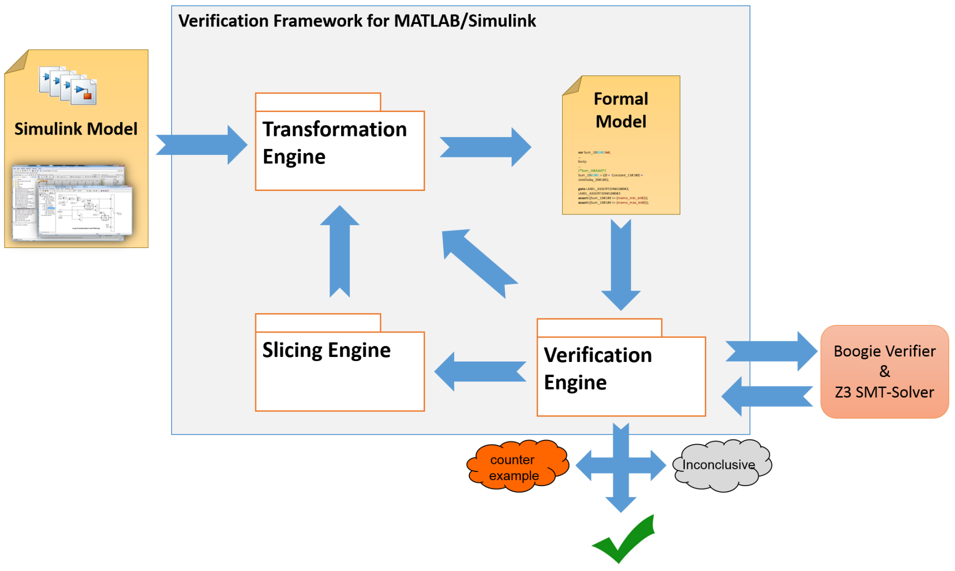


Figure 4.2: The Architecture of the Verification Framework

engine. Furthermore, the translation is parameterizable with a k to generate specifications for k-inductive invariant verification. This is necessary since Boogie currently does not support k-induction by itself. A detailed description of the transformation engine is given in Chapter 6.

Verification Engine The verification engine basically is the interface to the underlying Boogie verification framework. It takes the formal specification in Boogie2 and invokes the Boogie tool. When invoking the Boogie tool our engines also specifies some parameters and flags depending on some properties of the model (e. g., use of fixed point data types) and the desired verification method. To visualize results and ease debugging, our engine processes the output of the Boogie tool. Our engine uses the structural information about the model that is encoded into the formal specification to assign the errors to the blocks from the original Simulink model.

Slicing Engine The slicing engine is the component that applies our slicing technique to the models. A preliminary version of this technique has been published in [RG12]. In our framework, the slicing component is used to slice a model with respect to a certain error. More precisely, the blocks, for which errors were reported in a previous verification attempt, are used as slicing criterion for a backward slice. Since static slicing is a safe over-approximation and calculates all execution paths that may reach the block of interest, the resulting slice is suitable for subsequent translation and verification attempts.

To be practically applicable, all of these components are integrated in our MeMo tool suite. The transformation engine and the slicing engine require information about the model, which are usually not available from the MATLAB/Simulink model files. Hence, our tool provides a two-phased parser that

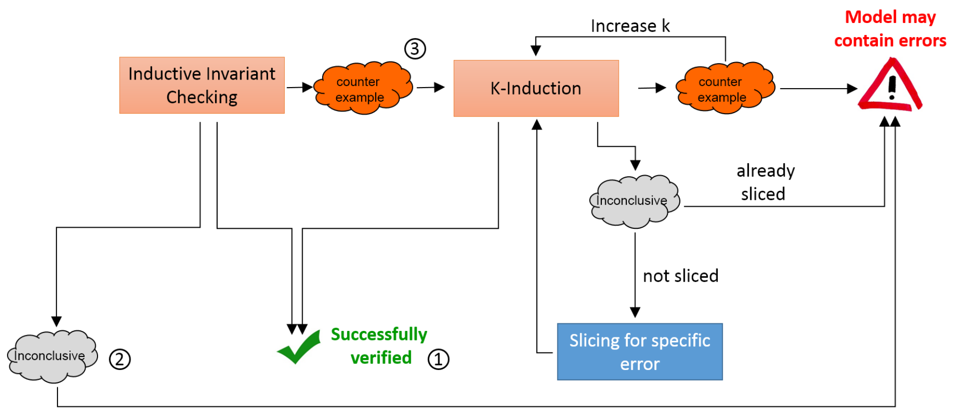


Figure 4.3: The Verification Process

enhances the information taken from the model file by adding the information inferred by the Simulink tool at run-time. This avoids the reimplementation of the inference mechanisms of MATLAB/Simulink, whose description is informal and incomplete in the MATLAB/Simulink documentation. Furthermore, the MeMo tool suite performs a number of analysis steps needed by the components, e. g., the identification of entry and exit point of signals in bus systems.

4.3 Verification Process

We propose a process how to use the components of our verification framework. This process is depicted in Figure 4.3.

The general idea of our process is to transform a given MATLAB/Simulink model into Boogie2, and then start with the most scalable verification technique first: inductive invariant verification. This results in three different outcomes: (1) the Boogie tool successfully proves the absence of errors, (2) the Boogie tool is not able to show the absence of errors within a given time bound, or (3) the boogie tool reports at least one counterexample. In case of (1), the process is finished since the model is verified. In case of (2), the model is too complex to show the absence of errors even with the most scalable technique. Hence, manual intervention is needed to make the model less complex, e. g., slicing or compositional techniques. In case of (3), where errors are reported, these counterexamples still may be false negatives (*spurious* counterexamples).

Spurious counterexamples may occur for two reasons, either they are caused by abstractions or by invariants that are not strong enough. However, abstractions are necessary due to restrictions to the arithmetics supported by verification tools and stronger invariants may require manual intervention. With k-induction it is possible to verify certain properties using weaker invariants as long as it is possible to find a suitable k.

Hence, the next step in our process is to try to eliminate the spurious counterexamples by performing verification attempts with an increasing k. Again, if we succeed in showing the absence of errors, we are finished. As long as errors are reported by the verification attempts, k is increased. However, since this technique requires the unrolling of the system for k times, it is less scalable than inductive invariant verification. Hence, it is possible that no verification results are obtainable within a given time bound for a k for all errors taken into account. At this point, slicing is used to reduce the model for a certain error and k-induction is applied to the reduced model. Finally, if the verifier is still not able to show the absence of errors, the counterexample has to be inspected manually. At least, since the counterexamples returned by k-induction usually describe a trace of k steps, it is more comprehensible than counterexamples returned by inductive invariant verification.

With the process presented in this section, we can combine our verification approach with our model reduction approach in order to achieve both: automation and scalability. The process consists of three stages:

1) Use the *most scalable* technique (inductive invariant checking) first to verify the absence of *all errors* under consideration.

2) Use k-induction that is more automated since it allows for the use of weaker invariants to show the absence of *all errors*.

3) Use automatic model reduction techniques to show the absence of a *specific error* from 2) on a model slice with a higher k than in 2).

With each stage, the number of properties that can be verified automatically increases for the cost of more computational effort. By combining the strength of inductive invariant verification, k-induction and slicing, we achieve a verification process that provides an effective trade-off between automation and scalability. This enables us to verify complex systems with reasonable computational effort and minimal user-interaction.

4.4 Summary

In this chapter, we have presented our general verification idea which is based on a translation of discrete-time MATLAB/Simulink models to Boogie2. Furthermore, we have presented the components of our verification framework and our proposed process for the efficient use of our framework. In the following chapters, we first present our slicing approach in Chapter 5. Then, we present our automatic translation of Simulink into the formal model in Chapter 6 before we introduce the automatic generation of the formal specification that consists of the formal model, verification goals and loop invariants tailored to the respective verification strategy in Chapter 7.

5 Slicing of Simulink Models

In this chapter, we present our slicing approach for MATLAB/Simulink. With this approach, we address the problem of reducing the complexity of a model. Therefore, we remove those parts of the model that do not affect a particular block. We require our slicing method to preserve the semantics of the residual model to be suitable for subsequent automatic verification techniques. Furthermore, the slice has to be as precise as possible since Simulink models may consist of a huge number of blocks where each irrelevant block results in unwanted complexity.

We propose a static slicing technique for MATLAB/Simulink models using a dependence graph based approach. The basic approach has been published in [RG12]. The core idea of our approach is to transfer the techniques known from classic program slicing to the model level. Therefore, we use the fact that the simulation semantics of the models in MATLAB/Simulink is sequential.

In classic program slicing, a static slice is a subset of the program that contains all statements that affect or are affected by a specific point in the program. In other words, a slice contains all possible execution paths from or to a specific point in the program. In terms of Simulink models, we consider (basic) blocks, similar to statements, as the smallest unit of execution. Hence, we define the slicing criterion for Simulink models as:

Definition 5.1 (Slicing Criterion for Simulink)**.** *The slicing criterion $\boldsymbol{C}$ for a Simulink model $\boldsymbol{m}$ is defined as a set of blocks: $\boldsymbol{C}(\boldsymbol{B})$ where $\boldsymbol{B} \subseteq \boldsymbol{m}$.*

Since the execution of the models is done by the simulation engines, we define a static Simulink slice as:

Definition 5.2 (Static Slice for Simulink)**.** *A static slice of a Simulink model for a slicing criterion $\boldsymbol{C}$ is the subset of all blocks that*

*(i) are either affected by the execution of the blocks in $\boldsymbol{C}$ (**forward slice**) or*

*(ii) affect the blocks in the set of blocks in $\boldsymbol{C}$ (**backward slice**).*

A block $\boldsymbol{b}_1$ affects a block $\boldsymbol{b}_2$ if the block $\boldsymbol{b}_2$ is directly or transitively control or data dependent on $\boldsymbol{b}_1$.

In other words, a Simulink slice contains all blocks that potentially may influence the calculations of the blocks in the slicing criterion or all blocks whose calculations are potentially influenced by the block in the slicing criterion in all possible executions of a model.

Our approach consist of two parts:

1. A **dependence analysis** for Matlab/Simulink models for control and data dependence.
2. A **slicing approach** for Matlab/Simulink that preserves hierarchy and semantics within the slice.

The main challenge in our slicing technique is the dependence analysis. This analysis gathers and calculates the information about the data and control dependences between the blocks that are necessary for the static slicing technique. Furthermore, it provides these dependences to our translation into the formal model in Boogie where they are used to calculate the Control Flow Graph (CFG) for the model.

In the subsequent sections, we first discuss the assumptions and limitations to the models to make our approach applicable in Section 5.1. Then, in Section 5.2, we present the dependence analysis in detail and discuss data and control flow in Simulink models. After that, in Section 5.3, we present our general slicing approach before we introduce further enhancements that increase the precision of the slices in Section 5.4.

5.1 Assumptions

Our slicing approach aims at two goals: On the one hand, we use it as an automatic model reduction technique within our verification framework to reduce the state space and, as a result, the size of the formulas. On the other hand, the slicing approach can be also used for tasks like model comprehension, reviews or deriving quality metrics [HLW12]. For the latter purposes, it is possible that only partial models are available in practice since model parts are developed by different partners. Hence, we propose a *basic slicing* approach suitable for even incomplete models but for the cost of precision that works on the Matlab/Simulink model file. Furthermore, we present and an enhanced *routing-aware approach* that requires complete models but provides more precise slices.

There are minimal requirements to the models to enable the basic slicing approach. For the basic approach, there are the following assumptions and limitations to the models:

- **No algebraic loops** The models are not allowed to contain cyclic signal flow without a delay element.
- **No multirate blocks** The documentation lacks a clear definition of *multirate* blocks. These blocks should be avoided since their presence

may lead to less precise slices due to over-approximations during the dependence analysis.

- **Over-approximations** Stateflow machines, `SFunction` blocks, bus systems, multiplexer, demultiplexer and unavailable references are over-approximated by assuming that all outputs depend on all inputs. This is necessary since they either implement complex behavior in other modeling and programming languages, or the model file does not contain enough information because it is inferred at compile time.

For the routing-aware approach, we require the model to be *executable* (complete). With the routing-aware approach we also support precise slicing of bus systems. However, Stateflow machines and `SFunction` blocks, multiplexer and demultiplexer are still over-approximated. The latter are over-approximated since there are a number of blocks that can directly operate on vectors and may change the ordering of the signals.

5.2 Dependence Analysis

In this section, we present our dependence analysis for Simulink to identify the dependences between the model elements. Since Simulink is a data flow oriented graphical notation, data flow is basically given by the structure and (more or less) directly modeled. Control flow, to a certain degree, is also modeled directly, but there are some mechanisms that introduce implicit control flow, too. In this section, we explain how we determine data and control dependence in Simulink models. To this end, we first discuss data flow and define data dependence. After that we discuss control flow, and the relation between Execution Context (EC) and control dependence. Finally, we present our algorithm for the calculation of control and data dependences.

5.2.1 Data Flow and Data Dependence

While in most imperative programming languages control flow can be easily derived from the structure of the program and data flow needs to be calculated using complex analyses, in Simulink the opposite is the case. Data flow in Simulink is modeled by signal lines. Hence, data dependence can mainly be derived by following signal lines.

Typically, data flow in Simulink is specified by defining a line L connecting two blocks b_1 and b_2. This means that an output signal l of b_1 is used as an input signal for b_2. In analogy to imperative languages, l is defined in b_1, referenced in b_2 and transmitted via L. In this case b_2 is directly data dependent on b_1.

Definition 5.3 (Data Dependence in Simulink). *A block b_2 is data dependent on a block b_1 if*

(i) b_1 and b_2 are connected by a line L and

(ii) L starts from an output of b_1 and ends in an input of b_2.

This definition is rather coarse. Especially in the presence of bus systems, it leads to imprecise slices since there may be multiple signals encoded in a single line that are not necessarily all relevant for a certain slicing criterion. However, since the actual structure of bus system signals is calculated at compile time (see Figure 2.5 in Section 2.2.2), the definition is suitable to slice models using only information from the MATLAB/Simulink model file. Hence, with that approach we are also able to slice incomplete models that are not executable.

To deal with executable models containing bus systems, we have developed an enhanced (routing-aware) analysis to automatically annotate the bus signals carried by a line. These annotations are then used for dependence graph construction. We discuss this extension in Section 5.4.

5.2.2 Control Flow in Simulink

Although Simulink is a data flow oriented notation, there are possibilities to model the conditional execution of model parts. To understand control dependence in Simulink, we first discuss how control flow is introduced into a given model.

Control flow in Simulink models can occur at

(1) *conditional subsystems* or

(2) *loop subsystems* or

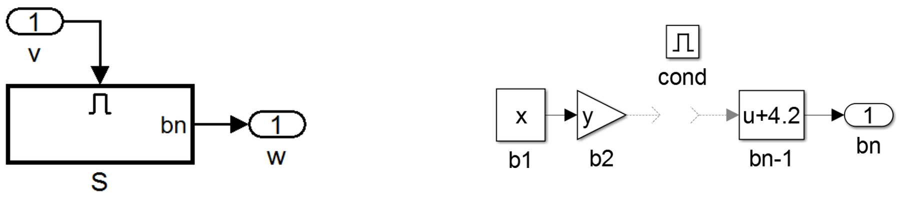
(3) *switch blocks* (`MultiPortSwitch` and `Switch` blocks).

Furthermore, implicit control flow may be introduced by certain parameters to the simulation engine or model elements. In the following three subsections, we discuss these model elements in detail.

Conditional Subsystems

Conditional subsystems are subsystems that are only executed during the simulation if a certain predicate holds. The predicate depends on the type of the conditional subsystem. There are `Enabled`, `Triggered`, `Enabled & Triggered`, `Action` and `Function-Call` subsystems. They are all characterized by (at least) one special input that controls the execution of the subsystem. Depending on the type of the subsystem, the signal connected to the special input is checked for a specific property.

Figure 5.1 depicts an `Enabled` subsystem and its contents. The special input, the *enable* port, is on the top of the subsystem (Figure 5.1a). The subsystem is executed if the signal connected to the input is greater than `0`. Within the subsystem (Figure 5.1b), there is an `EnablePort` block labeled



(a) An `Enabled Subsystem` (b) Contents of the `Enabled Subsystem`

Figure 5.1: An `Enabled Subsystem` and its contents

with `"cond"`. This block corresponds to the input on top of the subsystem in Figure 5.1a.

Blocks contained in a conditional subsystem are only executed if the condition given by the special port is fulfilled. Conditional subsystems are *atomic* and hence have their own execution context. Moreover, these subsystem need an additional configuration for their outputs since they may not be executed in every simulation step but their outputs are required by subsequent blocks. This may be either a specified default value or the outputs stored from the last execution.

Basically, conditional subsystems are comparable to *if-then-else* statements, in which the contents of the subsystems are executed in the *then*-branch and the default behavior is the *else*-branch. In addition, there exist `Switch-Case` and `If-Then` blocks. These are specialized blocks that drive `Action`[1] subsystems. Although, the actual conditional execution is done by `Action` subsystems, these block may increase the comprehensibility of the model.

Loop Subsystems

Besides conditional subsystems, Simulink also provides loop subsystems. The common loop constructs like *while* and *for* are realized by atomic subsystems. Similar to conditional subsystems, they contain a special block that controls the execution of the subsystem and are atomic subsystems. In contrast to conditional subsystems, the contents of loop subsystems can be executed multiple times in a single simulation step. Furthermore, the states of blocks are updated in every execution of the subsystem.

The block that triggers the execution of a loop subsystem is either a `WhileIterator`, a `ForIterator` or a `ForEach` block. A `WhileIterator` block triggers the execution of the subsystem as long as a condition is true. Depending on its configuration, the block may need an additional input for the initial condition if it models a *while* loop. For a *do-while* loop no initial condition is needed. A `ForIterator` block triggers the execution of the subsystem for a given number of times. A `ForEach` block iterates over dimensional input

[1]An `Action` subsystem behaves basically the same as an `Enabled` subsystem. The only difference is that the type of the signal is *"action"* that can only be produced by a few number of blocks, e. g., `Switch-Case` and `If-Then` blocks.

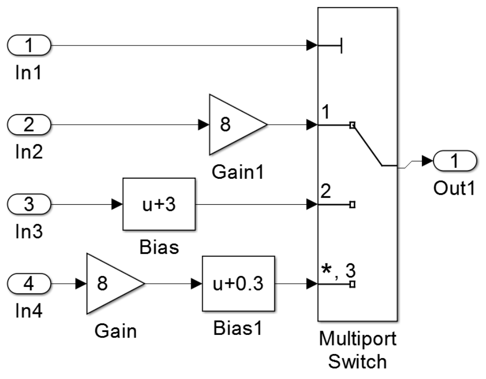


Figure 5.2: A MultiPortSwitch

signals and execute the contents of the subsystem for each specified element in the input signal.

Switches

`Switch` or `MultiPortSwitch` blocks are used in Simulink to forward a specific signal depending on a control signal. Depending on the value provided to the control input, the signal attached to the corresponding data input is forwarded to the output of the `Switch` or `MultiPortSwitch` block. While the `Switch` block only has two data inputs, a `MultiPortSwitch` block may have more. In both cases, during the simulation of these blocks only one data input is selected at a time. Hence, only the blocks that are relevant and connected to the selected input need to be executed while the execution of the others does not influence the result.

To optimize the simulation of a model, the Simulink environment provides the *Conditional Execution Behavior* mechanism to avoid unnecessary execution of model elements. If *Conditional Execution Behavior* is active, the Simulink environment only executes those blocks that supply data to the selected input port of the `Switch` or `MultiPortSwitch` block. For that purpose, the simulation engine internally creates `Enabled` subsystems for each data input.

Figure 5.2 depicts a `MultiPortSwitch` block. The topmost input is the control input. The three inputs below are data inputs. Depending on the value of the signal at the control port one of the data ports is chosen.

In conclusion, the conditional execution of blocks in Simulink is always realized by *conditionally executed atomic subsystems*. Hence, the key idea to determine control dependences is to investigate how Simulink internally treats conditional subsystems.

5.2.3 Conditional Execution Contexts and Control Dependence

As mentioned in Section 2.2.2, for each atomic subsystem a new Execution Context (EC) is created. The blocks contained in the subsystem are assigned to the corresponding EC and scheduled according to their data dependences. Further ECs that are introduced by these blocks are scheduled within their parent EC. Note that there always exists one execution context for the entire model, where all blocks not assigned to other ECs are scheduled according to their data dependences. We refer to this top-level context also as *root EC*.

Once the initialization phase of a given model is finished, two types of ECs can exist:

(1) Execution contexts that are executed in any step of the simulation loop created for plain atomic subsystems, and

(2) execution contexts that are only executed if a certain condition for the value of a specific signal is satisfied.

We refer to (2) as Conditional Execution Contexts (CECs). Moreover, the condition for the execution is realized by special port or iterator blocks. In the following, we refer to this special port or iterator block as *predicate block*.

Execution of Models

Once all execution contexts are calculated and all other initialization steps are done, the model is simulated. To this end, the selected simulation engine samples the model at various instants in time. Although Simulink is used to model reactive systems that are potentially non-terminating, the simulation always finishes after a given number of steps.

Figure 5.3 depicts an abstract control flow graph for the simulation of a model. Once the initialization is done, the simulation loop is entered. Then, the blocks in the root EC are executed in their sorted order. If the simulation engine encounters a conditional subsystem, the condition given by the predicate block (*Pred_1*) is evaluated. If the condition is not satisfied, the next block of the root EC is executed. Otherwise, the schedule of the corresponding child EC (*CEC1*) is executed. The execution does not return to the root EC until all blocks and further nested ECs are executed. Finally, when the last block in the root EC has been executed, the simulation loop invokes the execution for another step or the simulation is finished.

This means, Simulink models have the following execution behavior:

(1) A model is simulated for a given number of steps. In each of these simulation steps, the model is executed fully if no run-time error occurs. The simulation of a Simulink model stops if the simulation loop is left after executing the model.

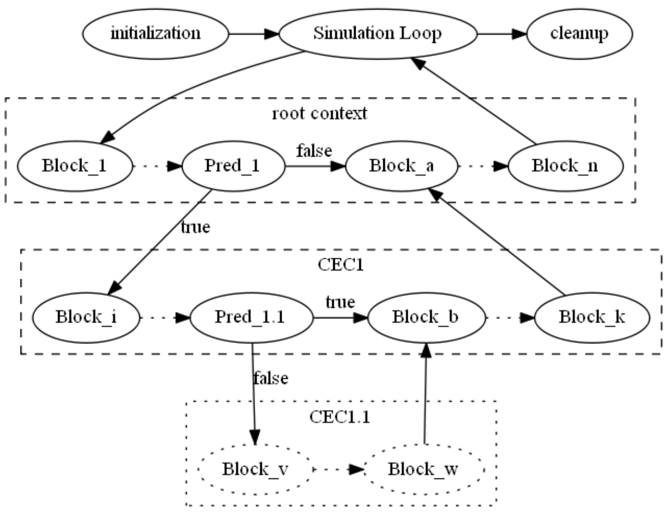


Figure 5.3: CFG for an Arbitrary Model with Nested Control Flow

(2) By definition of ECs (Definition 2.1) the following holds: Once an EC is entered, **all** blocks within the EC have to be executed. This also holds for the root context.

These properties enable us to transfer the notion of control dependence from imperative programming language to MATLAB/Simulink models. As presented in Section 2.5, control dependence is defined with respect to post-dominance, which in turn requires termination. Termination is given by property (1). Furthermore, (2) also implies that there is only structured control flow.

Control Dependence

With the properties presented in the previous section, it is generally possible to calculate control dependence for Simulink models by transferring the techniques known from imperative programming languages to the model level. However, these techniques require the calculation of the CFG for a given model.

It is possible to extract control dependence directly from the conditional execution contexts in the models: Assume there is a block b in a CEC e and e is not the root EC. The predicate block p that triggers the execution of e is part of the parent EC e_{parent}, which may be the root context. For this block b, it holds that

(i) there exists a path from p to b such that b post-dominates every block in this path excluding p and b, and

(ii) p is not post-dominated by b.

First, there exists a path from p to b because if p triggers the execution of e, all blocks in e are executed including b (atomic execution). Second, b post-dominates every block in the path because if e is executed,

(a) all blocks in e are always executed in the same sorted order, and

(b) the simulation terminates (there is a path to the exit node of the CFG).

Finally, p is not post-dominated by b since there is a path to the exit node where the execution of e is not triggered. Hence, b is control dependent on p.

Accordingly, we define control dependence for MATLAB/Simulink models as follows:

Definition 5.4 (Control Dependence in Simulink). *In a model m with the root execution context e_m containing the blocks b_1 and b_2, b_2 is control dependent on block b_1 if*

1. *b_2 is within a conditional EC $e \neq e_m$ and*
2. *b_1 is the predicate block controlling the execution of e.*

The blocks contained in e_m are control dependent on the simulation loop.

The key problem for the calculation of control dependence is that neither the CFG nor the ECs are available from the model file or from the MATLAB/Simulink tool itself.

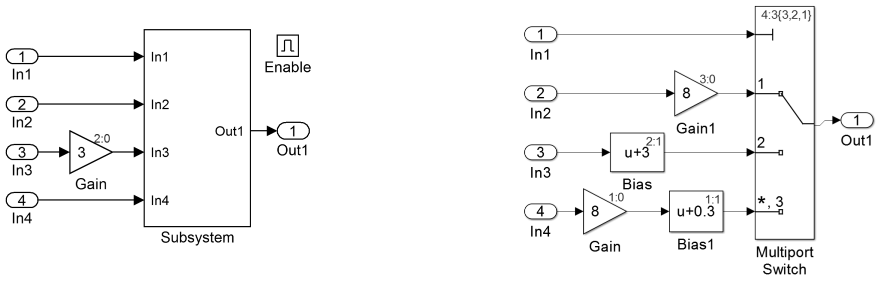
5.2.4 Calculation of Control Dependence

In the previous section, we have shown that control dependence for Simulink models can be derived from Conditional Execution Contexts. CECs are an internal representation in the MATLAB/Simulink tool that cannot be accessed, neither from the model file nor by MATLAB commands. This requires us to calculate the CECs in Simulink models first, in order to perform a dependence analysis. The main problem in the calculation of CECs is that under certain conditions, a CEC can be *propagated* to blocks outside of the corresponding conditionally executed subsystem.

Conditional Execution Context Propagation

Conditional Execution Context Propagation is a mechanism in Simulink that propagates the CECs of control flow elements through other blocks and different levels of hierarchy. It is activated in two ways: either it is triggered explicitly by conditional subsystems if the parameter `PropExecContextOutsideSubsystem` is set to `ON`, or it is activated for switch blocks if *Conditional Execution Behavior* is set active for a model.

In the first case, blocks that meet certain conditions and are connected to the inputs and outputs of a conditionally executed subsystem are added to its CEC. In the second case, for each data input of switch blocks, an `Enabled` subsystem is created internally with the propagation set active.



(a) The `Subsystem` containing the `MultiPortSwitch`

(b) A `MultiPortSwitch` with ECS and Sorted Order

Figure 5.4: Propagated ECs for a `MultiPortSwitch`

Figure 5.4 depicts an excerpt of a Simulink model with *Conditional Execution Behavior* set active. For the model, the displaying of the sorted order has been activated. Every non-virtual block has its execution context and its (zero-indexed) position displayed in the upper right corner separated by a colon. In addition, the `MultiPortSwitch` in Figure 5.4b has the CECs it introduces annotated in curly brackets. Its annotation $4 : 3\{3, 2, 1\}$ reads as follows: The block is assigned to CEC 4 and in position 3 in the sorted order. Furthermore, it introduces the CECs 1, 2, and 3 for the internally created `Enabled` subsystems. The CEC 3 is only propagated to the `Gain` block "`Gain1`". The block itself is on position 0 and the sole block within CEC 2. It is only executed if the first data input is selected. The CEC 2 is created for the second data input, the `Bias` block within the subsystem, and the `Gain` block outside the subsystem (see Figure 5.4a). Here, the CEC is propagated one level up in the hierarchy of the model. Both blocks are only executed if the second data input is selected.

In the MATLAB/Simulink documentation, MathWorks specifies the following rules for the propagation of execution contexts:

Definition 5.5 (Conditional Execution Context Propagation [Mat14b])**.** *A block is added to an execution context if*

(1) its output is required only by a conditionally executed subsystem or its input changes only as a result of the execution of a conditionally executed subsystem, and

(2) its sample time is inherited, and

(3) the output of the block is not a testpoint, and

(4) the block is allowed to inherit its conditional execution context, and

(5) the block is not a multirate block.

According to these conditions, the Simulink environment calculates the execution context for each block. Since we cannot extract the information about the CECs from the model, we propose our own algorithm for the calculation of CECs published in [RG12].

Calculation of Conditional Execution Contexts

The basic idea of our algorithm is to traverse the model top down in a *depth-first search* to identify all blocks that introduce a new CEC and then to perform the propagation if necessary. To this end, we analyze all paths leading to or coming from a block that propagates its execution context. By analyzing the paths we can detect if the blocks satisfy the condition (1) of Definition 5.5. Additionally, while analyzing the paths, we check the parameters of every block in the path. (2) or (3) can be directly derived from the parameters of the blocks (or of the inputs and outputs).

We do not evaluate condition (5) since it is not exactly specified what a multirate block is. Furthermore, the Simulink documentation lacks for sufficient information (except for two examples), which blocks are not allowed to inherit an execution context (4). The documentation gives two examples for such blocks, which are also respected by our analysis. We stop the path analysis if one of the first four conditions is not satisfied.

Note that not including (5) and only partially supporting (4) weakens the conditions for the propagation of CECs. That means, additional blocks may be added to a CEC in some very rare cases. This is actually an over-approximation of the existing control dependences. Hence, we preserve the correctness of the slices computed by our approach but the over-approximation may lead to a decrease in precision. Since (4) and (5) are block-specific properties like (2) and (3), the algorithm can be easily extended.

To compute the conditional execution contexts, we recursively traverse the model top-down through the hierarchy as shown by Algorithm 5.1. It starts on the highest hierarchy level of the model with the empty root CEC as *curCEC* and operates in three phases as described in the following.

First, the algorithm iterates over all blocks within the current hierarchy level to search for blocks that introduce a new conditional execution context. These are either conditionally executed subsystems or loop subsystems. If such a block is found, a new execution context is created and added to the CEC that is currently processed (*curCEC*). Then, the algorithm is invoked recursively on the newly created CEC with all blocks contained in the subsystem. The recursive traversal stops if the lowest hierarchy level is reached. If the parameter `PropExecContextOutsideSubsystem` is set for the subsystem, the execution context is propagated in forward and backward direction and the execution contexts are filled bottom up.

In the second phase, once all conditional and loop subsystems of a certain hierarchy level have been identified, all remaining blocks are processed. Hereby, the algorithm again iterates over all blocks of a hierarchy level searching for blocks that are not part of any of the previously constructed CECs. If a block is a virtual subsystem, the algorithm is invoked recursively on the blocks within the subsystem using the current CEC. Otherwise, the blocks are directly added to the current CEC.

Algorithm 5.1: Search for all Execution Contexts in a Model

```
Function findExecutionContext(blocks, curCEC) as
    /* find new CECs first */
    for b in blocks :
        if b is cond. or loop subsystem :
            create new EC e;
            add e to curCEC;
            CALL findExecutionContext(b.childs, e);
            if PropExecContextOutsideSubsystem = ON :
                for dataIn of b :
                    propagateBackward(b, dataIn, e, curCEC);
                for dataOut of b :
                    propagateBackward(b, dataOut, e, curCEC);
    /* add remaining blocks to current CEC */
    for b in blocks :
        if b is not in curCEC :
            if b is virtual subsystem :
                /* taverse the virtual hierarchy */
                CALL findExecutionContext(b.childs, curCEC);
            else:
                add b to curCEC;
    /* process switches if neccessary */
    if Conditional Execution Behavior is active for the model :
        for b in blocks :
            if b is Switch or MultiportSwitch :
                for dataIn of b :
                    create new EC e;
                    add e to curCEC;
                    propagateBackward(b, dataIn, e, curCEC);
end
```

The third phase of the algorithm is only triggered if *Conditional Execution Behavior* is enabled for the model. Then, the algorithm searches for switch blocks. If a switch block is found, an execution context is created for each data input and added to *curCEC*. Afterwards, the propagation of execution contexts in backwards direction is invoked.

The propagation of the execution contexts (*propagateForward()* and *propagateBackward()*) is basically implemented with a work-list algorithm that iterates over the inputs or outputs connections of blocks. The algorithm for the propagation in backward direction is depicted in Algorithm 5.2. The forward propagation is done analogously. Note that in our algorithm, line branches are resolved into signal lines between blocks.

The algorithm takes four parameters: a signal line connected to the input from which the execution context is propagated, the block from where the propagation starts, the new CEC that is propagated and the CEC of the

Algorithm 5.2: Propagation of Conditional Execution Contexts (Backward)

```
Function propagateBackward(sigLine, target, newCEC, oldCEC) as
    worklist = {sigLine}; addlist = $\emptyset$; inspectlist= $\emptyset$;
    while worklist $\neq \emptyset$ :
        for line in worklist :
            b = source(line);
            if check2(b) $\wedge$ check3(line) $\wedge$ check4(b) :
                if oneOutput(b) :
                    if getCEC(b) is oldCEC or none :
                        newCEC.add(b);
                        addlist.add(b.insignals);
                else:
                    inspectlist.add(b);
        worklist = addlist; addlist = $\emptyset$;
    for block in inspectlist :
        if targets(outputs(block)) $\subseteq$ newCEC $\wedge$ block $\notin$ newCEC :
            newCEC.add(block);
            for s in block.insignals :
                propagateBackward(s, block, newCEC, oldCEC);
end
```

block. In a first phase, the algorithm performs a breadth-first search over the input signals within the *worklist*. For every source block of an input signal, it is checked whether the conditions ((2),(3) and (4)) for the propagation of the execution context hold. If they hold, we differentiate whether the block has only one or more outgoing signals. Blocks that have only one outgoing signal are added to the execution context and their input signals are added to the temporary list (*addlist*) for the next iteration over the work list. Blocks that have more than one output signal are added to a list for further inspection (*inspectlist*) since they may violate the condition (1) of Definition 5.5. A block with two or more outputs violates (1) if at least one of its outputs is not required by the conditional executed subsystem. Once the work list has been processed, the newly identified signal lines are added to the work list. The first phase stops if no new signal lines are identified.

In the second phase, all blocks that have more than one output are processed. To identify the blocks that satisfy condition (1), the algorithm checks the target blocks of all outgoing signals. If all those blocks are already in the propagated CEC (*newCEC*), i. e., they have been added by the breadth-first search, condition (1) is satisfied. Then the block is added to the new CEC and the propagation algorithm is invoked recursively starting from this block for all its inputs, since it may be possibly propagate the context to its input signals. If not, the block is discarded.

The algorithm terminates since blocks already included in the propagated CEC are not processed twice (Line 8 and Line 15), and there exists only a finite number of blocks in the model.

Once all CECs for a model are calculated using the algorithms presented in this section, control dependence can be directly derived from the calculated CECs as shown in Definition 5.4.

5.2.5 Summary

In this section, we have presented our dependence analysis for MATLAB/Simulink models. Therefore, we have defined data and control dependence in Simulink models. Data dependence is directly derived from signal lines. Control dependence is derived from conditional execution contexts within the model. Since CECs are not accessible to the user, we needed to calculate these for a model first. To overcome this problem, we have presented an algorithm to calculate the CECs, which is also able to deal with *Conditional Execution Behavior* and *Conditional Execution Context Propagation*. We use the dependences calculated by our analysis presented to build up the dependence graph and slice the model.

5.3 Slicing of Simulink Models

In this section, we present our slicing approach for MATLAB/Simulink models. It is based on the use of dependence graphs for the model to calculate the slice. Hence, it consists of two parts: The construction of the dependence graph and the slicing of the models by performing a reachability analysis.

5.3.1 Building the Dependence Graph

Once we have identified data and control dependences within a given model using our dependence analysis, we can construct a MATLAB/Simulink Dependence Graph (MSDG). We define a MSDG as follows:

Definition 5.6 (MATLAB/Simulink Dependence Graph)**.** *A* MATLAB*/Simulink Dependence Graph is a directed, rooted graph* $G(V,A)$ *where*

(i) the set of nodes V *represent the blocks of the model and region nodes*

(ii) the set A *of ordered pairs* $V \times V$ *that represent the dependence edges.*

Region nodes are nodes that do not actually map to a block (like the entry *node). The* entry *node that cannot be the target of any dependence edge.*

Note that the `Enabled` subsystems internally created for each data input of switch blocks are represented as region nodes in the MSDG, too.

The construction of the MSDG is done in the following steps:

1. In a preprocessing step, all subsystems are flattened. The flattening is done by reassigning the signals connected to inputs and outputs of subsystems to the corresponding `Inport` and `Outport` blocks inside the subsystem.

2. Then, for each block in the model, a corresponding node in the set of nodes V of the MSDG is created.

3. The data dependences are added to the nodes by iterating over all blocks in the model. For each line connecting two blocks, a data dependence edge is added.

4. Finally, the entry node is created and starting with the root EC, control dependence edges are added. Therfore, all ECs are traversed. If a child execution context is found during the traversal, the node corresponding to the predicate block is added to the parent execution context and the child execution context is traversed recursively. At switch blocks a region node is created for every CEC that is not empty and a control dependence edge is added to all child nodes.

The resulting MSDG can now be used for static forward and backward slicing.

5.3.2 Computing the Simulink Slice

The slice for a model is computed using a reachability analysis in forward or backward direction and marking the visited nodes. The starting point for the analysis is defined by the set of blocks $\boldsymbol{B}$ given in the *slicing criterion* $\boldsymbol{C}(\boldsymbol{B})$ (see Definition 5.1). The reachability analysis is done using a work-list algorithm that starts with the slicing criterion and iterates over all outgoing or incoming dependence edges.

Algorithm 5.3 depicts the basic slicing algorithm. The algorithm takes two input parameters: the slicing criterion and a Boolean flag that indicates the direction of the reachability analysis. In a preprocessing step, subsystems within the slicing criterion are replaced by the corresponding port blocks depending on the desired direction. This is necessary since the dependence graph was constructed from the flattened model. However, if a subsystem is part of the slicing criterion it is sufficient to use the underlying `Outport` blocks for forward slicing and the `Inport` blocks for backward slicing. Then, for each block the corresponding node in the MSDG is added to the work-list. Now, the algorithm iterates over the work-list until no new nodes are added to the list. Each node in the list is marked to keep track of already visited nodes. Furthermore, for each node the dependence edges are traversed. Depending on the direction, if the target (*dataOut* or *controlOut*) or source (*dataIn* or *controlIn*) nodes have not been marked yet, they are added to the *addlist* for the next iteration. Note that, by definition, a node is directly control dependent to exactly one node.

Algorithm 5.3: Slicing Algorithm

```
Function slice(blockCriterion, forward:bool) as
    replaceSubsystemsByPortBlocks(criterion, forward);
    worklist = getNodesForBlocks(blockCriterion);
    addlist = $\emptyset$;
    while worklist $\neq \emptyset$ :
        for node in worklist :
            node.mark();
            if forward :
                for n in node.dataOut :
                    if isNotMarked(n) :
                        addlist.add(n);
                for n in node.controlOut :
                    if isNotMarked(n) :
                        addlist.add(n);
            else:
                for n in node.dataIn :
                    if isNotMarked(n) :
                        addlist.add(n);
                if isNotMarked(node.controlIn) :
                    addlist.add(node.controlIn);
        worklist = addlist; addlist = $\emptyset$;
end
```

The traversal stops if no new unmarked nodes are found. All marked nodes belong to the slice for the given criterion.

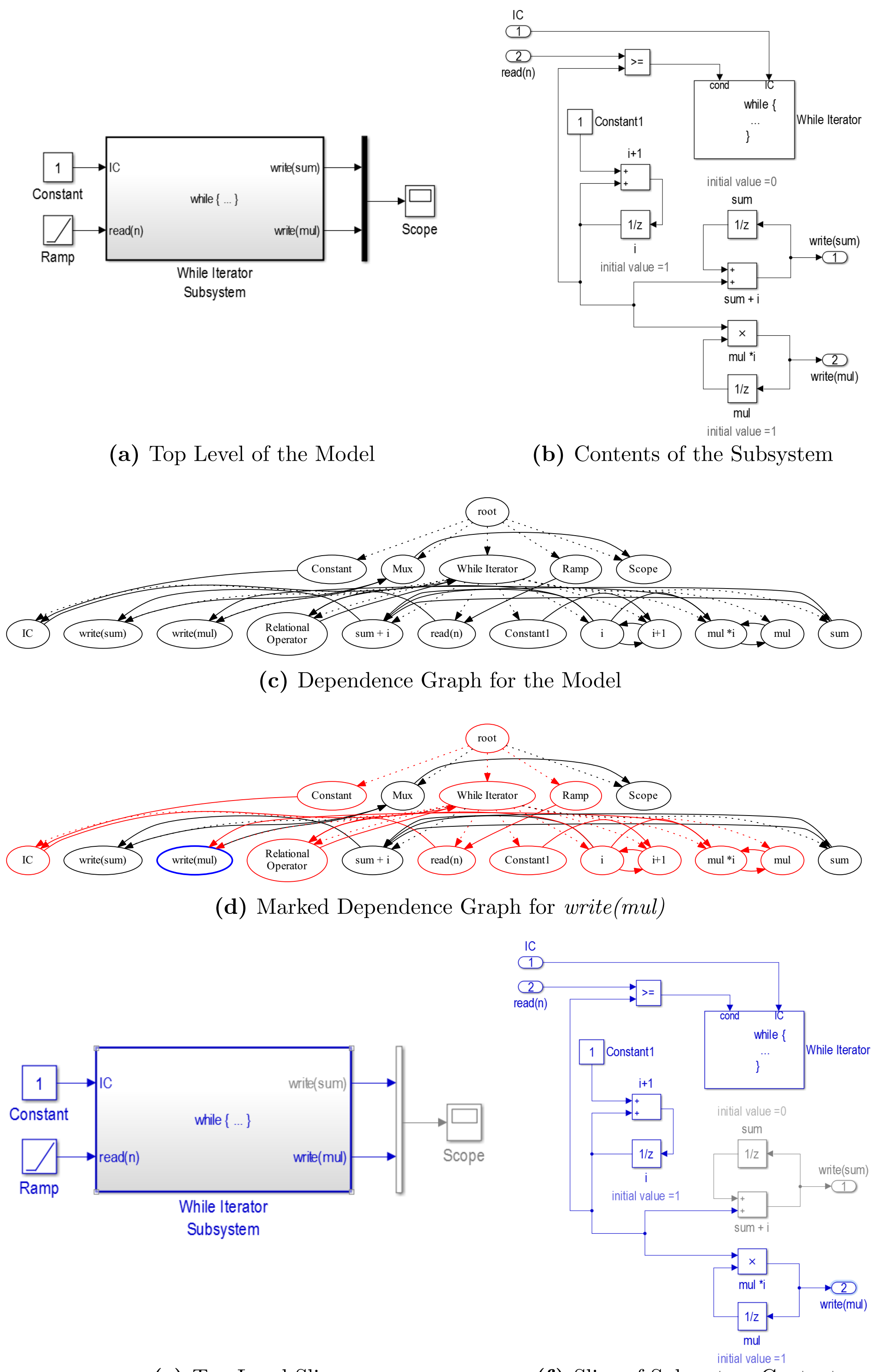
The resulting slice can now be visualized in the model (e. g., with our MeMo tool), or can be used for further processing, e. g., for formal verification. Note that since the MSDG is calculated on the flattened model, there may exist (virtual) subsystems that need to be included into the slice for visualization purposes. This can be easily achieved by a post-processing step, in which every subsystem is checked whether it contains at least one marked block.

5.3.3 Example

Remember the model introduced in Section 2.5 that corresponds to the program in Figure 2.8a. The model is depicted in Figure 5.5a and Figure 5.5b. For a detailed description of the model we refer to Section 2.2.2.

Assume, we want to slice the model for the slicing criterion $C(write(mul))$ backwards. The dependence analysis for the model identifies two execution contexts: the root EC and a CEC for the `WhileIterator` subsystem. The *root EC* contains the blocks

$$EC_{root} = \{\texttt{Constant, Ramp, While Iterator Subsystem, Mux, Scope}\}$$



(a) Top Level of the Model (b) Contents of the Subsystem

(c) Dependence Graph for the Model

(d) Marked Dependence Graph for *write(mul)*

(e) Top Level Slice (f) Slice of Subsystem Contents

Figure 5.5: An Example Simulink Model and its MSDG, the Marked MSDG and the resulting Slice for the Block `write(mul)`

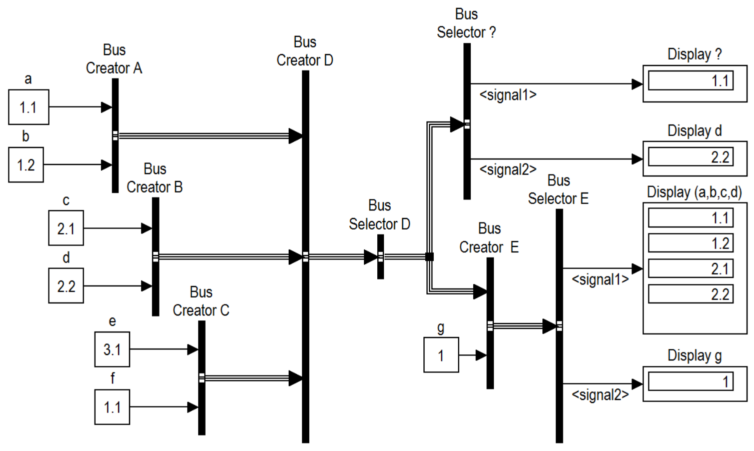


Figure 5.6: Example MATLAB/Simulink model with Bus Systems

and $CEC_{WhileIterator}$ contains the others.

The resulting dependence graph is shown in Figure 5.5c. In this figure, the dotted edges represent control dependence. Solid edges represent data dependence. To slice the model for $C(write(mul))$ the node $write(mul)$ is used as starting point for the reachability analysis.

Figure 5.5d shows the graph after the analysis is finished. The blue node, again, is the starting point. Every red node is reachable directly or transitively in backward direction from $write(mul)$ and belongs to the slice. Black nodes and edges do not belong to the slice.

In Figure 5.5e and 5.5f, we depict the sliced model. In the model, the blocks within the slice are drawn blue while the blocks not contained in the slice are drawn gray. As one can see, neither the block necessary for the calculation of the sum nor the `Mux` and the `Scope` block are contained in the slice since they do not influence the value at block `write(mul)`.

5.4 Increasing the Precision of Slices

In the previous sections we have presented our basic approach for the slicing of MATLAB/Simulink models. This approach is suitable for models taken from a file and to slice it for the blocks of interest.

However, the definition of data dependence is coarse and based on lines. Lines in Simulink represent signal flow between blocks and may contain one or more signals. Multiple signals on a line can either be a vector or a composite signal. Composite signals are usually created by `Mux` or `BusCreator` blocks. A specific signal in a composite signal or vector can be accessed using a selector. Multiple signals are extracted by using `Demux` and `BusSelector` blocks.

Figure 5.6 depicts an example model that contains a number of *Constant* blocks. The values of these *Constant* blocks are passed through a complex

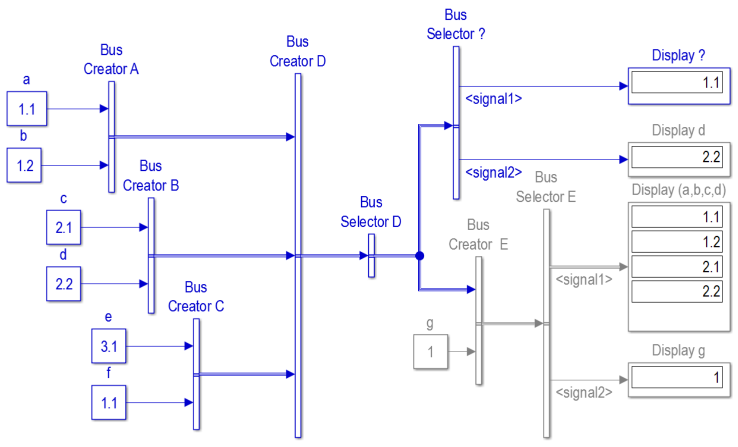


Figure 5.7: Slice for "`Display ?`" (Original Algorithm)

bus system and some of the constants are shown in the `Display` blocks. This model has already been compiled, which causes some lines of the bus system to be visually distinguishable (single and multiple lines) from the rest of the lines. In a model that has not been compiled yet, all lines would look the same (single line) since the inference algorithm for the signals is invoked at initialization time.

The depicted bus system is nested. This means that the bus signal created by the block "`BusCreator A`" is part of the bus signal created by the block "`BusCreator D`". In this example, "`BusSelector D`" extracts some signals from the bus signal and passes them to the block "`BusSelector ?`". This block again extracts some signals and passes them to the `Display` blocks.

By simply looking at the model, it is not possible to find out which of the constants ("`a`" or "`f`") is displayed by "`Display ?`". The slicing algorithm presented in the earlier sections would return all `Constant` blocks since using the original definition for data dependence leads to an over-approximation for bus signals. The slice returned by our algorithm is depicted in Figure 5.7.

Now, we extend our approach in two aspects: Firstly, we add a preprocessing step to our dependence analysis wherein we calculate the data dependences w. r. t. bus systems. Secondly, we extend our slicing algorithm such that it is able to use this information to calculate more precise slices. Note that the preprocessing step requires the model to be executable since it uses information that can only be obtained after the model is compiled.

5.4.1 Routing-aware Dependence Analysis

In our basic slicing approach, we transform the model into an intermediate representation. In that representation, hierarchies are flattened by reassigning the signal lines connected to the inputs and outputs of subsystems directly to the corresponding port blocks. Even though the resulting graph structure is well suited for signal flow analyses, it does not hold the information which signals are transported by a line.

To overcome this problem, we enhance our intermediate representation by annotating each line within a bus-structure with the source signal and the destination signal. However, this information is not explicitly given in a model file. Thus, we need to calculate this information by analyzing the bus system and block parameters.

Bus Systems in Simulink

The main problem in resolving bus systems is that the model file contains no information about the structure of the bus signals. This information is inferred by the MATLAB/Simulink tool while compiling the model in the initialization phase. But, in opposite to the execution contexts of the model, this information can be extracted from the tool. However, a model needs to be in some state after the initialization to extract this information.

To calculate the source and destination of signals within bus systems, we analyze the paths in *backward direction* starting at `BusSelector` blocks to the point where a signal enters the system because `BusCreator` blocks usually do not contain any information about the signals carried by the freshly created composite signal line. In contrast, `BusSelector` blocks have two parameters of interest for this analysis:

- **inputSignals** The `inputSignals` parameter describes a nested array containing the signals supplied to the block. For the "`BusSelector D`" from our example (Figure 5.6), the array would look like:
  `[signal1 [signal1, signal2], signal2 [signal1, signal2],`
  `signal3 [signal1, signal2]]`
- **outputSignals** The `outputSignals` parameter describes an arbitrary selection of these signals. Hereby it is possible to use a dot to select signals through the hierarchy levels of the composite bus signal. A possible selection would be `[signal1, signal3.signal2]`. This would select all signals of `signal1` (in our case, the signal created by "`BusCreator A`" containing two signals) and the second signal of the bus signal created by "`BusCreator C`" (the constant value 1.1).

In addition to `BusCreator` and `BusSelector` blocks, Simulink provides `BusAssignment` blocks that can be used to update specific signals in a bus system. The parameter `AssignedSignals` of this block describes the output signals of the bus to be replaced with new values in the same syntax as in `outputSignals` parameter in `BusSelector` blocks.

Calculating Signal Flow in Bus Systems

The basic idea of our calculation algorithm is to use the information provided by the `BusSelector` blocks and use it as *goals* to reach during traversal. More

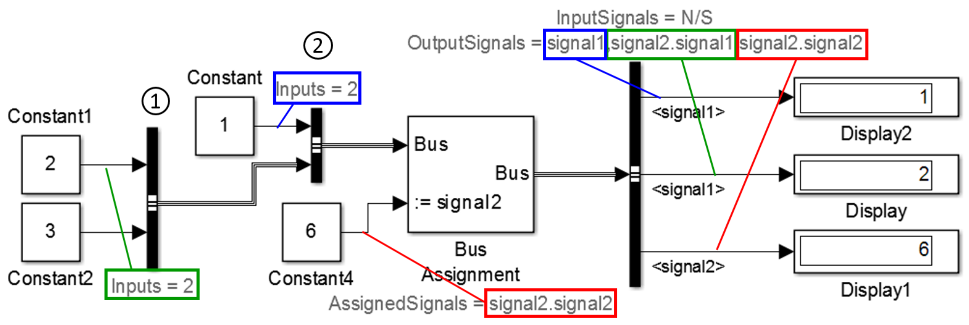


Figure 5.8: Model with nested Bus Systems and a `BusAssigment` block

precisely, we use the entries of the `outputSignals` parameter as our goals, and whenever we encounter a bus creator during traversal, we remove the highest hierarchy level of this bus. With that, we achieve a flattening of bus systems where all bus signals are resolved.

Figure 5.8 shows a small model with a nested bus system and a `BusAs-sigment` block. From the `BusSelector` we derive three goals: `signal1` (blue), `signal2.signal1` (green) and `signal2.signal2` (red). Note that the `In-putSignals` parameter is not available since the model is not in the compiled state and the names of signals are displayed without hierarchy.

The algorithm for the calculation of signal flow in bus systems consists of the following steps:

1. Collect all `BusSelector` blocks in a model. Then, for each `BusSelector` block that has not been processed do steps 2-4:

2. Create a traversal goal from each entry in `outputSignals`. Then for each goal (corresponding to an outgoing signal line of a `BusSelector`) do:

3. Traverse the incoming signal line of the `BusSelector` in backward direction to determine *a path* to the signal line entering the bus system. During the traversal, signals are added to the path, paths are duplicated, or the traversal stops depending on the next block in backward direction:

   - **BusSelector** If the backwards traversal finds a `BusSelector`, the path information for the corresponding output of the block is added to the current path. If there exist more than one path for the output, the current path is duplicated. If the `BusSelector` has not been processed yet, the algorithm is invoked recursively on the `BusSelector` block.

   - **BusAssignment** If the traversal finds a `BusAssigment` block on the path, the algorithm checks if the signals specified in the parameter `AssignedSignals` affect the traversal goals. A goal is affected if an entry of the parameter exactly matches the goal, or share the same signal hierarchy. Hereby, we distinguish between *full match*, *partial match* and *no match*. A full match is given if the parameter exactly matches the goal or higher hierarchy levels of the goal (e. g., the parameter is `signal2` and the goal is `signal2.signal2`). In this case, the traversal is finished since this is the entry point to the bus system w. r. t. data dependence. If the parameter entry and the goal do not match (e. g., the parameter is `signal2` and the goal is `signal1.signal2`), the traversal is continued since none of the signals of interest are affected. In case of a partial match, where the parameter entry is more precise than the goal (e. g., the parameter is `signal2.signal2` and the goal is `signal2`)), the paths have to be duplicated since the `BusAssigment` as well as the blocks found with further traversal are affecting the signal. In our example model from Figure 5.8 we have a full match for the red goal. Hence, the traversal is finished and a path from `Constant4` to the third output of the `BusSelector` is returned. For the blue goal and the green goal, we have no match, and for these goals the traversal is continued.

- **Merge, Switch, MultiPortSwitch** If the traversal encounters one of these blocks, all possible input signals have to be treated as relevant paths except for the control inputs of the `Switch` and `MultiPortSwitch` blocks. Hence, the current path is duplicated and used as prefix for the further traversal of the relevant signal lines connected to the inputs.
- **BusCreator** A `BusCreator` block is an entry point to the bus system. When encountering this block, the algorithm uses the signal structure given by the `inputSignals` parameter of the currently processed `BusSelector` block and the `Inputs` parameter of the `BusCreator` to identify the corresponding inputs. In case of a hierarchical signal, the highest hierarchy level is removed from the goal (e. g., the green goal `signal2.signal1` is changed to `signal1`) and the traversal is continued on the corresponding signal line. If the goal does not describe a hierarchical signal, the traversal is stopped and the path is returned. In Figure 5.8, the traversal for the blue goal is stopped in (2) and for the updated green goal in (1).
- **Sources** Besides `BusCreator` blocks some source blocks are also able to produce bus signals. Furthermore, (top-level) inputs to a model may contain bus signals. In both cases, the bus signals cannot be resolved further, and hence, the traversal is stopped.
- **Other Blocks** If the traversal finds a block of a type not mentioned in the list before, it duplicates the path for every input of the block. However, most of the other *bus-capable* blocks[2] have only one input.

4. Finally, for each path found with the algorithm, the entry and the exit signal are annotated to every line on the path.

[2]http://www.mathworks.com/help/simulink/ug/bus-capable-blocks.html

The enhanced intermediate representation can now be used for the construction of an enhanced dependence graph that enables precise slicing trough bus systems.

5.4.2 Routing-aware Dependence Graph

With the intermediate representation presented in the previous section, we are now able to enhance our slicing algorithm for precise and routing-aware slicing of bus systems. The basic idea is to add additional nodes to the dependence graph to model the paths of the signals through the bus systems explicitly. Instead of creating one node per block, we now create nodes for each signal that is passed through a block within a bus system. That requires some changes to the dependence graph construction and minor changes to the slicing algorithm. The changes to the slicing algorithm only apply if a block within the bus system is taken as slicing criterion.

For the calculation of the routing-aware dependence graph

1. we calculate the data and control dependences and build up the dependence graph, and
2. search for nodes in the dependence graph that are entry points into bus systems.

For each entry point into a bus system, the output signals of the subsequent blocks are traversed. For every block that has at least one output signal that is annotated with the same destination, a special *BusNode* is created. The *BusNode* replaces the previous node for the block and represents one possible path through the bus system. It maintains a reference to the block and inherits all control dependences from the original block. The data dependences are set only for the nodes corresponding to the current path. The traversal stops if one of the output signals of a block equals the desired destination. The data dependence edges of the last *BusNode* in a path are set to the destination node.

With the dependence graph enhanced in this way, we are now able to slice through bus systems using the same reachability analysis as for the original algorithm. Only two modifications are necessary. Firstly, if the slicing criterion is a block within a bus system, all *BusNodes* representing this block in some path have to be added to the slicing criterion. Secondly, the references to the original blocks have to be used in case of a *BusNode* when extracting blocks from the dependence graph.

Figure 5.9 depicts the routing-aware MSDG for the model presented in Figure 5.6. In this MSDG, the *BusNodes* are drawn as boxes. The slicing criterion ("`Display ?`") is the node with the blue border. A backwards reachability analysis starting from this node results in five reachable nodes. Four *BusNodes* and the `Constant` block `a`. The *BusNodes* represent the paths through the bus system and replace the original nodes except for bus selectors where a signal path is not selected. Those paths end in the original node for the `BusSelector` block.

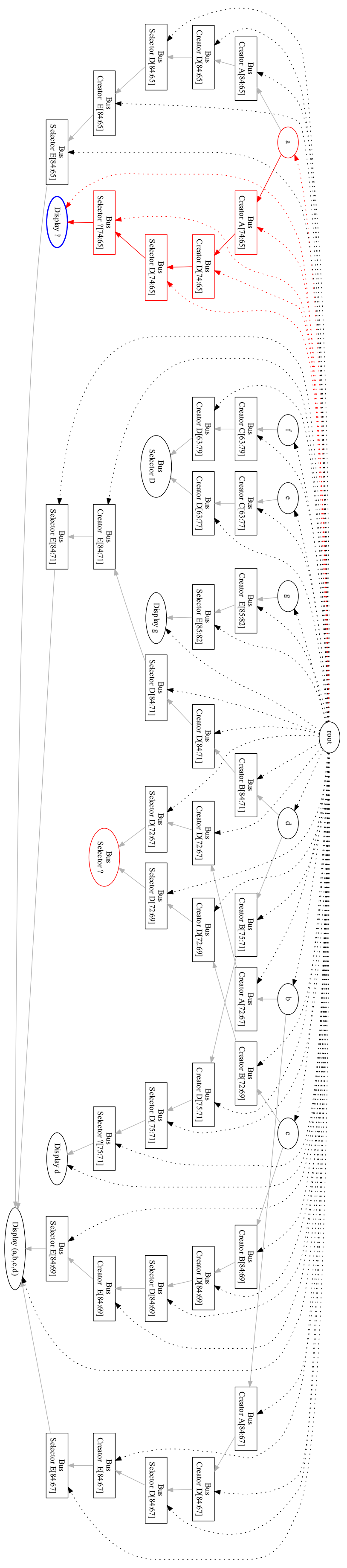


Figure 5.9: Routing-aware MSDG with Slice for “`Display ?`”

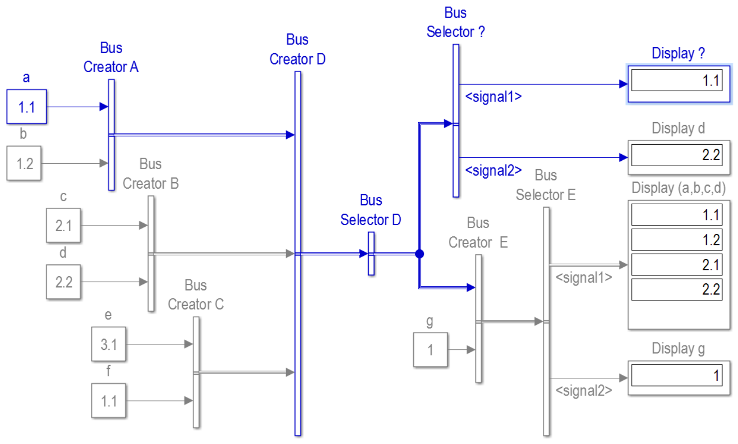


Figure 5.10: Routing-aware Slice for "`Display ?`"

Finally, Figure 5.10 shows the slice (for "`Display ?`") calculated using our improved approach. Now, it is clear that "`Display ?`" shows the value of the `Constant` block "`a`".

With this enhanced slicing approach we are able to calculate considerably more precise slices. In Section 9.2 we provide evaluation results for a comparison of the basic and the routing-aware approach.

5.5 Summary

In this chapter, we have presented our static slicing approach for Matlab/Simulink models. Our approach consists of a dependence analysis for Matlab/Simulink models w. r. t. the simulation semantics and control flow given by the conditional execution of blocks. With the calculated data and control dependences, we construct a Matlab/Simulink Dependence Graph (MSDG), which we use to calculate the slices with a reachability analysis. Besides our basic technique, which is suitable for any Simulink model, we have also presented an enhanced technique that is routing-aware w. r. t. signal flow trough bus systems within a given model. The latter technique is more precise than the basic approach but requires the models to be executable since it uses run-time information of the signals. With these slicing techniques, we are able to reduce a given Simulink model to a partial model, which consist of all blocks that possibly affect or are possibly affected by a given block of interest. Furthermore, the dependence analysis provides information about control and data flow to the transformation approach presented in the next chapter.

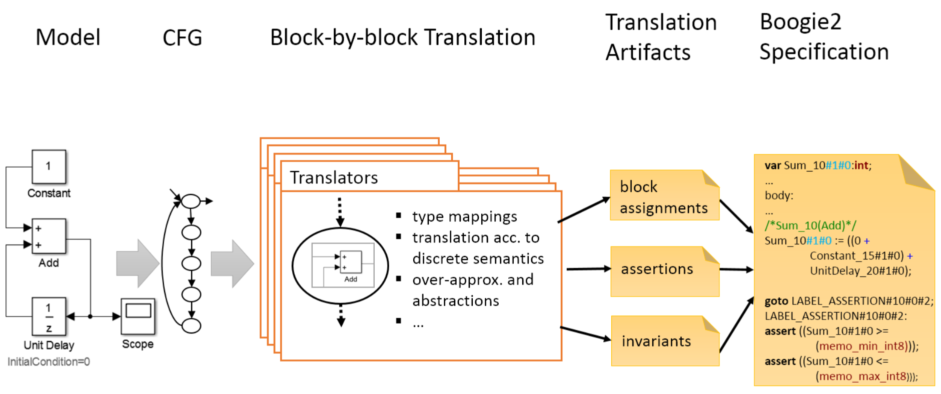
6 Transformation of Discrete-Time Simulink Models to Boogie

In this chapter, we present the core component of our approach: an *automatic* transformation (engine) to *translate* discrete-time Matlab/Simulink models into *formal* Boogie2 specifications that contain the *formal model*, *automatically* generated *verification goals* and *invariants*. The resulting specification enables the mostly automatic verification of Simulink models using the Boogie verification framework.

This chapter is structured as follows: First, we give an overview of the transformation approach in Section 6.1. Afterwards, we discuss the assumptions and limitations to the models suitable for our approach in Section 6.2. In Section 6.3, we introduce the basic concepts of our translation such as the modeling of blocks, signals, control flow and the mapping of the data types in our formal model. Then, we describe our translation in detail. To this end, we first describe in Section 6.4 how we calculate the control flow graph for a given Simulink model before we present the translation rules for each supported block type in Section 6.5. Finally, we describe how we handle implicit casting of data types in Simulink and the saturation of output signals, which may occur for multiple block types before we conclude this chapter in Section 6.6.

6.1 Overview

Matlab/Simulink is widely used in industry for the modeling and development of embedded controller software. Even though it is used for the development of safety critical software components, there exists no formal description of the semantics for the models. However, to verify Simulink models we need to obtain a formal semantics for them. There is a detailed documentation [Mat14b] but it is only an informal description of some but not all behaviors. Thus, it is incomplete and inaccurate (w. r. t. certain model and block behavior).



```
var Sum_10#1#0:int;
...
body:
...
/*Sum_10(Add)*/
Sum_10#1#0 := ((0 +
        Constant_15#1#0) +
        UnitDelay_20#1#0);

goto LABEL_ASSERTION#10#0#2;
LABEL_ASSERTION#10#0#2:
assert ((Sum_10#1#0 >=
         (memo_min_int8)));
assert ((Sum_10#1#0 <=
         (memo_max_int8)));
```

Figure 6.1: The Translation Process

This means that the key challenge for our translation and for later verification is to obtain a formal semantics for Simulink models. In our approach, we obtain a formal semantics by mapping MATLAB/Simulink models to the (formally) well defined Boogie2 language. For the formal model, we use two observations:

(1) The simulation engines of MATLAB/Simulink, also called solvers (see Section 2.2.2), give an informal semantics to the model. Depending on the selected solver, blocks may have different behaviors. In addition, the simulation is performed sequentially in a simulation loop.

(2) Every block type is a language element with a certain behavior depending on the solver, the block parameters and the inputs. Furthermore, MATLAB/Simulink changes over the years. This means that new block types are added and sometimes the behavior of a block changes.

The first observation leads to our first key idea, namely to explicitly model the simulation loop in our formal Boogie2 model. Within this simulation loop, we then place the blocks, which are translated w. r. t. their informal semantics, signal flow dependencies and control flow. To achieve this, we calculate a control flow graph for the model first. We reuse our dependence analysis presented in the previous chapter to calculate data and control dependences (more precisely the execution contexts) and use this information to determine the Control Flow Graph (CFG).

The second observation leads to our second key idea to provide a modular translation approach where for each supported block type, translation rules are defined (and performed by a translator). This set of block translator is easily extensible and maintainable in case of new blocks and modified behavior.

Figure 6.1 depicts the general translation process. First, a CFG is created from the model. Then, we iterate over every node of the CFG that represents a block and translate this block according to the translation rule defined for its type, its inputs and its parameters. Besides the Boogie2 statements that model the block behavior, the translator generates verification goals for certain error classes and loop invariants that are required for the verification of the model.

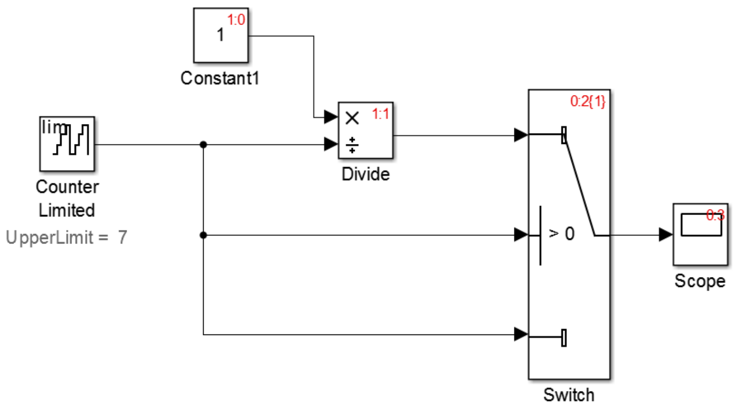


Figure 6.2: Example Model with Control Flow

We present a detailed description of the automatically generated verification goals and the loop invariants in the next chapter (Chapter 7). The resulting artifacts are composed into the formal specification.

Motivation for Using the Sequential Semantics

Before we present our translation in detail, we discuss the use of the sequential simulation semantics. Figure 6.2 depicts a small Simulink model with the execution order and the Conditional Execution Contexts (CECs) displayed in the upper right corner of each block. The model consist of a limited counter that repeatedly counts from zero to seven, which provides the value to a `Switch` and a `Product` block. Depending on the value of the counter, the `Switch` block either outputs the value of the counter (if the counter outputs 0) or the result of the division. With *Conditional Execution Behavior* enabled, the model consists of two execution contexts. A CEC consisting of the `Constant` and the `Product` block that is only executed if the control signal is not zero, and the *root context* that consists of all other blocks. A division-by-zero cannot occur since the `Product` block is only executed if the counter outputs a value that is not zero. However, when using a synchronous semantics that does not consider the execution contexts, the `Product` block is executed if the counter outputs zero, too. A division-by-zero is reported although it cannot occur in the actual model. Hence, our translation using the sequential Simulation semantics provides a more precise formalization of the model w. r. t. control flow.

6.2 Assumptions

Matlab/Simulink is used to model embedded controllers and their possibly continuous environment. Hereby, the controller, more precisely the controller software, is modeled in the discrete time domain. Since the aim of our framework is to verify controller software that is executed on discrete embedded controllers, we focus on the subset of blocks that operate in the discrete time domain and on the semantics of these blocks as given by a *discrete, fixed-step* solver.

Furthermore, we have to make further limitation to the models since MATLAB/Simulink enables the integration of legacy code and provides mathematical functions and operations that cannot be expressed with Boogie2. Hence, we constrain the models by the following limitations:

- **Discrete Subset** The models should only contain blocks from the discrete subset of the block library and stateless blocks, such as the block sets: *Discrete* blocks, *Lookup Tables*, *Math Operations*, *Ports & Subsystems*, *Signal Routing*, *Sources* and *Sinks*
- **Fixed Sample Time** We assume that the sample time is fixed for every block. Note that with our parsing framework, we can also support inherited (inferred) sample times.
- **Limited Support for S-Functions** We do not support external code like C code or MATLAB functions (M-code) introduced by `S-Function` blocks. However, since these blocks are not completely avoidable in practice, these blocks are over-approximated if necessary.
- **Stateflow** We only provide limited support for Stateflow blocks. Since Stateflow models are internally realized by `S-Function` blocks, they are also over-approximated by default. However, the ranges of output signals given by the parameters of Stateflow blocks can be used as assumptions for the verification.
- **Limited Bus Support** While we generally support bus systems, we do not model them explicitly. Instead, we use the algorithm presented in the previous chapter to resolve bus systems directly. We assume that the model does not contain bus-capable blocks and composite data types like bus objects.
- **No Matrix or Complex Signals** We only translate vector and scalar signals and element-wise operations on matrices. We do not support signals that represent complex numbers. Furthermore, we do not support enumerated data types.
- **Limited Support for Arithmetic Operations** There are blocks that represent calculations that are too complex, or for which it is even impossible to define a sufficient mapping with the types and operations provided by Boogie. For example, we cannot model some functions like the exponential function. To cope with these blocks, we use over-approximations of the behavior or of the calculated results.
- **Limited Support for Floats** While Boogie naturally supports integer and Boolean variables, it does not support floating-point types. Hence, we over-approximate them using rational numbers.

- **Limited Support for Simulation Time** Some Simulink blocks use the simulation time to calculate the outputs. For these blocks we use over-approximations since we do not want to explore the full state space of a model starting from the initial state.
- **No Iterator Subsystems** Although, we are generally able to translate loop subsystems with our approach, we do not support models containing loops since the verification may require additional (manually specified) invariants to succeed and they may greatly increase the verification effort. However, we discuss the general translation idea in Section 6.5 to give an intuition how the translation could be done.
- **No Algebraic Loops** We require the models to not contain *algebraic loops*. An algebraic loop is a cycle in the data flow without a block modeling a delay where every block depends directly on the outputs of all other blocks.

Even though our approach has a number of limitations, it is still suitable to be used on industrial case studies. Moreover, many of these limitations are rather minor issues and can be added in future, e. g., adding support for iterator subsystems or bus-capable blocks. Since we use parameters in our approach that are inferred at compile time, we also require the models to be executable. However, our approach still supports a large number of blocks that are commonly used for the development of embedded controllers and hence, it is suitable for the verification of Simulink models for these controllers.

6.3 A Formal Model for Simulink

The key idea for our formal model for MATLAB/Simulink models in Boogie2 is the explicit modeling of the simulation loop. We aim for a formal model that can be automatically generated and automatically verified w. r. t. certain classes of errors using inductive techniques and the Boogie verification framework. Furthermore, we define abstractions to over-approximate the behavior in our formal model to cope with the complexity of the original Simulink model. Some of these abstractions are necessary since Simulink models may use a number of arithmetic operations that are not supported by the Boogie verification framework and the Z3 theorem prover, others are used to reduce the complexity of the formal model and hence, to be able to perform verification fully automatically. Furthermore, we abstract from the actual simulation time to over-approximate the possibly unbounded state space of a given model.

For the abstractions in our formal model, we use safe over-approximations of the actual behavior of the Simulink model. Since these abstractions model more possible executions than the original model, it may result in false negatives during the verification. However, we aim for abstractions where the over-approximation of the behavior is small to reduce the number of false negatives.

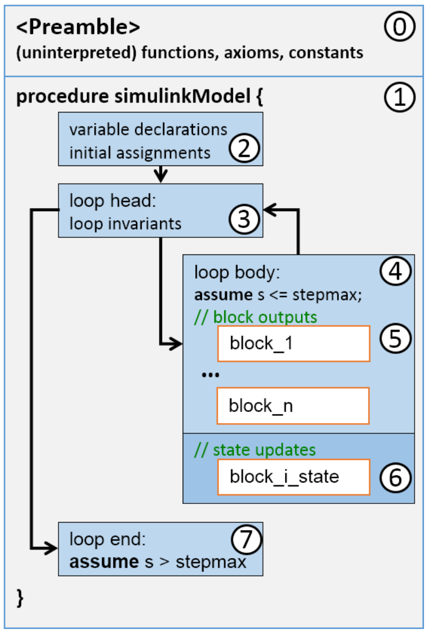


Figure 6.3: The Structure of the Formal Model

Figure 6.3 depicts the (schematic) structure of our formal model. Since Boogie2 is a procedural language, we generate an overall procedure for the model (1). Within the procedure, there is an initialization section (2) and a simulation loop (3-6) corresponding to the initialization and simulation phases presented in Section 2.2.3. In the initialization section (2), the variables representing the blocks are declared, and set to their initial values.

For the simulation loop, we use the desugared representation realized by `goto` statements (see Formula (2.10), Section 2.3.2). The desugared representation consists of a *loop head* (3) where the *loop invariants* are specified, the *loop body* (4) which contains the behavior of the model, and the *loop end* (7) which corresponds to the end of the simulation. The upper execution limit *stepmax* is a symbolic constant that is constrained to be greater than the number of steps for the simulation specified in the model parameters to model a possibly unbounded execution. Within the loop body, there are two sections. The first section corresponds to the first step of the simulation (see Section 2.2.3) where the new outputs for the blocks are calculated (5). The output functions of the blocks are mapped to Boogie2 statements and positioned according to the CFG. The second section corresponds to the second step of the simulation where the states of the blocks are updated (6). Note that since we require the models to have a fixed-step discrete sample time, we can omit the steps for zero-crossing detection and for the calculation of the next step length.

In addition to the procedure corresponding to the Simulink model, our formal model also contains a preamble (0). This preamble mainly serves for two purposes. On the one hand, all constant values for the model are declared and initialized in the preamble. Constant values are either outputs from `Constant` blocks, constant values provided by the MATLAB/Simulink tool[1] or constants for data type bounds. On the other hand, the preamble also contains a number of abstractions or over-approximations using (uninterpreted) functions. Furthermore, operations and functions provided by the underlying Z3 theorem prover that are not part of the Boogie2 language are declared here using the `{:bvbuiltin}` directive.

When abstracting the behavior of blocks in our formal model, we assume that a block (or an intermediate result) has a range of valid values. The block is over-approximated by assuming that it can output any value in the range of valid values. The range of valid values is either given by a minimum and maximum value or linear functions. The actual abstractions are described for each block type in Section 6.5.

In the following sections, we discuss general concepts for our formal model and transformation, before we then present the translations for the specific block types in Section 6.5.

6.3.1 Modeling Blocks

Our basic idea for the translation of blocks is to map the output and (state) update functions of blocks to statements in Boogie2.

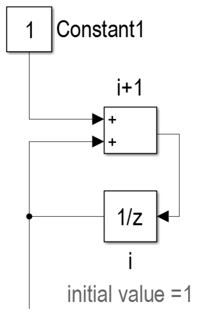
A Simulink block $B = \{I, S, O\}$ generally consists of three (possibly empty) sets that depend on the block type and its parameter configuration. The set of inputs $I = \{in_1, \ldots, in_n\}$ consist of the block inputs where a signal is connected to. Note that $in_i = \{in_{i1}, \ldots, in_{ik}\}$ is a set of signals if the input is a vector. The set of outputs $O = \{out_1, \ldots, out_m\}$ consists of the block outputs where an output signal can be connected to the block and can output a vector of signals analogously. The set of block states $S = \{s_1, \ldots, s_j\}$ are the internal states. Depending on the block type, the sets may be empty (e. g., a `Constant` block has no inputs and no states).

In Simulink, the output and state update functions for a block b are defined as

$$\begin{aligned} O_b = &\quad f_{Out}(I_b, S_b, time) \\ S_b = &\quad s_{Upd}(I_b, S_b, time) \end{aligned}$$

where $time$ is the simulation time used in the calculations of some blocks. We abstract from the actual simulation time in our formal model and over-approximate the behavior of blocks w. r. t. the time. However, many blocks do not use the simulation time and for some other blocks it is not relevant if a fixed-step, discrete solver is used for simulation. Hence, we translate the output function into a set of Boogie2 assignments corresponding to the set of outputs:

[1]Simulink models can access values from the *workspace* of the MATLAB/Simulink tool. These values are loaded into the workspace with configuration files written in M-code or by callback function like *onLoad()* specified in the model.



(a) Excerpt from Figure 5.5

```
// Constant1
$Constant1.out_1$ := 1;

// Sum (i + 1)
$Sum.out_1$ := $Sum.in_1$ + $Sum.in_2$;

// UnitDelay (1/z)
// output assignment:
$UnitDelay.out_1$ := $UnitDelay.state_1$;
// state assignment:
$UnitDelay.state_1$ := $UnitDelay.in_1$;
```

(b) Contents of the Subsystem

Figure 6.4: An Excerpt from the Example Simulink Model (Figure 5.5) and the Output and Update Functions

```
$out_{11}$ := $f_{11}$(I, S);
$out_{12}$ := $f_{12}$(I, S);
...
$out_{mk}$ := $f_{mk}$(I, S);
```

Here, f_{ij} is the function that calculates the output expression for a specific element in the set of outputs for a block. The statements for the state update function are translated analogously:

```
$state_{11}$ := $g_{11}$(I, S);
$state_{12}$ := $g_{12}$(I, S);
...
$state_{mk}$ := $g_{mk}$(I, S);
```

The function g_{11} returns the state update expression. Note that in_{ij}, out_{ij} and $state_{ij}$ are replaced by the corresponding variable names in the actual translation. The mapping for every supported block type is presented in Section 6.5.

Figure 6.4 depicts an excerpt of the example model from Figure 5.5 and the corresponding update and output statements. The three blocks realize a simple counter that increments a value by 1 in every simulation step and is initialized with 0. The `Constant` block has no internal states, no inputs and its output is constantly 1. The `Sum` block only uses its inputs to calculate it outputs, and the `UnitDelay` simply outputs its state. Since the latter is the sole block with a state, it also has an update function that sets the new state to the input value. Note that the two inputs in_1 and in_2 can be replaced by output variables of the `Constant` and the `UnitDelay`. In the next subsection, we define a naming convention for variables in the model.

6.3.2 Modeling Signals

We use variables in Boogie2 to model the signal lines between blocks. More precisely, we map the output signals of all blocks to variables. Therefore, we flatten the hierarchical structure of the model. For every unique signal, one variable is created in the formal model.

To achieve comprehensibility and to enable us to trace the verification results back to the Simulink model, we use the following naming convention for variables representing signals:

```
[blocktype]_[uniqueID]#[portNumber]#[signal-hierarchy-suffix]
```

Since in Simulink blocks may have the same identifiers within different hierarchy levels, blocks are identified by the full path through the hierarchy and the name, or by an unique ID in newer versions of the MATLAB/Simulink tool. Hence, we use a combination of an unique ID and the block type. Furthermore, we encode the output of the block (so-called *port*) into the variable name by using the `portNumber`, which is an one-based index that indicates the output the signal is connected to. Finally, the hierarchy is encoded into the variable name by using a zero-based index indicating the position in every dimension of the signal. One-based and zero-based indicates whether the index 1 specifies the first or the second element of the input vector.

Assume that the `Sum` block from Figure 6.4a has the unique ID 5. Since it has one output and it outputs a scalar value, the resulting variable name is `Sum_5#1#0`. Note that in our formal model, we do not model bus signals explicitly. Instead, we use the information from the analysis presented in Section 5.4.1. With this information, we can directly relate signals that enter the bus systems to the corresponding signals leaving the system.

6.3.3 Block States and Intermediate Results

In our formal model we also map the states of blocks and intermediate results (e. g., variables for predicates of execution contexts) to Boogie2 variables. For these variables we use the same naming conventions as for signals and prefixes. For example, assume the `UnitDelay` block from Figure 6.4a has the id 4. Then the state is modeled with the Boogie2 variable `state_UnitDelay_4#1#0`.

6.3.4 Initialization

To model the initialization phase of a Simulink model, all variables representing an output signal or an internal state are either initialized to a value specified in some block parameter or set to 0 or *false* if no initial value is specified in the Simulink model. In our formal model, we initialize the variables for output values first. Then we then initialize state variables, and finally the variables for intermediate results.

6.3.5 Modeling Control Flow

The basic idea for the realization of control flow in our formal model is to map conditionally executed subsystems to *if-then-else* statements. Since control flow at switches is also realized with conditional (`Enabled`) subsystems, we use *if-then-else* statements to model these, too. For both, the basic idea is to map the contents of the corresponding Conditional Execution Context (CEC) to the *then*-branch. The default behavior of the outputs of conditional subsystems is mapped to the *else*-branch. Note that in case of the internally created conditional subsystems for switches, there is no default behavior defined since they are always expected to select one data input or to throw an error.

In our formal model, we again use `goto` statements corresponding to the desugared version for an *if-then-else* statement as introduced in Formula (2.8) of Section 2.3.2. An arbitrary conditional subsystem S with a corresponding CEC EC_S, a default behavior D_S and a predicate block that introduces a predicate p (e. g., $in_p > 0$), is mapped to:

```
          goto $then_S$, $else_S$;
$then_S$:  assume p; $Transform(EC_S)$; goto $end_S$;
$else_S$:  assume ¬p; $Transform(D_S)$; goto $end_S$;
$end_S$:   ...
```

Here, $T(EC_S)$ is the translation for the contents of the CEC for S and $T(D_S)$ are statements mapping the default behavior as defined in the block parameters of the subsystem.

6.3.6 Data Types

MATLAB/Simulink supports a wide range of data types for signals. Signals can have different types such as *integers* of different bit widths, *single* and *double* floating point values or *Boolean* values. However, the Boogie verification framework and the underlying Z3 theorem prover are limited to mathematical integers, rational numbers and booleans. Hence, we map the Simulink types to the corresponding Boogie types as follows:

Table 6.1: Mapping of data types

MATLAB/Simulink	**Boogie**
boolean	bool
int8, uint8, int16, uint16, ..	int (mathematical integer)
single, double	real (rational numbers)

Table 6.1 shows the mapping of basic data types from MATLAB/Simulink to Boogie. Note that the mapping of *double* and *single* data types is an unsound approximation: Although every floating point value can be represented as a rational number, the arithmetic operations in the theory of reals is not sound for floating point arithmetic due to rounding. However, despite a small number of rounding errors, this approximation is still suitable to detect general design flaws.

Since the *integer* and *real* data type representations in Boogie are unbounded, we keep the bounds of the data types with the automatically created assertions for the over- and underflow checks. This means that over- and underflows are always considered as an error and reported to the user. Note that in MATLAB/Simulink data types for signals can be inherited. This means, that Simulink infers the actual data types for the signals at run-time. To avoid a reimplementation of the inference algorithm of Simulink, we use the `CompiledPortDataTypes` parameter, which is only available in some state after the initialization of the model.

Our translation of MATLAB/Simulink models consists of two phases. In the first phase, we calculate a CFG of the model. For that purpose, we use the execution context calculation algorithm presented in Chapter 5 and calculate the sorted order of the blocks within the Execution Contexts (ECs). Then, we iterate over the CFG and translate every block according to the translation rules for the specific block type. In the following section, we present how we calculate the CFG for a Simulink model.

6.4 Calculation of the Control Flow Graph

In theory, the CFG for a Simulink model can be easily derived from the structure of CECs in a model and the sorted order, in which the MATLAB/Simulink environment executes the blocks. Every CEC corresponds to a branch in the control flow graph where the predicate can be derived from the predicate block, and the *then*-branch can be derived from the sorted blocks within the CEC. The *else*-branch is empty, that means it contains no blocks. However, it may contain some default behavior defined at the conditionally executed subsystem blocks.

The main challenge for the calculation of the CFG for a model is that the sorted order is part of the execution context information, which is not accessible to the user. Hence, we need to calculate the execution order of blocks in order to derive a CFG for the model using our own calculation for the sorted order. Therefore, we first discuss how the MATLAB/Simulink tool suite calculates the sorted order.

6.4.1 How Simulink calculates the Sorted Order

MathWorks specifies the following rules for the calculation of the sorted order in the Simulink documentation:

Definition 6.1 (Rules for Sorting Blocks [Mat14b][2]). *To sort blocks, Simulink uses the following rules:*

(1) If a block drives the direct-feedthrough port of another block, the block must appear in the sorted order ahead of the block that it drives.

This rule ensures that the direct-feedthrough inputs to blocks are valid when Simulink invokes block methods that require current inputs.

(2) Blocks that do not have direct-feedthrough inputs can appear ***anywhere*** *in the sorted order as long as they precede any direct-feedthrough blocks that they drive.*

Placing all blocks that do not have direct-feedthrough ports at the beginning of the sorted order satisfies this rule. This arrangement allows Simulink to ignore these blocks during the sorting process.

Applying these rules results in the sorted order. Blocks without direct-feedthrough ports appear at the beginning of the list ***in no particular order****. These blocks are followed by blocks with direct-feedthrough ports arranged such that they can supply valid inputs to the blocks which they drive.*

Rule (1) means that direct-feedthrough blocks have to be sorted according to their data dependences. Rule (2) means that the data dependences of non-direct-feedthrough blocks can be ignored for those inputs that do not influence the output of the block directly. For example, an `Integrator` block can have up to three inputs. Besides the value to be integrated, it can also have a reset input and an input to specify the initial value. While the input for the value may be a non-direct-feedthrough input depending on the selected integration method, the inputs for reset and the initial value always are direct-feedthrough.

A minor problem with this description is that it is ambiguous. For parallel data dependences multiple block schedules may satisfy the rules. The actual calculation algorithm implemented by the MATLAB/Simulink tool suite is undocumented. Hence, our algorithm for the calculation of the sorted order produces a correct schedule in accordance to the above rules, but does not necessarily produce the same schedules as Simulink does.

6.4.2 Calculating the Sorted Order

To derive the CFG for a model, we calculate the sorted order of the blocks within each EC and CEC. To this end, we have developed an algorithm that

[2]http://de.mathworks.com/help/simulink/ug/controlling-and-displaying-the-sorted-order.html

calculates the sorted order within a (conditional) execution context using the data dependences derived from the dependence analysis. The challenge for our algorithm is to calculate a correct schedule since blocks may also have data dependences to other execution contexts. The calculation of the CFG is required since there is currently no way to obtain the sorted order from the MATLAB/Simulink tool or the model file. To cope with that problem, we annotate all data dependences outside of a certain execution context to the corresponding predicate or region node. As presented in Section 5.3.1, predicate nodes are created during the dependence analysis for blocks that trigger the execution of a CEC. Region nodes are created for ECs that are always executed, i. e., for the root context or atomic subsystems.

For calculating the sorted order, we annotate the dependences to predicate and region nodes using a recursive depth-first traversal of all execution contexts starting with the *root EC*. For every node representing a block found during the traversal of an execution context, the data dependences are annotated to the corresponding predicate or region node if they are not pointing to a block in the same execution context. Whenever the traversal encounters a predicate or region node, the corresponding execution context is analyzed first. Then, the annotated data dependences of the predicate or region node are checked whether they point to a block of the execution context under consideration. If they do not belong to the current execution context under consideration, they are added to the corresponding predicate or region node. With that, we can ensure that data dependences spanning over multiple levels of nested execution contexts can be respected for the scheduling in every level of nesting.

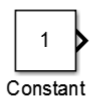
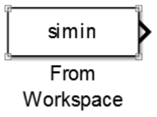
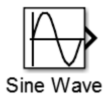
The actual scheduling algorithm then schedules the blocks within each execution context locally. Therefore, we create an empty list and add a predicate or region node if one is available in the current EC. If the execution context does not contain a predicate, an arbitrary block is added. Then, we iterate over the remaining blocks and determine the lowest position (*low*) in the list where all output dependences and the highest position (*high*) in the list where all input dependences are satisfied. Non-direct-feedthrough dependences are ignored. If no dependences are found, the block is added at the end of the list. If we find direct-feedthrough dependences and we have a valid position ($\textit{high} > \textit{low}$), the block is added to the list at *low* position. If we cannot find a valid position ($\textit{low} > \textit{high}$), we remove the sublist from *low* to *high* from the schedule and add the block at its *low* position. The removed sublist is again added to the blocks that have not been sorted yet. These steps are repeated until all blocks are added to the list of sorted blocks.

In the resulting sorted order, (1) is satisfied since blocks are only added to the sorted list if they precede all other direct-feedthrough blocks they drive ($\textit{high} > \textit{low}$). Because we ignore non-direct-feedthrough dependences, (2) is also satisfied. The algorithm terminates, if it has found a valid sorted order and the set of blocks that have not been sorted yet is empty. Note that it only terminates if there exist a valid schedule for the model, i. e., every cycle in data dependence contains at least one non-direct-feedthrough connection. Since we require the models to not contain algebraic loops, this is always the case. Once all execution contexts are sorted, the CFG can be derived straightforwardly and is traversed to translate the blocks.

6.5 Mapping of Simulink Blocks to Boogie2

In this subsection we present our mapping from the supported Simulink blocks into the formal Boogie2 model. The subsection is structured according to the block sets from the Simulink block library. In the following, we focus on basic blocks that are used to realize a number of further blocks from the Simulink block library.

6.5.1 Sources Block Set

Constant	From Workspace	Clock	Sine Wave
			

Our translation currently supports four blocks from the *Sources* block set, namely `Constant`, `Clock`, `FromWorkspace` and `SineWave` blocks. Note that these are basic blocks, which are used to model many other blocks from the *Sources* block set.

Constant

The `Constant` block outputs a constant value at every simulation step. It is part of nearly every composite block. The value is given by the `value` parameter and is either a value or a variable name for a variable stored in the workspace of the MATLAB/Simulink tool or in the block parameters of a surrounding composite block.

The constant block is directly translated to Boogie2 by creating constants for each output signal according to the data types of the signal:

```
const Constant_ID#1#[signal-hierarchy-suffix] : mapped_type;
axiom Constant_ID#1#[signal-hierarchy-suffix] ==
    value[signal-hierarchy];
```

Here, `mapped_type` is the Boogie type according to the mappings from Section 6.3.6 and `value[signal-hierarchy]` is the extracted value from the `value` parameter corresponding to the structure of the signal specified by `signal-hierarchy`.

Clock

The Clock block outputs the current simulation time. In our formal model, we abstract from the simulation time since we do not want to explore the entire state space of a model. Hence, the Clock block is over-approximated by assuming that its output signal has a positive real value:

```
havoc Clock_ID#1#[signal-hierarchy-suffix]; // set the block to an arbitrary
    value
assume 0 <= Clock_ID#1#[signal-hierarchy-suffix];
```

FromWorkspace

The FromWorkspace block takes a variable from the workspace and outputs its values during simulation. These blocks are among others used to realize SignalBuilder blocks that provide models with simulation inputs. The values to output during simulation are specified with the VariableName parameter. In our translation, we only support variables that are structures containing arrays where one array specifies some simulation time instants (*breakpoints*), and all other (*value*) arrays specify the values of the signals at these breakpoints. Depending on the InterpolateData parameter, values of signals are linearly interpolated between two corresponding entries in the value array for time instants between the breakpoints or set to the most recent value. Furthermore, the parameter OutputAfterFinalValue specifies the behavior when the simulation time advances past the last specified breakpoint. Depending on this parameter, the signal is either set to the value given by the last breakpoint (1), is repeated cyclically (2), is set to 0 (3), or is linearly extrapolated (4).

In our translation, we currently over-approximate the outputs of FromWorkspace blocks since we do not model the simulation time explicitly. We iterate over each value array to determine the minimal and maximal value for each signal. If OutputAfterFinalValue is set to (1) or (2) we can assume that the output value is always between the minimal and maximal value. If it is set to (3), for the output always holds $min(min,0) <= out <= max(0,max)$. For (4), we need to calculate the slope between the values for the last two breakpoints. If the slope is negative, we can assume that $out \leq max$. Since signals are bound by their type, we assume that $min == type.min - 1$ instead. This is done analogously for a positive slope.

For the final translation, we over-approximate the FromWorkspace block as follows:

```
havoc FromWorkspace_ID#1#[signal-hierarchy-suffix]; // set the block to an
    arbitrary value
assume min <= FromWorkspace_ID#1#[signal-hierarchy-suffix] &&
    FromWorkspace_ID#1#[signal-hierarchy-suffix] <= max;
```

Here, we set the output value for a signal to an arbitrary value and constrain the output value to the previously calculated minimum and maximum.

Sine Wave

The `Sin` block outputs a sine curve

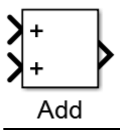
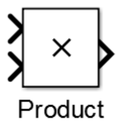
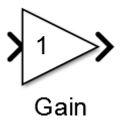
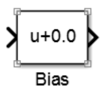
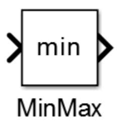
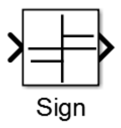
$$out = \texttt{Amplitude} * sin(\texttt{Frequency} * time * \texttt{Phase}) + \texttt{Bias}$$

where the amplitude, the frequency, the phase and the bias are specified with parameters. Since Boogie does not support trigonometric functions and we do not model the simulation time explicitly, we over-approximate this block as follows:

```
havoc SineWave_ID#1#0; // set the block to an arbitrary value
assume -1 * Amplitude + Bias <=
    SineWave_ID#1#[signal-hierarchy-suffix] &&
    SineWave_ID#1#[signal-hierarchy-suffix] <= Amplitude + Bias;
```

Since the sine function outputs a value in the interval $[-1, 1]$ we can assume that for the output signal always holds: $Amplitude * (-1) + Bias \leq out \leq Amplitude * 1 + Bias$.

6.5.2 Math Operations Block Set

Sum	Product	Gain	Bias	MinMax	Signum
					

In this subsection, we present the rules for the basic blocks representing mathematical operations in Simulink, which are currently supported by our translation. We support `Sum`, `Product`, `Gain` and `Bias` blocks, which can be directly mapped to arithmetic operations in Boogie2. Furthermore, we present translation rules for the `MinMax` block and the `Signum` block, which we map to functions in our formal model. All these block are direct-feedthrough blocks.

Sum

The `Sum` block can be used to add or subtract input signals. Its most important parameter is the `Inputs` parameter, which specifies the operations to be performed by the block. For each input of the block either a "+" for addition or a "–" for subtraction has to be specified[3]. Each "+" or "–" operation is applied to the corresponding input, e. g., the first "+" or "–" corresponds to the input with the port number 1, the second corresponds to the port number 2, etc. If the inputs of the block non-scalar signals, they need to have the same dimensions. Scalar signals are expanded if necessary (e. g., 1 is expanded to $(1, 1, 1)$

[3]It is also possible to specify the inputs by a number. Then, all inputs are treated as "+".

if another signal has the width of 3). Furthermore, if only one (non-scalar) input is specified, all vector signals are added (*sum-over mode*) up into one scalar output.

Let $in_1 \dots in_n$ be the inputs of width k. For each signal at position j in the output vector with $0 \leq j < k$, the translation into Boogie2 is the following:

```
Sum_ID#1#j := 0 op[0] $in_{1j}$ op[1] ... op[n] $in_{nj}$;
```

Note that `op` is an array containing the operations specified in the `Inputs` parameter and 0 is the neutral element for the addition. In sum-over mode, for a non-scalar input[4] of size k the translation is defined as:

```
Sum_ID#1#0 := 0 + $in_{10}$ + ... + $in_{1(k-1)}$; // portNumber 1, position 0 since
                                             // the output is a scalar signal
```

Note that both code samples are templates. In the translation for some specific model, the inputs in_{ij} are replaced by the outputs of those blocks that are connected to the corresponding input.

Product

The `Product` block calculates a scalar or non-scalar product from its input signals. The type of the calculation is given by the `Multiplication` parameter and is either set to "`Matrix(*)`" or "`Element-wise(.*)`". Since we currently do not support matrices in our translation, we only translate blocks with the parameter set to the latter. Analogously to the `Sum` block, the `Product` block has an `Inputs` parameter that specifies the operation ("`*`" for multiplication or "`/`" for division), expands scalar inputs if necessary and has a *multiply-over mode*. The translation for some inputs $in_1 \dots in_n$ of width k is defined analogously to the `Sum` block:

```
Product_ID#1#j := 1 op[0] $in_{1j}$ op[1] ... op[n] $in_{nj}$;
```

Note that the `op` array contains the operators for multiplication and division and the neutral element is 1. For the multiply-over mode, the translation is defined as:

```
Product_ID#1#0 := 1 * $in_{10}$ * ... * $in_{1(k-1)}$;
```

Gain

The `Gain` block multiplies the signal connected to its input with a value or expression given by the `Gain` parameter. Like the `Product` it is also capable to process matrices, but in our translation we only support the `Multiplication` method "`Element-wise(K.*u)`". If the `Gain` parameter specifies an expression,

[4]Since we do not support matrices in our translation, we do not need to evaluate the `Sum Over` and `Dimension` parameters that are used to calculate the sum over specific dimensions of matrices.

it is translated to Boogie2 as long as it only uses operations supported by Boogie2, the Z3 or our formal model. For its sole input in_1 of width k, the block is translated as follows:

```
Gain_ID#1#j := (expr) * in1j;
```

Here, `expr` is either a literal value or a previously transformed expression and $0 <= j < k$.

Bias

The `Bias` block add a value given by the `Bias` parameter to the signal connected to its input. Like for the `Gain` block, expressions in the `Bias` parameter are translated to Boogie2 as long as it only uses supported operations. For is sole input in_1 of width k, the `Bias` block is translated as follows:

```
Bias_ID#1#j := (expr) + in1j;
```

Again, `expr` is either a literal value or a previously transformed expression and $0 <= j < k$.

MinMax

The `MinMax` block calculates the minimum or maximum of the signals supplied to its inputs. If the input is a vector, the output is the minimum or maximum over the vector elements. Scalar values are expanded if necessary. The function to be performed is defined by the `Function` parameter and is either "`min`" or "`max`". Since these functions are also part of the MATLAB/Simulink expression language, we have defined these functions in the preamble of our formal model:

```
function {:bvbuiltin "if"} builtin_if_real(x1:bool, x2: real, x3:real)
    returns (real);
function min_real(x:real, y:real) returns (real);
axiom(forall x,y:real :: {min_real(x,y)} min_real(x,y) ==
    builtin_if_real(y >= x, x, y));
function max_real(x:real, y:real) returns (real);
axiom(forall x,y:real :: {max_real(x,y)} max_real(x,y)==
    builtin_if_real(y <= x,x,y));
```

Note that we use the *if* function of the Z3 theorem prover. Since Boogie2 requires functions to be typed, the depicted sample is the formalization for the real type. The formalization for integer types is done analogously. Note that this formalization can be also used to translate expressions to Boogie. For every vector element j in the input signals, the corresponding output is calculated as follows:

```
MinMax_ID#1#j := fun(in1j, fun( ... , fun(in(n-1)j, innj)));
```

Here, `fun` is the function name (e. g., "`min_real`") derived from the value of the `Function` parameter and the types of the input signals.

Signum

The `Signum` block indicates the sign of a signal (element-wise). It returns 1 for a positive signal, -1 for a negative signal, and 0 for a signal that is exactly 0. Again, this block represents a function (*sign*) from the MATLAB/Simulink expression language. Hence, we have formalized the function in the preamble:

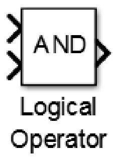
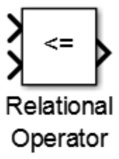
```
function signum_real(x:real) returns (real);
axiom(forall x:real :: {signum_real(x)} signum_real(x) ==
    builtin_if_real( x > 0.0, 1.0, builtin_if_real (x < 0.0, -1.0,x)));
```

Using this formalization, we translate the block for each vector element j into

```
Signum_ID#1#j := fun(in_1j);
```

Note that `fun` is derived from the type of the input signal (e. g., `signum_real` for a real type).

6.5.3 Logic and Bit Operations

Logical Operator	RelationalOperator
	

In this subsection we present the translation of `Logic` and `RelationalOperator` blocks, which are basic blocks used in many of the other composite blocks within this library.

Logical Operator

The `Logic` block applies a propositional logics operation to its inputs. The operation is specified with the `Operator` parameter that is either “`AND`”, “`OR`”, “`NAND`”, “`NOR`”, “`XOR`”, “`NXOR`” or “`NOT`”. The block accepts scalar and vector inputs of the same size and performs the specified operation on each vector element. If necessary, scalar inputs are expanded and, if the “`NOT`” operation is selected, the block supports only one input. Since Boogie2 supports the Boolean operations $\neg$, $\wedge$ and $\vee$, the translation for “`AND`”, “`OR`”, “`NAND`”, “`NOR`” and “`NOT`” is straightforward. For example, if the operation is “`AND`”, then for each element j in the vector for the inputs $in_1 \dots in_n$ the translation is as follows:

```
Logic_ID#1#j := in_1j && ... && in_nj;
```

For the translation of “`XOR`” we use a function provided by the Z3 solver:

```
function{:bvbuiltin "xor"} bxor(bool,bool) returns (bool);
Logic_ID#1#j := bxor(in_{1j}, bxor( ... , bxor(in_{(n-1)j}, in_{nj})));
```

For “NAND”, “NOR” and “NXOR” the whole expression is negated.

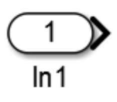
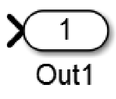
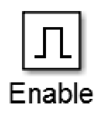
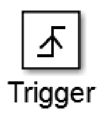
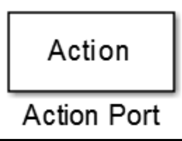
Relational Operator

The RelationalOperator block compares two input signals using a relational operation specified in the Operator parameter. Possible values for the parameter are “==”, “∼=” ($\neq$), “<”, “>”, “<=” and “>=”. These operations are supported by Boogie whereby “∼=” translates to “!=”. The block accepts two inputs of the same size where scalar inputs are expanded to the size of the vector input. For each element j in the output vector, the translation for the RelationalOperator block is the following:

```
RelationalOperator_ID#1#j := in_{1j} op in_{2j};
```

Here, op is taken from the Operator and “∼=” is mapped to “!=”. Note that the RelationalOperator block also has a *one-input mode* where the block checks some properties of floating-point values. These properties are not modeled since we over-approximate floating point types by the *real* type in Boogie. Hence, we currently do not support this mode in our translation.

6.5.4 Ports & Subsystems Block Set

Inport	Outport	Enable	Trigger	Action
				

In this subsection, we present the translations for the following blocks from the *Ports & Subsystems* block set. This block set contains Inport and Outport blocks, all the predicate blocks (e. g., the EnablePort block), and virtual, atomic and conditionally executed subsystems.

Inport and Outport Blocks

Inport and Outport blocks are used to model signal flow through (sub)system boundaries. Although those are virtual blocks, we model them explicitly for two reasons: Firstly, since we flatten all subsystems in the models (see Section 5.3.1), we use the Outport blocks of conditionally executed subsystems to model the default behavior. Secondly, we also need to model inputs from and outputs to the environment. In our flattened model, we treat the inputs and outputs of subsystems as inputs and outputs of the corresponding port blocks. This means, a port block has only one input and output where one of them is inherited from the surrounding subsystem. The signal can either be a scalar or a vector of signals.

Basically, `Inport` and `Outport` blocks are translated into Boogie2 statements that forward the signal value: For every element j in the input vector $in1$, the translation is as follows:

```
Inport_ID#1#j := in_1j;
Outport_ID#1#j := in_1j;
```

`Inport` blocks on the highest hierarchy level, which model an input from the environment, are over-approximated as follows:

```
havoc Inport_ID#1#j;
assume type_min <= Inport_ID#1#j && Inport_ID#1#j <= type_max;
```

In other words, the variable is set to an arbitrary value and we assume that the value is within the bounds (`type_min` and `type_max`) of the data type of the signal. `Outport` blocks on the highest hierarchy level are treated as sinks.

Predicate Blocks and Conditionally Executed Subsystems

As discussed in Section 6.3.5, we model the control flow introduced by conditionally executed subsystems by translating the corresponding CEC directly to Boogie2. Hence, when encountering a predicate block during translation, we perform three steps: (1) we translate the predicate block, and (2) we create a Boogie2 block (*then*-branch) in which the contents of the CEC are translated, and (3) we create a Boogie2 block (*else*-branch) for the default behavior of the system. Since (2) is explained in Section 6.3.5, we focus on (1) and (3) in this subsection.

Our basic idea for the translation of predicate blocks is to model them as Boolean variables. The Boolean variables are used as predicate for the *then*- and *else*-branch. Like `Inport` blocks, predicate blocks have no inputs. Hence, we use the inputs corresponding to the surrounding subsystem, which can either be a scalar or an array of signals. Furthermore, predicate ports can output information about their incoming signals (connected to the predicate input of the surrounding subsystem). However, we currently do not support this option in our translation.

The translation for predicate blocks works as follows: An `Enabled` subsystem is executed if at least one signal connected to the enable port is greater than zero. Hence, an `Enable` block with the input in of width k is translated into:

```
Enable_ID#1#0 := in_0 > 0 || ... || in_k-1 > 0;
```

Here, in_0 to in_{k-1} are the elements of the input vector. `Enable_ID#1#0` is then used as predicate for the *then*- and *else*-branch.

`Action` ports as used in `Action` subsystems are internally realized by signals of type *action*, we map this type to Boolean values. Since `Action` subsystems accept only scalar inputs, the translation for an action port is straightforward:

```
Action_ID#1#0 := in_0;
```

`Triggered` subsystems are realized using `Trigger` port blocks that can be configured to execute on a rising, on a falling, or on both flanks of a signal. Additionally they can be configured to accept signals of the type *function-call*. However, to detect changes in the flanks of signal in a discrete simulation, up to three simulation steps may be required. Since we cannot directly model that for inductive invariant checking, we over-approximate `Trigger` blocks by

```
havoc Trigger_ID#1#0; // arbitrary boolean value
```

Note that we always assume that both, the *then*- and the *else*-branch are executable.

To create the *else*-branch (3), we first collect all `Outport` blocks of the surrounding subsystem. For each `Outport` block, we extract the `InitialOutput` and `OutputWhenDisabled` parameters that are only available if the `Outport` block corresponds to an output of a conditionally executed subsystem. If the `OutputWhenDisabled` parameter is set to `reset`, we translate the block in the else branch into:

```
Outport_ID#1#j := InitialOutput[j];
```

`InitialOutput[j]` contains the initial value for the *j*th element of the output signal. If `InitialOutput` is a scalar value, all vector elements are set to this value.

If the `OutputWhenDisabled` parameter is set to “`held`”, the subsystem outputs the value calculated in the last execution until it is executed again. Hence, we need to create an additional variable for this block:

```
state_Outport_ID#1#j : type;
```

Here, `type` is the same data type as `Outport_ID#1#j`. The state variable is set to the following in the initialization phase:

```
state_Outport_ID#1#j := InitialOutput[j];
```

Furthermore, a statement is added to the *then* branch to update the state when executing the subsystem:

```
state_Outport_ID#1#j := Outport_ID#1#j;
```

In the *else* branch, the block is translated to the statement:

```
Outport_ID#1#j := state_Outport_ID#1#j;
```

In other words, the output is set to the value saved in an internal state variable.

For example, a `Enable` port block with a scalar input to the predicate port (in_p) of the surrounding subsystem that configured to hold the state for a scalar output signal (`Outport_ID#1#0`) we generate the following statements:

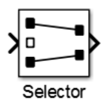
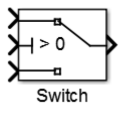
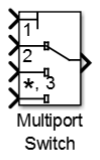
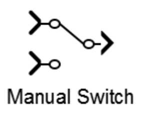
```
Enable_ID#1#0 := in_p > 0;
goto EnablePort_ID_then_branch, EnablePort_ID_else_branch;
..
EnablePort_ID_then_branch:
assume Enable_ID#1#0;
// block translations for corresponding conditional execution context
...
Outport_ID#1#0 := ... ;
state_Outport_ID#1#0 := Outport_ID#1#0;
goto EnablePort_ID_end;

EnablePort_ID_else_branch: // default case
assume !Enable_ID#1#0;
Outport_ID#1#0 := state_Outport_ID#1#0;
goto EnablePort_ID_end;
```

Note that *then*-branch also contains the statements for the execution context corresponding to the `Enable` block.

Loop Subsystems*

Although we currently do not yet support loop subsystems in our implementation, we give a sketch for the translation idea in this paragraph. Besides the `ForEach` subsystem whose number of executions is limited by the width of the incoming signal, loop subsystems are executed as long as a given predicate is satisfied or up to an upper bound (that may be infinite). Both are given by the corresponding iterator block. All executions of a loop subsystem are performed in one simulation step of the outer simulation loop. Furthermore, within the loop subsystem, block states may change in every loop execution and are initialized at the first execution. This behavior is basically the same as for the simulation loop of a model. Hence, it is possible to translate loop subsystems into a separate, nested simulation loop within the actual simulation loop. However, this imposes several problems for the later verification: If there is no finite upper bound for the loop subsystems, we cannot guarantee termination of the loop subsystem. Furthermore, to verify the nested loop, we may require additional manually specified loop invariants, which in most cases decreases the degree of automation in our approach. Finally, when using k-inductive invariant verification, we need to unroll each inner loop k times in each unrolling of the surrounding (simulation) loop. This increases the size of the verification condition exponentially in the size of k and the nesting depth. Since none of our case studies contains loop subsystems, we have chosen to currently not support loop subsystems in our translation of Simulink models for the above reasons.

6.5.5 Signal Routing Block Set

Mux	Demux	Selector	Switch	MultiPortSwitch	ManualSwitch
					

In this subsection, we present the translations for the blocks from the *Signal Routing* block set that are depicted in the above table.

Mux and Demux

`Mux` and `Demux` blocks are virtual blocks, which are used to manipulate vector signals. The `Mux` block creates a vector signal from its input signals and the `Demux` block decomposes a vector signal into a number of output signals. Although it is possible for these blocks to operate on bus signals, we do not support this option (`BusSelectionMode`). Mathworks also discourages the use of this operation mode. For `Mux` blocks, the parameter `Inputs` and for `Demux` blocks the parameter `Outputs` specifies how the vector signals are constructed or split. However, for our translation we do not use these parameters. Instead, we exploit the fact that the order given by the input signals is preserved while composing and splitting the vector signals and use the information about the signal widths given by the `CompiledPortWidths` parameter. A `Mux` with the input signals $in = \{in_1 \dots in_n\}$ is translated to the following:

```
Mux_ID#1#i := $in_{jk}$;
```

Here, j indicates the jth input and k is the kth signal in in_j. Then i is calculated with $i = \left(\sum_{m=1}^{m<j} w(in_m)\right) + k$ where $w()$ is a function returning the width of a signal. Analogously, a `Demux` block with n outputs is translated as follows:

```
Demux_ID#i#j := $in_{1k}$;
```

In this statement, i with $0 < i \leq n$ is the number of the output, j is the position in the output vector, and $k = \left(\sum_{m=1}^{m\leq i} w(out_m)\right) + j$.

Selector

The `Selector` block is used to extract some subset of a composite signal. The input signal can either be a vector or a multidimensional signal. The `NumberOfDimensions` parameter specifies the number of dimensions of the input signal, and the `IndexOptions` parameter specifies which signals are selected for each dimension. While the `Selector` block supports that signals to select can be supplied by an additional port, we do not support these operation modes since they may result in output signals of variable size. Hence, we only translate blocks with the `IndexOption` set to "`Select all`", "`Index vector (dialog)`" or "`Starting index (dialog)`". For the latter two, the signal indices to select are stored in the parameter `IndexParamArray`. Furthermore, the `IndexMode` parameter specifies whether the indices are given one-based or zero-based (whether the indexing starts wit 0 or 1) and `OutputSizes` specifies the widths of the output signal. To translate the block, we iterate over each input dimension. If the `IndexOption` is set to "`Select all`" for a dimension j, and i is the ith element in the input vector, we create the following statement:

```
Selector_ID#1#j#i := in_1ji;
```

If the `IndexOption` is set to "`Index vector (dialog)`", the parameter `IndexParamArray` specifies an array of indices $idx[]$. The translation for a dimension j and an output vector element i is the following:

```
Selector_ID#1#j#i := vec[idx[i-o]];
```

Note that `vec[idx[i-o]]` selects the vector element specified by the ith value in the idx array with respect to an offset $o = 1$ for the *zero-based* index mode and $o = 0$ otherwise. If the `IndexOption` is set to "`Starting index (dialog)`", the parameter `IndexParamArray` only specifies a start index idx. The translation for a dimension j and an output vector element i is the following:

```
Selector_ID#1#j#i := vec[idx + i - o]];
```

The offset o is again corresponding to the `IndexMode` parameter.

Switches

The `Switch` block forwards one of its two data inputs (the first and third inputs) depending on a predicate at its control input (the second input). The predicate is specified by the `Criteria` parameter and is either "`u2~=0`" or "`u2 < Threshold`" or "`u2 <= Threshold`" where "`u2`" represents the control input and the `Threshold` parameter specifies a value to compare with. We translate the `Switch` block analogously to conditional subsystems by creating an extra Boolean variable for the predicate, which is one of the following three statements:

```
predicate_Switch_ID#1#0 := in_2 != 0 ;
```

```
predicate_Switch_ID#1#0 := in_2 < Threshold;
```

```
predicate_Switch_ID#1#0 := in_2 <= Threshold;
```

This predicate variable is then used for the CECs as presented in Section 6.3.5. Note that we initialize the predicate variable with its predicate expression to obtain initial state that is consistent with the model.

The `MultiPortSwitch` block forwards one of its data inputs depending on the signal connected to its control input (first input). The `DataPortOrder` specifies the selection mode of the data inputs and is either "`Zero-based contiguous`", "`one-based contiguous`", or "`Specify indices`". In the first two cases, the indices are integer numbers starting from zero (or one if one-based) up to the number of data ports. In the last case, the user can specify the index values manually in the parameter `DataPortIndices`. If the control signal matches one of the index values, the corresponding data port is forwarded. Furthermore, the default case (marked with a `*`) is specified with the `DataPortForDefault` parameter and either the last data input in the index array or a dedicated input. However, it is always the last data input.

The translation of the `MultiPortSwitch` block is similar to the translation of the `Switch` block except that we do not use a dedicated variable for the predicate. Instead, we use the predicate directly in the assume statement for each branch.

```
goto MultiPortSwitch_ID_data_1, ..., MultiPortSwitch_ID_data_n;
..
MultiPortSwitch_ID_data_i:
assume $in_i$ == idx[i];
...
MultiPortSwitch_ID#1#0 := $in_i$ ;
goto MultiPortSwitch_ID_end;
...
MultiPortSwitch_ID_data_n: // default case
assume !($in_2$ == idx[2]) && .. && !($in_{n-1}$ == idx[n-1]);
...
goto MultiPortSwitch_ID_end;
```

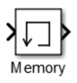
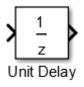
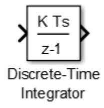
Here, `idx` maps the number of a data input to its corresponding index entry according to the `DataPortOrder` parameter and n is the number of the data inputs including the possibly extra default port. For the default case, we assume that none of the previous index values has been selected. Note that we currently do not support the *index vector* mode, where the `MultiPortSwitch` block selects an element from a vector signal connected to the first (and only) data port. Support for this mode can be easily added in the future but was not necessary for our case studies.

Besides `Switch` and `MultiportSwitch` blocks, Simulink also provides the `ManualSwitch` block type. This block forwards one of its two input signals depending on its internal state. The internal state is changed by a mouse click on the block, which can be used to model sensor failures. We translate such blocks similar to `Switch` blocks but without a predicate. With that, we over-approximate the behavior by assuming that both inputs can be forwarded in any simulation step.

Goto, From and Bus Systems

`Goto` and `From` blocks are mainly used to avoid the crossing of signals lines. A signal that ends in a `Goto` block with a certain label can start in a `From` block with the same label. While parsing the model, we analyze it for such blocks and resolve them by connecting those with *virtual* signal lines that are only present in our representation. Hence, we can translate these blocks like port blocks. Furthermore, we use the information gained by our analysis for bus systems (Section 5.4.1) and do not model the bus systems explicitly. Instead, we connect blocks at the beginning and at the end of a bus system directly in our translation.

6.5.6 Discrete Block Set

Unit Delay	Memory	Discrete Integrator
		

In this subsection, we present the translation of `UnitDelay`, `Memory` and `DiscreteIntegrator` blocks. These are basic blocks from the *Discrete* block set and used in many other composite blocks.

Memory and Unit Delay

The `Memory` block stores the value of the signal connected to its input for one major[5] simulation step. The `UnitDelay` stores a value for one sampling step specified in its sample time. In our translation, we currently only support `UnitDelay` blocks with the same sampling time as the model. In a fixed-step discrete simulation, every simulation step is a major step (where the model is sampled). Hence, both blocks have the same behavior and we explain the translation using the `UnitDelay` block.

For the translation of both block types, we create output and state variables corresponding to the input signal, which is either a scalar or a vector signal. The state variables are initialized with the values stored in the `InitialCondition` parameter. For an element i of the signal vector, we create the following statement for the initialization:

```
state_UnitDelay_ID#1#i := init[i] ;
```

Here, `init[i]` is an array containing the initial values taken from the `InitialCondition` parameter. Since the output of the block depends on the state, the output function is translated to the following:

```
UnitDelay_ID#1#i := state_UnitDelay_ID#1#i;
```

Finally, the state update function is translated as follows:

```
state_UnitDelay_ID#1#i := in_1i ;
```

[5] A major simulation step is a step where the model is sampled. In a variable-step simulation it can differ from the actual sampling rate of the model.

This statement is placed at the end of the execution context in which the block is contained. This placement corresponds to the Simulink simulation loop, where the outputs are calculated first and the states are updated afterwards. The translation for the `Memory` block is done analogously.

Discrete Integrator

The `Discrete Integrator` block integrates or accumulates a value during the simulation. It supports three different integration methods: *forward euler*, *backward euler* and *trapezoidal* integration. In our translation, we currently only support the default method (*forward euler* integration or accumulation), which is specified with the `IntegratorMethod` parameter set to "`Integration: Forward Euler`" or "`Accumulation: Forward Euler`". With this method, the state s and the output o of the block are calculated for a time step n by

$$\begin{aligned} s(n) &= s(n-1) + g * T * in(n) \\ o(n) &= s(n) \end{aligned}$$

where g is a gain factor, T is a time factor and $in(n)$ is the input of the block in simulation step n. The time factor $T = t(n) - t(n-1)$ describes the time difference between two major simulation steps. Since we assume the models to have a discrete fixed step sample time, T is exactly the sample time of the block. Furthermore, if the `IntegratorMethod` is set to accumulation, $T = 1$ holds. The gain factor is specified with the `gainval` parameter, the sample time of the block is stored in the `CompiledSampleTime` parameter. Furthermore, the initial condition can be specified either in the `InitialCondition` parameter or by an additional input port, if `InitialConditionSource` is set to "`external`". Hence, the state variable for the `Discrete Integrator` block is initialized with one of the following statements:

```
state_DiscreteIntegrator_ID#1#i := init[i]; // via initial condition
```

```
state_DiscreteIntegrator_ID#1#i := in_ni; // via port; always the last input
```

Note that i is an element in the input vector, and either `init[i]` is used, which is a vector of values specified in the `InitialCondition` parameter, or in_{2i} corresponding to the input for the external initial condition. The output and state update function are translated as follows:

```
DiscreteIntegrator_ID#1#i :=
      state_DiscreteIntegrator_ID#1#i // output function
state_DiscreteIntegrator_ID#1#i :=
      DiscreteIntegrator_ID#1#i + (( G * T ) * in_1i); // state update
```

Here, `G` is the gain derived from the `gainval` parameter and `T` is either 1 for accumulation mode or derived from the `CompiledSampleTime` parameter for integration mode.

Additionally, a `Discrete Integrator` can be configured to output its state (`ShowStatePort` parameter) on a dedicated output (the second) of the block.

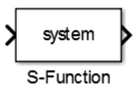
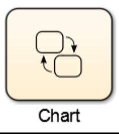
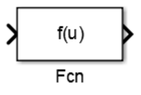
The translation is straightforward by simply using the state variable as an output. The state of the block can be reset by an additional port with the `ExternalReset` parameter, which is currently not supported by our translation. Finally, the output values of the block can be limited with the `LimitOutputs` parameter for which the bounds are specified with `UpperSaturationLimit` and `LowerSaturationLimit` parameters. Then, the block outputs its value unless it is above or below the given limits. If the value exceeds the bounds, it outputs either the upper or the lower bound.

We realize this behavior in our translation with the following function:

```
function {:inline} limit_DiscreteIntegrator_ID#i ( x: typeId) returns
    (typeId) {
  builtin_if_typeId(x < lsl[i], lsl,
      builtin_if_typeId( x > usl[i], usl, x)
      )
}
...
DiscreteIntegrator_ID#1#i :=
    limit_DiscreteIntegrator_ID#i(DiscreteIntegrator_ID#1#i);
```

Here, `usl[i]` and `lsl[i]` are the upper and lower bounds as specified in the corresponding parameters, and `typeId` is the type of the output signal as explained in the description for the `MinMax` block. Note that we use inlining (`{:inline}`) of the function in this example, which requires us to define a body of the function instead of axioms. Functions defined this way are translated into a macro by the Boogie tool, which enables the Z3 theorem prover to handle the function more efficiently for the cost of decreased readability of the model in Boogie2.

6.5.7 User-defined Functions Block Set

Function	S-Function	Stateflow
		

In this subsection, we describe the translations and approximations for the `Function` and `S-Function` blocks from the *User-defined Functions* block set and for unsupported blocks.

S-Functions

`S-Function` blocks enable the use of *system-functions* during simulation. System-functions can be written in various programming languages like MATLAB Code (M-code), *C++* or *Fortran*. Other programming languages than M-code are compiled into MATLAB/Simulink executables (*MEX-files*) and dynamically linked into the model. This also holds for *Stateflow* state machines. In our translation, we over-approximate `S-Function` blocks by assuming that they may output any value within the range outputs data type. Since we do not know what happens internally in an `S-Function`, the closed over-approximation we can make is to assume the outputs to be within their type boundaries. Hence, for every output i, we assume that every vector element j is within its type bounds:

```
havoc S-Function#i#j;
assume type_min <=S-Function#i#j && S-Function#i#j <= type_max;
```

If an `S-Function` block is used to model a Stateflow chart, we analyze the parameters of the corresponding chart object in order to constrain the approximation. To this end, we analyze whether for data objects of the charts, the `scope` parameter is set to "`OUTPUT_DATA`", which indicates an output signal. For each output data object, we extract the `props.range.minimum` and `props.range.maximum` parameters and use them for the approximation:

```
assume props.range.minimum <=S-Function#i#j &&
      S-Function#i#j <= props.range.maximum;
```

However, this is only a safe approximation if the above parameters of the Stateflow chart are correctly specifying its behavior. If those parameters are not set for an output signal, we treat it as only bounded by its data type.

Function Block

The `Fcn` block applies a mathematical expression (in M-code) to its input signals. Although this expression may contain mathematical operations that we cannot map to Boogie2, there is still a number of operations we are able to map or to approximate (see below). Hence, we parse the `Expr` parameter of the block and check whether it contains an operation we currently not support in our translation. If it does, we use an over-approximation only bounded by the type of the output signal like for `S-Function` blocks. Otherwise, we translate the expression to Boogie2. In this expression, input signals are referenced with the variable u and vector elements are accessed with $u(i)$ where i is an one-based index. Arithmetic, relational and logical operators are translated according to the mappings presented before (see `Sum`, `Product`, `RelationalOperator` and `Logic` blocks). Furthermore, we over-approximate the mathematical functions by introducing additional variables and constraining these with upper (`max`) and lower (`min`) bounds:

```
havoc fun_UID_ID;
assume min <=fun_UID_ID && fun_UID_ID <= max;
```

The name for the variable is constructed using the name of the mathematical function (`fun`), the ID of the block (`ID`) and an additional counter `UID` to avoid collisions if a function is used multiple times in an expression. The bounds for the approximations of supported mathematical functions are defined as follows:

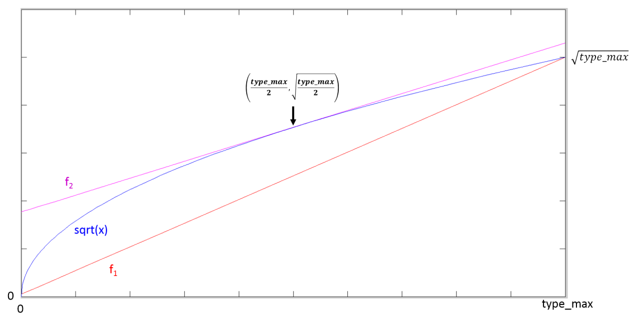


Figure 6.5: Over-approximation of *sqrt* with Linear Functions

Functions	min	max
sin, cos, tanh	-1	1
atan	$-\frac{\pi}{2}$	$\frac{\pi}{2}$
atan2	$-\pi$	π
rem(x,y)	$sign(x) \leq 0 \rightarrow sign(x) * abs(y)$	$sign(x) \leq 0 \rightarrow 0$
	$sign(x) > 0 \rightarrow 0$	$sign(x) > 0 \rightarrow sign(x) * abs(y)$
sqrt(x)	$x \leq 1 \rightarrow x$	$x \leq 1 \rightarrow 0.5x + 0.5$
	$x > 1 \rightarrow f_1(x)$	$x > 1 \rightarrow f_2(x)$

For the approximation of *sqrt*, we use linear functions. The functions f_1 and f_2 are calculated depending on the upper bound for the data type. The function f_1 is chosen as the linear function that intersects the points $(1,1)$ and $(type\_max, \sqrt{type\_max})$ and f_2 as the tangent of the point $\left(\frac{type\_max}{2}, \sqrt{\frac{type\_max}{2}}\right)$. The functions f_1 and f_2 are depicted in Figure 6.5 and together from a linear over-approximation of the *sqrt* function in the interval $[1, type\_max]$.

For example, a `Fcn` block wit a scalar input and the `Expr` parameter set to "`4*rem(u,3)`" is translated into:

```
havoc rem_1_ID;
assume (signum_real(in_1) <= 0) ==>
  (signum_real(in_1)*abs_real(3) <= rem_1_ID && rem_1_ID <= 0);
assume (signum_real(in_1) > 0) ==>
  (0 <= rem_1_ID && rem_1_ID <= signum_real(in_1)*abs_real(3));
Fcn_ID_1#1 := 4 * rem_1_ID;
```

6.5.8 Sinks Block Set

In our translation, we do not translate blocks from the *Sinks* block set, e. g., `Terminator`, `Scope` or `Display` blocks, since they do not influence the behavior of the model.

So far, we have introduced the translation mappings for specific block types. However, there are some mechanisms we need to consider for multiple (but not all) block types. We discuss these mechanisms in the following subsection.

6.5.9 Implicit Casts and Mechanisms for Multiple Block Types

In the previous section, we have presented our transformation rules for specific block types. In this section, we discuss mechanisms that affect almost every block. We describe how we deal with implicit casting in our translation before we discuss the saturation of output signals, which can be enabled for various block types.

Implicit Casts

Simulink does not require explicit typing of the data types for signals within a model. Instead, it supports inheritance of data types and offers different inference algorithms to calculate the inherited data types. An important benefit of our approach is the use of run-time information, which we extract from the Matlab/Simulink tool for models in some state after initialization. Hence, we have the information about the actual data type the simulation environment is using.

However, Simulink is weakly typed and performs implicit casting. This means it is possible in Simulink to add an integer to a floating point variable or to use floating point variables or integers as inputs for `Logic` blocks. Furthermore, `MultiPortSwitch` blocks accept floating point signals although their index values are integers and implicitly cast these signals. In contrast, Boogie2 is strongly typed. Every built-in arithmetic, logical, relational, or assignment operation requires all arguments to be of the same type and rejects the specification otherwise. To be able to translate Simulink models, we need to make implicit casts explicit: For every operation that we translate, we need to ensure that all types are suitable and that we cast a variable into the required type, if necessary. A major problem is that the Boogie verification framework does not support even explicit casts. However, the Z3 theorem prover does (at least for integers and reals). Hence, we extend the preamble of our formal model with the following Z3-functions:

```
function {:bvbuiltin "to_real"} int_to_real(x:int) returns (real);
function {:bvbuiltin "to_int"} real_to_int(x:real) returns (int);
```

In addition, we need to define the cast operations for real and integer signals that are treated as Boolean signals. According to the Simulink documentation, a numeric value is interpreted as *false* if it is 0 and as *true* otherwise. Hence we define the cast operations in our preamble as follows:

```
function {:inline} bool_to_real(x:bool) returns (real){
    builtin_if_real(x, 1.0, 0.0) }
function {:inline} bool_to_int(x:bool) returns (int) {
    builtin_if_int(x, 1, 0) }
function {:inline} real_to_bool(x:real) returns (bool){
    builtin_if_bool(x == 0.0, false, true) }
function {:inline} int_to_bool(x:int) returns (bool) {
    builtin_if_bool(x == 0, false, true)}
```

To take care of implicit casts, we check the types of the signals whenever we create an assignment or an expression in Boogie2 during the translation of any Simulink element whether they have compatible data types. If they are not compatible, we use one of the casting functions corresponding to the type of the output signal. For example, assume there is a `Sum` block that has a floating point signal connected to its first input (in_1), an integer signal connected to its second input (in_2) and outputs a floating point signal. The resulting translation for the block in Boogie2 is as follows:

```
Sum_ID#1#0 := in1 + int_to_real(in2);
```

Note that Simulink also supports explicit casting with `DataTypeConversion` blocks. In our translation we treat these blocks like port blocks and apply the cast function to the outputs.

Saturation

Many blocks in Simulink provide an automatic saturation of output signals in case of overflows. The saturation is activated if the parameter `SaturateOn-IntegerOverflow` is set to "on". If an output signal is saturated, it cannot exceed the bounds of its data type. If a calculation leads to a value bigger (or smaller) than the bound, the block outputs the maximum (or minimum) value of the data type instead. To support this behavior in our translation, we create saturation functions for each data type used in the model. For each data type we create the following function:

```
function {:inline} saturate_type ( x: boogieType) returns (boogieType) {
    builtin_if_boogieType(x < type_min, type_min,
    builtin_if_boogieType( x > type_max, type_max, x))
}
```

In this statements, `type` is a Simulink type that is mapped to `boogieType` in our translation and `type_min` and `type_max` are the lower and upper bound of the Simulink type. Note that this function has a similar behavior like the limit function of the `Discrete Integrator` block. For example, the saturation function for the Simulink type *uint8* is defined as follows:

```
function {:inline} saturate_uint8 ( x: int) returns (int) {
    builtin_if_int(x < 0, 0, builtin_if_int( x > 255, 255, x))
}
```

Here, *uint8* is mapped to the Boogie2 type `int`. Assume there is a `Sum` block that adds two signals of the type *uint8* and has saturation enabled. With our translation, the resulting assignment is:

```
Sum_ID#1#0 := saturate_uint8(in_1 + in_2);
```

Both, saturation and cast functions are added during the translation if necessary, i. e., if the saturation flag is set, or if signals have different data types.

6.6 Summary

In this chapter, we have presented our approach for an automatic transformation of Matlab/Simulink models into the formal model in Boogie2. To this end, we have first given an overview over the general translation process. Then, we have discussed the assumptions and limitations to the models to make our automatic approach applicable. We have introduced the basic concepts of our formal model and how we calculate the control flow graph for the model. In our automatic transformation, we iterate over the graph to translate block by block. Hence, we have discussed the behavior and parameters of the supported block types and presented our translation into our formal Boogie2 model for each block. Finally, we have presented how we deal with saturation and implicit casts, which can occur at various blocks within a Simulink model.

Overall, with the transformation presented in this chapter, we can translate a Simulink model automatically in a formal model that preserves or over-approximates the behavior. By enhancing this formal model with assertions and invariants, which we both also create during the translation, we can automatically verify the model using the Boogie verification framework. In the next chapter, we present in detail how we generate the assertions and invariants for a given Simulink model, and how we compose the formal specification that is passed to the verification tool chain.

7 Automatic Inductive Verification of Simulink Models

In this chapter, we present our verification approach for Simulink models, which is based on the formal model generated by the translation presented in the previous chapter. In our approach, we use two verification strategies to automatically verify a given Simulink model: inductive invariant checking and k-induction. To enable the automatic verification of the generated formal model w. r. t. the absence of important error classes, we compose the model, the verification goals and invariants into a common formal model. In the following, we refer to this composition as *formal specification*. The formal specification, which is generated according to a particular verification strategy (with a number of unrolls for the k-induction), can be verified using the Boogie tool chain.

This chapter is structured as follows: First, in Section 7.1 we describe how we compose the formal specification from the formal model, verification goals and loop invariants for inductive invariant checking and k-induction. In Section 7.2, we describe how we apply our slicing technique to obtain smaller specifications for k-induction to reduce the complexity of the verification task. We present in Section 7.3 how we generate the verification goals for certain error classes during the translation into the formal model. Then, we present in Section 7.4 which loop invariants we automatically generate when translating a given Simulink model.

7.1 Automatic Generation of Formal Specifications

To verify a Simulink model, we compose the formal model presented in Chapter 6 with verification goals and invariants to obtain a formal specification. Depending on the use of the particular verification strategy, more precisely inductive invariant checking or k-induction, the structure of the formal model is different. In this section, we present the general composition for both verification strategies.

As described in Chapter 4, the idea of the verification process is to start with inductive invariant checking since it is the best scaling method. Moreover, inductive invariant checking is naturally supported by the Boogie verification framework. When transforming the formal specification into verification conditions, the Boogie tool automatically derives some invariants using abstract interpretation and automatically performs loop cutting. However, inductive invariant checking requires stronger loop invariants than k-induction.

The second step in the process is to use k-induction for the verification. k-induction requires more computational effort (depending on k) but also requires weaker invariants. Since k-induction is currently not directly supported by the Boogie tool, we generate a formal specification that is tailored to the desired k and adopt the approach for *combined case k-induction* presented by Donaldson et al. [DHKR11] (see Section 2.3.2).

In the next two subsections, we describe how we generate the specifications for each verification strategy.

7.1.1 Inductive Invariant Checking

In the first step in our verification process, we attempt to verify the model using inductive invariant checking. The idea of inductive invariant checking for loops is to show the following base case and induction step:

- **Base Case** All initial states S_0 before the loop is executed satisfy the loop invariant I: $S_0 \models I$.
- **Induction Step** For an arbitrary state S_n that satisfies the loop invariant I, the subsequent state S_{n+1} (obtained by the execution of the loop body) still satisfies I: $S_n \models I \longrightarrow S_{n+1} \models I$.

In our formal model, we represent the simulation loop as a (*while*) loop in the Boogie2 specification. This means all blocks are translated into Boogie2 assignments in the loop body. Furthermore, all Boogie2 variables generated for the block outputs and internal states are initialized according to initial values specified in the block parameters or set to 0. Since Boogie is naturally supporting inductive invariant checking, we model the simulation loop as a (desugared) loop without loop cutting (see Section 2.3.2) to invoke the loop detection of the Boogie tool. This is necessary since the Boogie tool only performs the invariant inference if it detects a loop.

When constructing the formal specification, automatically generated assertions are added to the loop body either before or after the assignments for the corresponding block. Furthermore, loop invariants are added to the loop head. These invariants are either automatically generated during the translation or manually specified by the user.

Figure 7.1 depicts how we generate the formal specification for inductive invariant checking from the translation artifacts. Firstly, we add the declarations for each variable (according to naming scheme and the data type) used in the

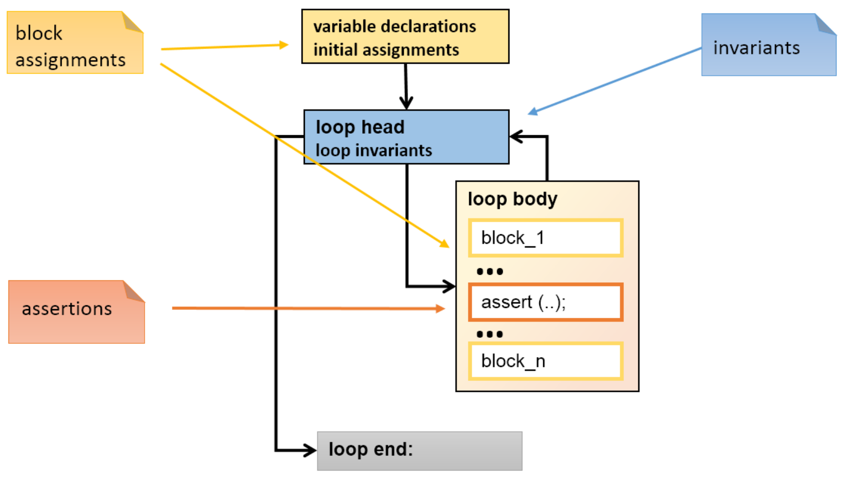


Figure 7.1: Formal Specification for Inductive Invariant Checking

formal model. After that, the initial assignments are added corresponding to the initialization phase. Then, the simulation loop is modeled as a (desugared) loop. The loop invariants are added to the `goto`-block representing the loop head. The loop body contains the assignments that represent the semantics of the Simulink blocks obtained by the translation presented in Chapter 6. In addition to the statements obtained by the translation, we also add verification goals for the error classes of interest as assertions.

With this specification, we realize the following verification idea: First, we show that if the model is executed on a valid state, it is in a valid state afterwards using inductive invariant checking. We do this by showing that some initial state S_{init} satisfies the loop invariant I initially. Then, we show for an arbitrary state S_n that satisfies I that the consecutive state S_{n+1} satisfies the invariant I. Second, we use verification goals to show that none of the considered errors may occur during one execution step of the model. In other words, we ensure that every intermediate state between two simulation steps is valid w. r. t. the error classes of interest.

If the Boogie tool succeeds in proving these formulas, the model is free of such errors. Otherwise, the Boogie tool reports an error with the line number, the trace, and a counterexample.

There are three possible types of errors: The Boogie tool either reports a

(1) *loop invariant entry error* or an

(2) *invariant maintenance error* or an

(3) *assertion violation.*

If it reports (1), then the invariant was not satisfiable even before entering the loop which usually indicates a problem with a manually specified invariant. If it reports (2), this could also indicate a problem with the manually specified invariants or that the automatically generated invariants are to weak to verify the model. Finally, if it reports (3), then one of the checks for error classes of

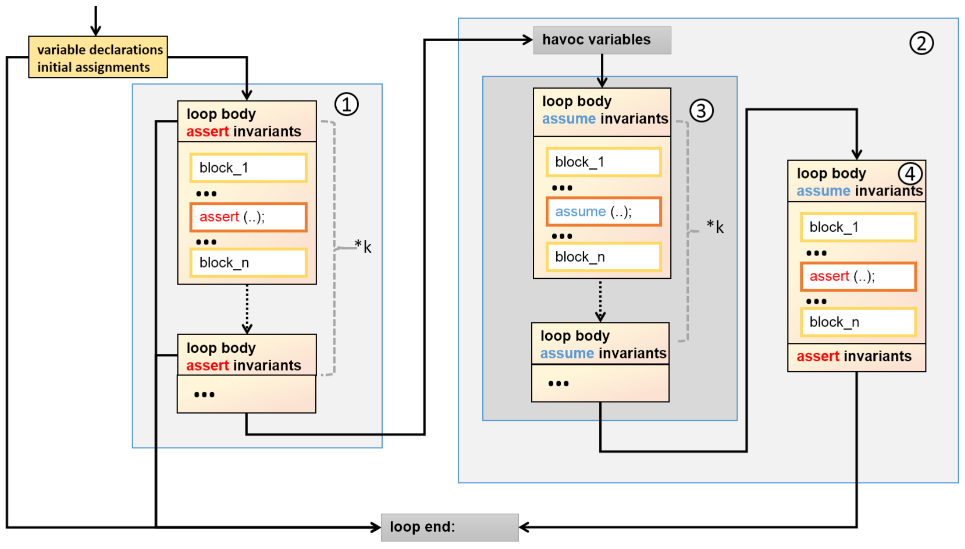


Figure 7.2: Formal Specification for K-Induction

interest failed. This can either indicate an actual error in the Simulink model, or that the loop invariants are not strong enough to verify the formal specification. Then, we have to inspect the counterexample whether it is spurious. If this is the case, the user has to decide whether he wants to add further invariants manually or whether he wants to try k-induction first. For the latter, we generate a formal specification tailored to a particular k as presented in the next subsection.

7.1.2 k-Induction

If inductive invariant checking was not suitable to prove the absence of errors in a formal specification due to weak invariants, we use k-induction in our process to verify the specification. As introduced in Section 2.3.2, the verification idea of k-induction is the following:

- **Base Case** Starting from some initial state S_0 we show that k consecutive states $S_0,\dots,S_k$ satisfy some property I (e. g., a loop invariant): $S_0 \models I \wedge S_1 \models I \wedge \dots \wedge S_k \models I$.
- **Induction Step** For k arbitrary, consecutive states $S_n,\dots,S_{n+k}$ (obtained by k executions of the loop body) that satisfy the property I, we show that the subsequent state S_{n+k+1} still satisfies I: $S_n \models I \wedge \dots \wedge S_{n+k} \models I \longrightarrow S_{n+k+1} \models I$.

Since the Boogie tool currently does not support k-induction, we adopt the approach for *combined-case k-induction* by Donaldson et al. [DHKR11] (see Section 2.3.2) and generate a formal specification tailored to a particular k. Here, we encode both, the base case and the induction step, into one verification condition.

Figure 7.2 depicts the specification generated for k-induction. The initialization phase is modeled similar to inductive invariant checking. For the base case (1), we unroll the loop body k times (indicated by the "$*k$") before the loop is entered. Each unrolled loop body corresponds to one simulation step in the Simulink model. For every step, we ensure that the loop invariants are satisfied by adding them to the loop body as assertions. Furthermore, we add checks for the considered error classes to the unrolled loop body as assertions. To model the arbitrary start state for the induction step (2), we first set all variables of the model to an arbitrary value using the `havoc` command.

Then, the loop body is unrolled k times (3) where all invariants and verification goals are assumed to hold in k consecutive steps. In other words, (3) describes all possible executions of k steps where neither the invariants nor the verification goals are violated. Finally, the loop body is unrolled another $(k+1)$th time (4). In this last unrolled loop body, all invariants and verification goals are checked with assertions.

Note that we apply loop cutting in the formal specification generated for k-induction. Hence, the resulting Boogie program is free of cycles. More precisely, the base case of the specification is actually a sequence of k loops as obtained by loop cutting. Furthermore, the jumps to the end of the loop are preserved to be able to also model execution paths where the loop finishes in less than k steps. However, since the specification is acyclic, the Boogie tool does not detect a loop and, hence, neither invokes its invariant inference procedure nor distinguishes whether an assertion is an invariant or an error check. For that reason, we have decided to encode the type of the assertion directly into a label in the formal specification.

When verifying the formal specification wit the Boogie tool, it either returns an *assertion violation* or that it has been successfully verified. In case of Boogie returning an error, we evaluate the label corresponding to the violated assertion. From the label

```
LABEL_ASSERTION#[ID]#[TYPE]#[STEP]
```

we extract the `TYPE` and the `STEP` parameter. The `TYPE` parameter indicates whether it is an assertion for an invariant or for an error check. The following table shows the possible error types:

TYPE	Description
0	over- and underflow check
1	division-by-zero
2	range violation
3	loop invariant (automatic)
4	loop invariant (manual)

If `TYPE` is set to 0, 1 or 2 it indicates that an error check failed which means there is an error in the model. If it is set to 3 or 4 the violated assertion is an invariant that is either too weak or indicates that a manually specified invariant is not suitable (4).

The STEP parameter indicates in which of the unrolled loop bodies the assertion was violated. A value from 1 to k indicates that the violated assertion originates from the base case. If an error check failed in the base case, it indicates that there is an error in the model. If STEP has the value of $2k+1$ the assertion was violated in the induction step. This indicates that there either may be an error in the model, that the chosen k is not high enough, or that the invariants are not strong enough. Hence, we have to review the counterexample and either find an actual error or a spurious counterexample. To eliminate the latter we increase k or add some manually specified invariants. If we increase k, a new formal specification is generated tailored to the new k and the Boogie tool is invoked with the new specification again.

However, since the size (and complexity) of the verification condition increases with the size of k, the time needed for the verification also increases. To overcome this problem, we use slicing to reduce the complexity of the Simulink model and the resulting verification problem for a particular assertion violation discovered in an earlier verification attempt.

7.2 Slicing for k-Induction

The basic idea of our approach is to start with the best scaling verification strategy (inductive invariant checking) first. If we are not able to show certain properties with this strategy, we start using k-induction, which scales less well since the size of the verification condition increases with k. To overcome this problem, we use the slicing technique presented in Chapter 5 to reduce the size of the verification condition by focusing on a particular property.

The key idea for this phase in our verification process is to translate and afterwards verify only a *slice* of the model instead of the entire model. To calculate the slice, we obtain the slicing criterion from the assertion violations of an earlier verification attempt, e. g., the result of the last k-induction run that did not finish in reasonable time.

Figure 7.3 depicts the basic procedure to apply slicing within our verification process. Starting from a given Simulink model, we generate both the Control Flow Graph (CFG) and the MATLAB/Simulink Dependence Graph (MSDG). Then, we use the error label to determine the actual block b where the error has occurred. This block is used as the slicing criterion to perform a static backward slice (2) as presented in Chapter 5. As result, we obtain a MSDG where all nodes that belong to the slice are marked. Once the slice is calculated, we start the translation of the model into the Boogie2 specification by iterating over the CFG as discussed in Chapter 6. However, in contrast to the original translation algorithm, we only translate those blocks that are marked in the MSDG. Finally, the translation artifacts are composed into a specification for k-induction (3) as presented in the previous section.

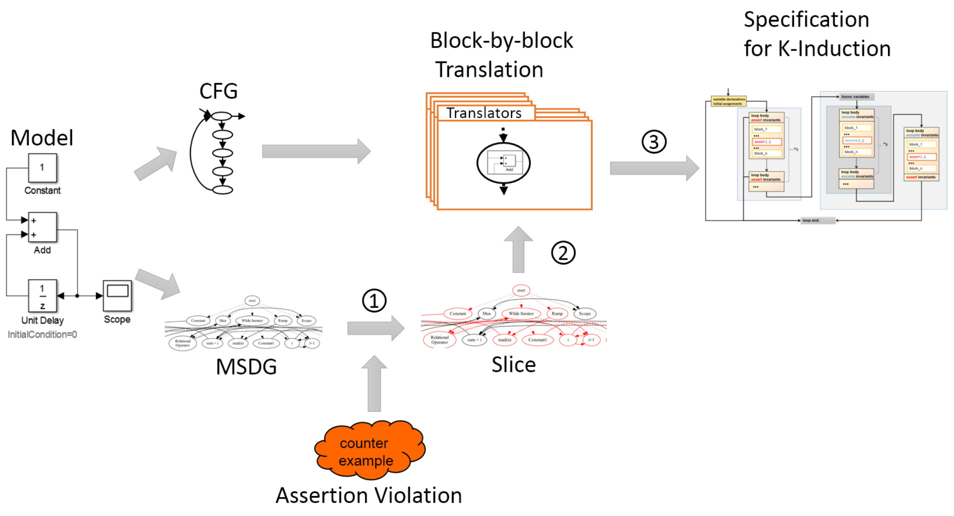


Figure 7.3: Slicing Procedure for K-induction

The slice of the formal specification only contains the assignments, invariants and verification goals that directly or transitively influence the assignments and verification goals for the block b. With that, we can reduce the size of the verification condition and hence, the time needed for the verification w. r. t. the properties of interest for a particular Simulink block. This in turn enables us to use greater values for k than without slicing.

Since we use static backward slicing to calculate the slice for the block b, we safely over-approximate the set of blocks that influence the calculation result at block b. This is the case since the calculation of data and control dependence in our approach and the calculation of the slice is conservative. More precisely, the slice also contains blocks from possibly infeasible execution paths and all blocks that are relevant to calculate these. Furthermore, the slice contains all blocks on which b is transitively data dependent including dependences that range over more than one simulation step. However, all blocks in the slice are also part of the original model and all statements generated for the slice are also generated for the original model. Hence, if we verify the absence of an error for block b in the slice, this also holds for the original specification. In other words, the verification of the slice is sound w. r. t. the original model.

7.3 Automatic Generation of Verification Goals

In this section, we present the automatic generation of specific verification goals w.r.t. important and typical run-time errors that are usually hard to detect using testing. With the generated verification goals, we are able to apply the general verification framework as presented in Section 7.1 in a fully automatic way to ensure the absence of the error classes of interest. The currently supported classes are the following:

- **Overflows and Underflows** We consider every occurrence of an over- or underflow in the model as an error. An over- or underflow occurs if the value of a signal exceeds the bounds of its data type.
- **Range Violations** The second class of errors we consider are range violations at the control input of `MultiPortSwitch` blocks. These occur if the value for a control input is outside of the ranges specified in the block parameters. It is possible that this behavior is intentional and the model behaves correctly using the signal connected to the default port. However, it might also be unintended since if not configured manually, the last data input is the default input by definition. Hence, we consider every range violation as an error.
- **Division-By-Zero** We consider every division-by-zero as an error. Moreover, we assume that they are not modeled intentionally, e. g., by setting a parameter to a value like "`1/0`". Hence, we focus on blocks that may use an input signal as a divisor. Those are the `Product` and the `Fcn` block. Note that `S-Function` blocks, which are over-approximated in our formal model may also use input values in a division. Since we consider these blocks as black boxes and a signal value of `0` may also be intentional, we require the user to specify a manual invariant for the inputs to `S-Function` blocks if `0` is not a valid for a particular input signal.

In our approach, we want to ensure that none of the above errors can occur at any time during the execution of a given model. Since loop invariants can be violated in the Hoare calculus or weakest precondition approach within a loop as long as they are satisfied at the end of the loop, we need to use an additional mechanism to ensure that no error occurs in any intermediate state of the loop. To realize this, we add verification goals to the specification at every point where one of the above errors may occur. These verification goals are modeled with *assertions* in the formal specification. Therefore, we automatically generate the verification goals depending on the block type and the parameter configuration during the translation of the blocks into the formal model. These assertions are added to the formal specification. Assertions for range violations and division-by-zero errors are placed before and assertions for over- and underflows after the statements representing a block.

While simply adding these assertions is suitable to verify the models for the absence of errors, it is not well suited to track back the cause of an error in case that Boogie reports a counterexample. As presented in Section 2.3.2 the Boogie tool translates every `goto` block within the specification into a block equation. If it finds a counterexample, it returns the label of the block equation that could not be satisfied, a line number for the assertion, a trace of labels for satisfied block equations and the variable assignments. To identify the block corresponding to the assertion, we have to analyze the context of the `assert` statement. To obtain more precise traces, we add additional `goto` blocks into the Boogie2 specification for every assertion. With that, we can directly encode the corresponding Simulink block and type of the error into the label that is reported if an error is found. Hence, we translate the assertions for all error classes into:

```
goto LABEL_ASSERTION#ID#TYPE#STEP;
LABEL_ASSERTION#ID#TYPE#STEP:
assert ( ... );
```

Here, ID is the unique identifier of the block. Furthermore, TYPE indicates the error class and STEP specifies the unroll step if k-induction is used as presented in Section 7.1.2. With that, we can directly use the trace to identify the Simulink block and the type of an error and report it to the user.

7.3.1 Encoding Over- and Underflow Checks

The assertion for over- and underflows checks if the result of a calculation in a block can exceed the data type bounds of the output signal. Hence, the assertion has to be placed after the calculation of the block outputs. We generate the following assertion:

```
// translation for a block with ID = 42,
out_ij := (...);

goto LABEL_ASSERTION#42#0#STEP;
LABEL_ASSERTION#42#0#STEP:
assert (type_min <= out_ij && out_ij <= type_max);
```

Here, out_{ij} represents the variable for the ith output (and the jth signal in a vector) of the block, type_min and type_max are the bounds of the data type of out_{ij} and the “0” in the label indicates an over- and underflow check.

This assertion is generated for every block type that performs a calculation which might possibly exceed the bounds of its data type. Such blocks are for example Sum, Product, DiscreteIntegrator, Fcn, and Gain blocks.

7.3.2 Encoding Division-by-zero Checks

A division-by-zero error can occur in Product and Fcn blocks. For Product blocks, the generation of assertions is straightforward. We generate an assertion for each input that is used as a divisor, i. e., the entry in the Inputs parameter for the block is a “/”. This assertion is placed before the output of the block is calculated:

```
// assertion for a Product block with ID = 42
// where the input in_ij corresponds to a "/" in the Inputs Parameter

goto LABEL_ASSERTION#42#1#STEP;
LABEL_ASSERTION#42#1#STEP:
assert ( in_ij != 0);

// translation for a block with ID = 42,
out_ij := (... / in_ij ..);
```

Note that the “1” in the label indicates that this assertion is a division-by-zero check.

For `Fcn` blocks, we have to consider that the inputs are part of an expression specified in the `Expr` parameter. Hence, it is not suitable to only check the inputs. Instead we need to analyze the expression and need to produce checks for each subexpression used as a divisor. For a `Fcn` block that uses a subexpression sub_n as divisor in the expression specified in `Expr`, we generate the following assertion to check for a division-by-zero:

```
// assertion for a Fcn block with ID = 42
// where sub_n is a subexpression of Exp

goto LABEL_ASSERTION#42#1#STEP;
LABEL_ASSERTION#42#1#STEP:
assert ( sub_n != 0);

// translation for a block with ID = 42,
out_ij := (... / sub_n ..);
```

Note that an assertion is created for each subexpression used as divisor.

7.3.3 Encoding Range Violation Checks

Range violations can occur at `MultiPortSwitch` blocks if the signal connected to the control port has a value that is not in the range specified by the `DataPortOrder` and `DataPortIndices` parameter. Hence, we place the assertion for this check before the translation of the block. However, the assertion is only generated if the block uses its last data input as default input since we assume it may be used unintentionally.

The assertion is constructed from the contents of the `DataPortOrder` and `DataPortIndices` parameter, which we store in a `idx` array that maps the number of a data input to its index value. For a `MultiPortSwitch` block, we generate the following assertion:

```
// assertion for a MultiPortSwitch block with ID = 42

goto LABEL_ASSERTION#42#2#STEP;
LABEL_ASSERTION#42#2#STEP:
assert (in_1 == idx[2] || ... in_1 == idx[n] );

// translation for the MultiPortSwitch with ID = 42,
goto MultiportSwitch_42_data_2, ... MultiPortSwitch_42_data_n;
...
```

In the above listing, n is the number of inputs to the block, in_1 is the control input and the 2 in the label indicates that the assertion is a check for a range violation.

7.3.4 Other Verification Goals

In our framework, we currently support the automatic generation of verification goals described in the previous subsections. However, it is generally possible to generate assertions for further goals automatically. For example, many Simulink blocks have parameters that enable the specification of a minimal and maximal output value. However, these bounds are not mandatory. If a block violates them, a warning may be reported by the simulation engine but there are no further consequences for the behavior of the model. Since these are not run-time errors, we currently have not included these parameters into our automatic generation of verification goals. Another possible extension may be the translation of `ProofObjective` and `Assumption` blocks introduced by the *Simulink Design Verifier* toolbox. These blocks can be directly encoded into assertions in our formal specification. However, since they are not part of the standard Simulink block library, we decided to currently not support them in our approach.

With the automatically generated verification goals presented in this section and the formal model from the previous chapter, we are able to ensure that none of the errors of interest can occur within a given Simulink model. However, to verify the model with inductive techniques, we also need to derive suitable loop invariants to obtain useful verification results.

7.4 Loop Invariants

In this section, we describe the automatic generation of some loop invariants needed for the verification of the simulation loop in our approach. Furthermore, we also enable the user to specify additional invariants manually.

As discussed in Section 7.1, the verification of loops using the Boogie verification framework (and weakest preconditions) generally consists of two steps:

1. We show that the program state before the execution of the loop satisfies the loop invariant (base case).
2. For an arbitrary program state satisfying the loop invariant, we show that the program state after the execution of the loop body (still) satisfies the loop invariant (induction step).

Since in our formal model a Simulink model is represented by its simulation loop, we need to specify invariants that are suitable for the entire model. That means we need to specify invariants that are satisfied between two arbitrary, consecutive simulation steps.

Note that a major challenge arises from the fact that in Boogie2 the types are unbounded while Simulink types have bounds. Since for the verification of loops an arbitrary model state is used, we need to ensure with loop invariants that this state is valid w. r. t. data type bounds in the Simulink model. Hence, we generate invariants that limit the range of each variable to the bound given by the type. Furthermore, we generate invariants that are not required to verify the model but greatly reduce the needed verification time.

In Boogie2, loop invariants are modeled as assertions that are placed in the code block representing the loop head (see Section 2.3.2, Formula (2.10)). Similar to the verification goals, we use labels to encode the corresponding Simulink block, the type of the invariant, and the unroll step for k-induction. An automatically generated invariant is indicated with the `TYPE` segment in the label set to 3. Manually specified invariants are indicated by a 4.

7.4.1 Automatically Generated Invariants

In our approach, we automatically generate two kinds of invariants: The first kind are invariants that limit the variables in our formal model to the corresponding Simulink data types. They are necessary to verify the formal model for the absence of errors for the error classes discussed in the previous section. The second kind of invariants are generally not necessary to verify the formal specification. However, adding these invariants greatly improves the verification time.

Data Type Invariants

Besides Boogie2 assignments and verification goals, we also generate invariants for each block directly during the translation. For an invariant limiting the variables to their type bounds, we generate the following assertion that is placed in the loop head:

```
//invariants for a block with ID = 42,
goto LABEL_ASSERTION#42#3#STEP;
LABEL_ASSERTION#42#3#STEP:
assert ( type_min <= out_ij && out_ij <= type_max);
...
```

Here, `type_min` and `type_max` are the lower and upper bounds for the Simulink data type of an output signal out_{ij}. A data type invariant is generated for each output or state variable in our formal model.

Control Flow Invariants

While the above presented data type invariants are well suited for the use of inductive invariant checking, they may lead to scalability issues when using k-induction. The problem is that the invariants are assumed to hold k times in the induction step regardless of whether the execution context of the corresponding block is executed or not. To show that all invariants hold, the underlying theorem prover Z3 has to explore every feasible execution path in the model. For every feasible path, it has to explore all possible assignments for each variable, even if the variable does not belong to the execution path and hence, is unrelated to any of the variables in the feasible path. In other words, the above presented data type invariant leads to an unnecessary large state space when using k-induction.

To reduce this problem, we generate invariants for the data type bounds that include control flow dependencies. More precisely, we specify when a data type bounds invariant has to be satisfied. Hence, we generate the following data type invariant for each block b that is within a conditional execution context e:

```
assert (p_e ==> inv_b);
```

In the above listing, p_e is the predicate specified by the predicate block of the execution context (e. g., the predicate variable generated for an `Enable` block). The data type invariant inv_b for block b is generated as presented in the previous subsection. In other words, we only require a signal value to be within its data type bounds if it is part of a feasible execution path. With that, we can limit the search space for the solver to valid execution traces of the model.

Note that these more precise invariants for data type bounds limit the search space for the solver in each unrolled simulation step and hence greatly decrease the verification time for our case studies when using k-induction. To improve the verification time even more, we constrain the arbitrary start states for the induction step to be consistent with the predicates of conditional execution contexts. Hence, we generate invariants that relate the variables used to calculate a predicate to the more precise data type invariants. For a `Switch` block, we generate an invariant that describes how the predicate is calculated:

```
//invariants for a Switch block with ID = 42,
goto LABEL_ASSERTION#42#3#STEP;
LABEL_ASSERTION#42#3#STEP:
assert (predicate_Switch_42#1#0 == criteria);
```

Analogously to the mapping presented in Section 6.5, the `criteria` is either "in_2 `< Threshold`", "in_2 `!= 0`" or "in_2 `<= Threshold`".

7.4.2 Manually Specified Invariants

Besides the automatically generated invariants, we also enable the user to specify additional invariants manually. Manually specified invariants might be required to succeed with the verification if the model is too complex to find a suitable k for k-induction or if the automatically generated invariants are too weak to be used with a smaller k.

For this purpose, we enable the user to specify further invariants that are automatically added to the formal specification during the translation process. For manually specified invariants, we require the following syntax.

```
[ID]@[Invariant]@[Port]
```

Here, `ID` is the identifier of the block and `Invariant` is an expression over a variable generated for the block in the formal model that evaluates to a Boolean value. The `Port` indicates the corresponding output of the block and is used to apply casts to the expression if data types are not matching.

For example, the invariant “`769@(state_UnitDelay_769#1#0 <= 9)@1`” for a `UnitDelay` block with the ID 769 and one integer output signal is translated into the following:

```
goto LABEL_ASSERTION#769#4#STEP;
LABEL_ASSERTION#769#4#STEP:
assert (state_UnitDelay_769#1#0 <= 9);
```

Note that the value 4 for the `TYPE` indicates that it is a manually specified invariant.

To define an invariant over a range of values, it is possible to use a `range` macro: “`702@range(state_UnitDelay_702#1#0,-1000.0,1000.0)@1`”. The macro translates into the following:

```
goto LABEL_ASSERTION#702#4#STEP;
LABEL_ASSERTION#702#4#STEP:
assert (-1000.0 >= state_UnitDelay_702#1#0 &&
        state_UnitDelay_702#1#0 <= 1000.0);
```

Note that if the user specifies an invariant that is not consistent with the model, an *invariant maintenance error* is reported by the Boogie tool.

7.5 Summary

In this chapter, we have presented our verification approach for Matlab/Simulink models. We have first discussed the two verification strategies we use in our approach: inductive invariant checking and k-induction. For both, we have presented how we automatically generate a formal specification from the formal model. These specifications are tailored to the desired verification strategy. We have shown how our slicing technique is used in our verification approach to obtain a slice of the formal specification, which can be used to significantly reduce the verification effort. Furthermore, we have presented the automatic generation of verification goals for certain run-time errors and the automatic generation of invariants that are necessary to automatically verify the absence of these errors in a model.

8 Implementation

To show the practical applicability of our approach, we have implemented our automatic slicing and verification techniques presented in the previous chapters in our *MeMo* tool suite. The *MeMo* tool suite implements all steps of our verification process from parsing via the formal verification itself of the model to a visualization of the counterexamples obtained from the Boogie verification framework. Furthermore, it provides a graphical user interface that guides the user through the verification process.

8.1 The MeMo Framework

The MeMo tool suite generally consists of three modules corresponding to the verification process presented in Chapter 4, a parsing module and a graphical user interface. Figure 8.1 depicts the basic components of our tool.

- **Parsing Engine** The parsing engine provides the other components in our tool suite with the necessary information. It does not only parse the model file to create an intermediate representation used by the other com-

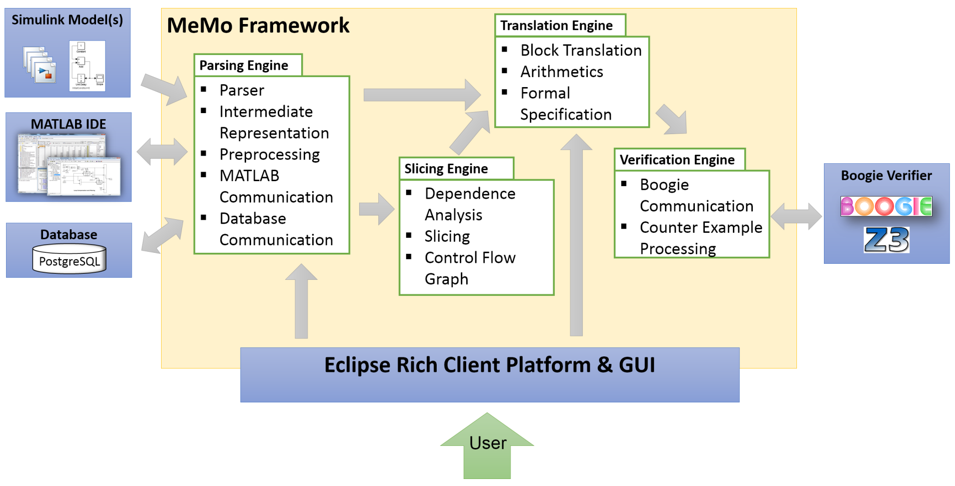


Figure 8.1: The MeMo Tool Suite

ponents. It also enhances the parsed model with information extracted from the MATLAB/Simulink tool and performs some preprocessing steps like the flattening of subsystems and the bus analysis (as presented in Section 5.4.1). Furthermore, it stores all information gathered in a database for subsequent use without the need to parse the model again.

- **Slicing Engine** The slicing engine takes the model in our intermediate representation and performs the dependence analysis (as presented in Section 5.2) to create a dependence graph representation. This representation is then used to calculate static forward and backward slices (as presented in Section 5.3). Additionally, it calculates the sorted order, and hence, the control flow graph using the execution contexts and dependences calculated during the dependence analysis and provides it to the translation engine.
- **Translation Engine** The translation engine uses the intermediate representation of the model and the control flow graph provided by the slicing engine to perform the block-by-block translation of the Simulink model into the formal Boogie2 model as presented in Chapter 6. Furthermore, it manages the data type mappings of all Simulink data types within a model, the mappings for the supported arithmetical operations and mechanisms that are not specific to a certain block type (see Section 6.5.9). Finally, it generates the formal specifications for the desired verification technique depending on a given k.
- **Verification Engine** The verification engine uses the formal specification provided by the translation engine to verify it using the Boogie verification framework. Furthermore, it processes the counterexamples such that they can be displayed within the graphical user interface.

The MeMo tool suite is mainly written in the platform independent programming language *Java*. It is written for the *Eclipse Rich Client Platform (Eclipse RCP)*[1], which enables us to reuse and to extend best practices and design patterns provided by the Eclipse RCP framework. Currently, the MeMo suite consists of five eclipse plug-ins that together comprise about 50.000 lines of code[2]. In the following sections, we summarize the main features of the implementation for the components presented above.

8.2 Parsing Engine

The main purpose of our parsing engine is to provide our techniques presented in the previous chapters with the required information stored in a suitable and complete intermediate representation. To obtain this representation, we do not only need to parse the file for the Simulink model of interest, but also to obtain all information about referenced library blocks or models as well as run-time

[1]http://eclipse.org/

[2]Note that this number also includes code comments and few classes that are automatically generated.

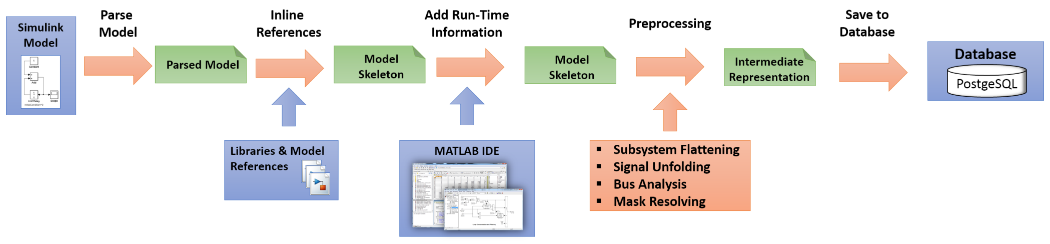


Figure 8.2: The Parse Process for a Simulink Model

information that is not available directly from model files. Figure 8.2 depicts the steps of our parsing process realized by our parsing engine. In the following subsections, we give a brief description of the steps and their implementation.

8.2.1 Parsing of Model and Inlining of References

In our parsing step we start by parsing the model file for the model of interest first. Therefore, we use a parser for Simulink MDL files, which is part of the *Simulink Library for Java*[3] originally developed at the Technische Universität München which is based on the *CUP Parser Generator for Java*[4]. Note that we only use the generated parser and the data structures to create an Abstract Syntax Tree (AST) from the Simulink Library for Java and create our own intermediate representation.

Once the model is parsed, we traverse the abstract syntax tree to create a skeleton in our intermediate representation. Whenever the traversal encounters a section in the AST that represents a block of the type `Reference`, we search in an user-defined list of models and libraries for the referenced subsystem and parse the corresponding file if necessary. The `Reference` block is then replaced by the referenced subsystem. This is done recursively for replaced references such that in the final AST all sections represent basic blocks. Then, the AST is transfered into our intermediate representation to provide a skeleton for the next step.

8.2.2 Enhancement with Run-time Information

Since the slicing and verification approach presented in this work requires information that is not available from the model files, we need to extract these parameters directly from the MATLAB/Simulink tool. To control and access the MATLAB/Simulink tool, we use the *MatlabControl*[5] library. MatlabControl is able to connect to the *Java MATLAB Interface* that resides in a Java Virtual Machine running within the MATLAB instance. To this end, MatlabControl provides proxy classes to execute procedure calls remotely using Java

[3]http://www.cqse.eu/en/products/simulink-library-for-java/overview/
[4]http://www2.cs.tum.edu/projects/cup/
[5]http://code.google.com/p/matlabcontrol/

Remote Method Invocation. However, the capabilities of the *Java MATLAB Interface* are limited to an *eval* method that can be used to execute MATLAB command, and *get* and *set* methods to read from or write values to the workspace. Note that the *get* and *set* methods only support primitive data types and arrays.

To obtain run-time information, e. g., the inferred data types of signals, we start MATLAB and load the model (and user-defined initialization files) into the MATLAB/Simulink tool and set it to the compiled state. Then, we traverse through all blocks within the skeleton. For each block, we execute a small script in MATLAB Code (M-code) using the `eval` command that extracts all parameters of a block and saves it to the workspace in a temporary variable. Then, we read the variable from the workspace and update the block parameters if necessary. At the end, we obtain a model in our intermediate representation that is enhanced with run-time parameters and ready for further preprocessing steps.

8.2.3 Preprocessing

Once all information for the model is collected, we apply a number of preprocessing steps to prepare the intermediate representation for the algorithms presented in Chapter 5 to Chapter 7. Besides the analysis of bus systems already presented in Section 5.4.1, we also flatten the subsystem within the model, we unfold branches in signal lines and resolve parameters from masked subsystems.

To ease the calculation of data flow trough the model, we flatten the hierarchical structure of the models w. r. t. data flow. To this end, we iterate through the model and for every subsystem found, we reconnect the signal lines connected to inputs and outputs of the subsystem block to the corresponding port blocks (e. g., `Inport` or `Enable` blocks). This has the benefit that in subsequent analyses, we can directly traverse the signal lines through the model. Furthermore, we analyze the model for `Goto` and `From` blocks, which are used to model signal flow without using lines to avoid line crossings. In our intermediate representation, we create virtual signal lines from every `Goto` block to its corresponding `From` block.

An important preprocessing step is the unfolding of signal lines. In Simulink models, a line can branch into two or more lines to model signal flow from one block to many blocks. In the model file, such branches are modeled as subelements of a line (or a branch element) specification that only contain a source block or a destination. To unfold these branches, we iterate over all lines in the model and create a new line for every branch.

A preprocessing step that is especially important for our translation approach is that we resolve the parameters of masked `Subsystem` and `Reference` blocks. In Simulink masks can be used to specify additional variables and constants within masks that are used by the (basic) blocks within the subsystem. For example, the composite `Counter Limited` block from the *Sources*

Block Set requires the specification of an "`upper limit`". This "`upper limit`" is then used as the mask parameter for the composite `Wrap To Zero` block, in which it is used as the `Threshold` parameter for a `Switch` block. To lift this information directly to the basic blocks, we iterate over all blocks in the model and check whether the use parameters from their masked parent subsystems. If they do, we traverse through the hierarchy to determine a concrete value for this parameter and replace the mask variables in the basic blocks.

8.2.4 Persisting the Parsed Model

Since the parsing process is rather costly, we decided to save the parsed model with all preprocessing steps applied to a database. The main idea is to parse only once, but to verify and slice multiple times using the intermediate representation stored in the database. To persist the parsed model, we currently use a *PostgreSQL*[6] database management system. The communication with the database is done using the *Java Persistence API*, more precisely the *Hibernate*[7] framework. With that, we can save and load the Java objects of the intermediate representation directly to and from the relational *PostgreSQL* database.

8.3 Slicing Engine

The slicing engine basically implements the techniques presented in Chapter 5 and the calculation of the CFG as presented in Section 6.4. Furthermore, it can visualize the slices by colorizing the blocks directly in the model using the *MatlabControl* library. Note that all figures depicting slices in Chapter 5 have been created with our MeMo tool suite.

8.4 Translation Engine

The translation engine implements the transformation from our intermediate representation to the formal specification as presented in Chapter 6 and Chapter 7. Our main idea for the transformation engine is to provide an implementation that can be easily extended by translation rules for further Simulink blocks and for additional mappings for data types, e. g., for the theory of floating point numbers as proposed by Rümmer and Wahl in [RW10].

To achieve that, the translation engine consists of three main components: A `BlockTranslatorFactory`, an `ArithmeticsMapping` component and a data model for the formal specification, the `BoogieModel`.

[6]http://www.postgresql.org/
[7]http://hibernate.org/

8.4.1 BlockTranslatorFactory

The `BlockTranslatorFactory` implements the factory pattern and returns an instance of a translator class for a certain block type. It is invoked during the traversal of the CFG for every block. The translator classes are required to extend the abstract class `AbstractBoogieTransformer`, which is instantiated with a block and an `ArithmeticsMapping`. To add a type specific translation, a concrete translator class needs to override the `calculateBoogie` method that takes a `BoogieCodeBlock` from the `BoogieModel` as parameter. Note that for all statements generated during translation, the expressions have to be created by `ArithmeticsMapping`.

8.4.2 ArithmeticsMapping

In our implementation, the `ArithmeticsMapping` component manages all data type mappings and arithmetic operations for the translation. The component is designed to be replaceable, as long as it implements the `ArithmeticsMapping` interface. Amongst others, this interface requires the implementation of a `calc` and a `getPreamble` method. Here, `calc` method takes an operator, a data type, and the left and right hand side and returns the corresponding Boogie2 expression. The `getPreamble` returns those parts of the preamble that are not specific to a certain block, e. g., data type bounds, functions for the over-approximation of behavior, and built-in functions from the Z3.

8.4.3 BoogieModel

The `BoogieModel` is our data model for a Boogie2 specification. It models a tree of the different elements of a Boogie2 program (e. g., `BoogieProcedure`, `BoogieLoopBlock`, `BoogiePredicateBlock`, `BoogieAssignmentBlock`, `BoogieStatement`). During translation, these model elements are created depending on the type of the Simulink block and passed to the block translator. All model elements implement the abstract class `BoogieCodeBlock` that provides methods to add statements, invariants, assertions, functions and variables. Block translators can use this methods to add their translation artifacts to the model. If necessary, these artifacts are passed through the hierarchy of the `BoogieModel` to their corresponding elements. For example, variables are passed to the `BoogieProcedure` and invariants are passed to the next `BoogieLoopBlock` in hierarchy. Note that manually specified invariants are added to the `BoogieModel` once all blocks are translated.

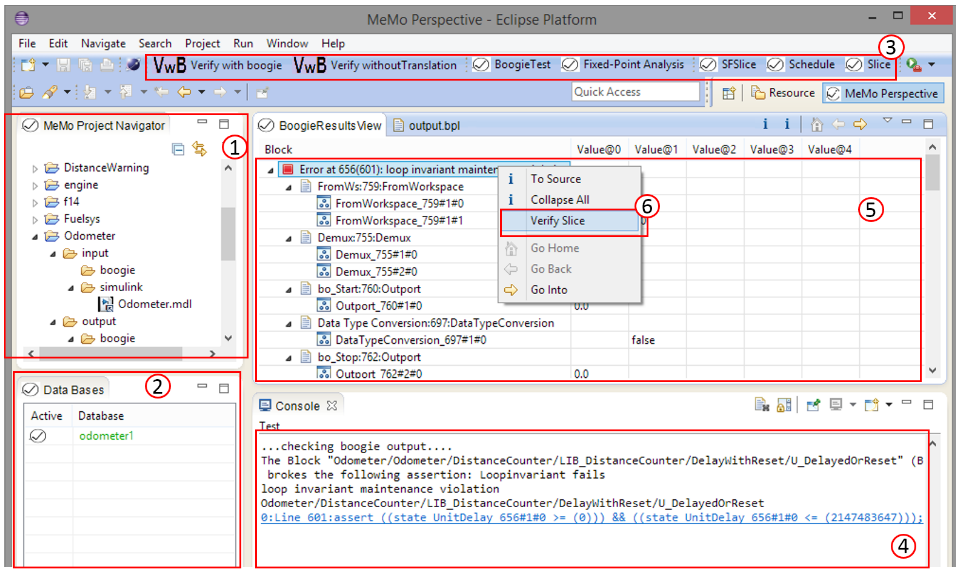
Finally, all elements of the `BoogieModel` have a `print` function that prints the Boogie2 statements represented by this element according to a k given as parameter that specifies the number of steps for k-induction. Moreover, every element calls the print function for each of its child elements. Hence, calling `print` on the `BoogieModel` returns the complete Boogie2 specification.

8.5 Verification Engine

Once a Simulink model is translated, the verification engine invokes the Boogie verification framework with the formal specification that has been written to a file. Furthermore, we configure the Boogie tool to write the counterexamples and the error traces into some files. When the verification is finished (i. e., the Boogie tool terminates), we analyze the trace for the verification results. If the verification succeeded or was inconclusive (i.e. the file containing the trace is empty), we directly report these results to the user. If the trace contains errors, we analyze the counterexamples and the error traces to post-process the results for the user. For that purpose, we extract the block and the type of the error from the assertion label. Additionally, we analyze the counterexample to match the variable assignments to the corresponding Simulink blocks. Due to optimizations within the Z3 theorem prover, assignments from the formal model that simply forward a value (e. g., port blocks, `Mux` and `Demux` blocks) are sometimes not contained in the counterexample. Hence, to obtain a counterexample that is consistent with the Simulink model, we calculate a backward slice for the block where the error occurred. Then, the variable assignments for the blocks are propagated backwards to the blocks contained in the slice that do not have a corresponding assignment in the reported counterexample.

8.6 Graphical User Interface of the MeMo Tool

To support our verification approach we have developed a graphical user interface based on the Eclipse RCP framework to guide the user through the verification process. Figure 8.3 depicts this user interface and in the following, we briefly present its core features.



Block	Value@0	Value@1	Value@2	Value@3	Value@4
Error at 656(601): loop invariant mainten					
FromWs:759:FromWorkspace					
FromWorkspace_759#1#0					
FromWorkspace_759#1#1					
Demux:755:Demux					
Demux_755#1#0					
Demux_755#2#0					
bo_Start:760:Outport					
Outport_760#1#0	0.0				
Data Type Conversion:697:DataTypeConversion					
DataTypeConversion_697#1#0		false			
bo_Stop:762:Outport					
Outport_762#2#0	0.0				

```
...checking boogie output....
The Block "Odometer/Odometer/DistanceCounter/LIB_DistanceCounter/DelayWithReset/U_DelayedOrReset" (B
 brokes the following assertion: Loopinvariant fails
loop invariant maintenance violation
Odometer/DistanceCounter/LIB_DistanceCounter/DelayWithReset/U_DelayedOrReset
0:Line 601:assert ((state UnitDelay 656#1#0 >= (0))) && ((state UnitDelay 656#1#0 <= (2147483647)));
```

Figure 8.3: The MeMo Tool

In our tool suite, we utilize the Project structure within Eclipse. A *MeMo Project* (1) consists of a Simulink model and a number of file artifacts that are automatically generated. The parsing process is triggered by a context menu entry of the model and provides a wizard that lets the user specify additional models and configuration scripts for the model. Our tool suite enables the user to work with multiple versions of a model and to select the database for a model version of interest (2). The verification and slicing techniques can be easily started using buttons (3). The verification results are displayed in a console (4) and if counterexamples have been reported, they are presented in a more human readable way in the verification results view (5). From this view, the user can also start the verification for a slice of the model using an entry (6) in the context menu for the displayed errors.

8.7 Summary

In this chapter, we have presented our implementation for the techniques of our verification approach: the *MeMo tool suite*. First, we have discussed the general architecture of the tool suite. After that, we discussed each component in more detail and especially focused on the parsing and the translation engine. We have presented some features of our parsing engine to show how we obtain all the information that is required by our verification and slicing techniques. Furthermore, we have discussed our implementation of the translation engine, which is designed to be easily extensible by adding new block translators and arithmetics. Finally, we have briefly presented the verification engine and our post-processing for the counterexamples before we have introduced our graphical user interface that guides the user through our verification process.

9 Evaluation

In this chapter, we present an evaluation of our slicing and verification techniques. We use the implementation presented in the previous chapter to show the practical applicability of our approach. The most important measures are performance and quality. For our verification approach, we evaluate quality by measuring the capability to detect errors and to show the absence of errors from the error classes of interest. To this end, we measure the number of actual errors in the model and the number of spurious counterexamples that are reported for verification runs. The latter may occur if the automatically generated invariants are too weak. However, with k-induction we are able to eliminate most of these spurious counterexamples fully automatically. To this end, we assess whether a model is verified fully automatically or user intervention in terms of additional manually specified invariants is required. For the slicing technique, we evaluate the quality by measuring the precision of the calculated slices. We have performed a set of experiments on a number of Simulink designs that comprise both synthetic benchmarks and models provided by industrial partners.

All experiments have been performed on a machine with an Intel (Dual)Core i7-4500U CPU 1.8 GHz and 8 GB main memory. For both techniques, we have measured the computational effort by monitoring the execution times of our implementation. To this end, we have instrumented the code with *System.currentTimeInMillis()* statements. This means that the smallest unit of measurement is milliseconds. For all measurements with run-times of a few milliseconds, it is possible that rounding errors might have occurred for execution time below one millisecond. Another factor that might have impacted the measured execution time is internal caching performed by the Java virtual machine (e. g., for the processing of counterexamples). Note that for the combination of the slicing and verification technique the slicing criterion is taken from a previous verification attempt. This means that the dependence analysis and the scheduling have already been done for the previous attempt and are reused. Finally, there are factors out of our control like the garbage collection of the Java virtual machine and the multi-threading of the Windows operating system that might have affected the reported execution times.

In the following sections, we start with a brief description of the Simulink designs used to evaluate our approach in Section 9.1. After that, we present our experimental results for our slicing technique in Section 9.2. Then, in Section 9.3, we present our evaluation of our verification approach. Finally, we summarize this chapter in Section 9.4.

9.1 Case Studies

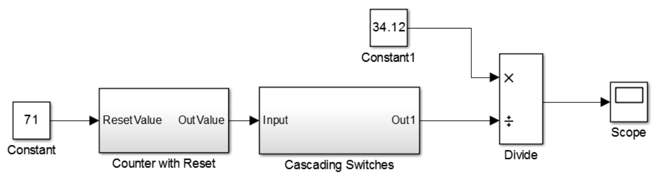
For the evaluation of our verification approach, we use three models: a carefully designed synthetic benchmark *TestModel*, a model of an *Odometer* and a design representing a distance warning (*DistWarn*) system for a car. The latter two models have been provided by our industrial partners within our *MeMo* project. For the experimental evaluation of our slicing technique, we use additional models: some synthetic models we have designed to contain certain language features like nested control flow or bus systems, which have been used as examples in the previous chapters, and a number of models taken from the set of examples shipped out with the Simulink tool suite.

In the next four subsections, we introduce the three case studies used for the evaluation of our verification technique and then briefly discuss the additional models.

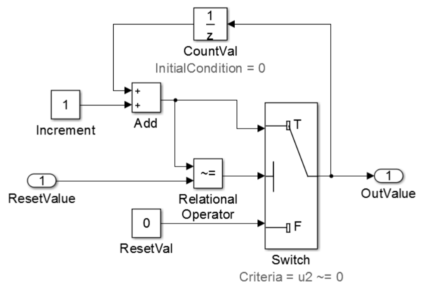
9.1.1 TestModel Design

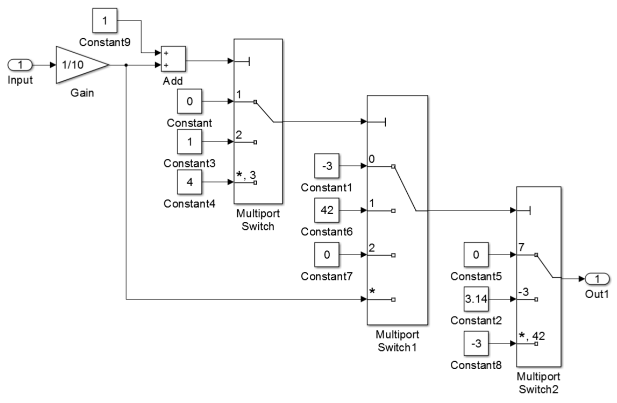
For a first evaluation of our approach, we have constructed a synthetic benchmark that features a number of possible and actual errors w. r. t. the error classes of interest, namely over- and underflows, range violations and division-by-zero errors. It consists of two subsystems: one is implementing a counter with a reset and the other contains a sequence of `MultiPortSwitch` blocks. Furthermore, the *TestModel* contains a `Product` block performing a division on the highest hierarchy level. Figure 9.1 depicts the model and its components. The model consists of a total of 31 blocks, where 6 blocks are virtual blocks (2 subsystems and 4 port blocks). All blocks except for the subsystems are basic blocks.

The "`Counter with Reset`" subsystem (Figure 9.1b) represents a simple counter that adds up a value up to a specified limit. We use an `UnitDelay` block to store the counting value that is initialized to 0 and incremented by 1 in every simulation step. The incremented value is compared to the reset value with a `RelationalOperator` block for inequality. As long as the values are not equal, the upper input (`T`) of the `Switch` block is forwarded and passed to the output of the subsystem. Otherwise, the value is reset. The contents of the second subsystem ("`Cascading Switches`") are depicted in Figure 9.1c. It uses the value supplied by the counter to select a value that is provided to the output of the subsystem. This output value is then used as divisor in the `Product` block (see Figure 9.1).



(a) Top Level View of the Model





(b) Counter with Reset Subsystem **(c)** CascadingSwitches Subsystem

Figure 9.1: Structure of TestModel

The *TestModel* is designed to contain at least one block for each error class of interest. In this model, over- and underflows may occur at both `Sum` blocks in the subsystems in Figure 9.1b and Figure 9.1c. Range violations may occur at the `MultiPortSwitch` blocks "`MultiportSwitch`" and "`MultiportSwitch2`" in the `CascadingSwitches` subsystem (Figure 9.1c). Finally, a division-by-zero error may occur at the `Product` block in Figure 9.1a.

Note that we have designed the *TestModel* model to be erroneous. If the counter exceeds the value of 19, the control input for "`MultiportSwitch`" becomes 3. Then, a range violation occurs[1] at the block "`MultiportSwitch2`" since the signal at its control input becomes 3, too. Furthermore, if the counter exceeds a value of 29, a range violation occurs at "`MultiportSwitch`" since its control input becomes 4. Finally, if the counter reaches 70, a division-by-zero occurs at the `Product` block since the default cases for "`MultiportSwitch`" and "`MultiportSwitch1`" are executed, which leads to a value of 7.0 at the control input of "`MultiportSwitch2`". Then, the block "`MultiportSwitch2`" outputs a 0, which is used as divisor in the `Product` block.

9.1.2 Odometer

Our second case study is a Simulink design modeling an odometer for a car. It has been provided by our project partner *Model Engineering Solutions GmbH*

[1]As mentioned in Section 6.5.9, floating point control signals are cast to integers in `MultiPortSwitch` blocks.

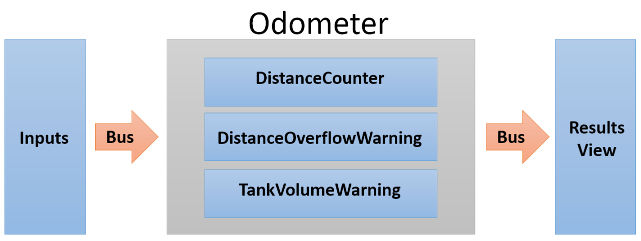


Figure 9.2: Architecture of the Odometer Model

within our *MeMo* project. It counts the distance traveled and outputs a warning signal in case of an overflow in the distance calculation. Furthermore, it also tracks the fuel consumption and outputs a signal once the tank volume falls below a given threshold. The architecture of this model is depicted in Figure 9.2.

The *Odometer* model consists of three components on the highest hierarchy level. The `Inputs` component provides values for simulation using `Constant` blocks and `Signal Builder` blocks that provide values from the workspace. `Signal Builder` blocks are composite blocks that internally use `FromWorkspace` blocks to generate signals. The Odometer component consists of three further subcomponents that realize the above mentioned tasks and contain a number of hierarchical subsystems. The communication between all above mentioned components is realized by bus signals.

The *Odometer* model makes heavy use of model referencing and user-defined libraries. Besides the main model (`Odometer.mdl`), it has references to three different external libraries that are used multiple times. Note that one model reference contains a Stateflow design, which models a state-based controller to output a modulated signal that can be used to control an optical or acoustic warning indicator. Furthermore, the *Odometer* model is configured with two scripts in MATLAB Code (M-code) to set up the paths to the referenced models and parameters used by various `Constant` blocks in the model.

With all model references included, the *Odometer* model consists of 190 blocks including 25 subsystems. It is a medium sized model that does not contain floating point arithmetics and is ready for code generation. Due to its explicit modeling of the overflow detection on the distance counter it does not contain any error w. r. t. the error classes of interest. Hence, it is perfectly suitable to show the absence of errors on a correct model.

9.1.3 Distance Warning (DistWarn)

Our third case study models a distance warning system that tracks the position of up to two vehicles driving ahead to warn the driver if the distance falls below a given threshold. It has been provided by our project partner *Berner & Mattner Systemtechnik GmbH* within our *MeMo* project. Figure 9.3 depicts the top level components of the model. It consists of a component

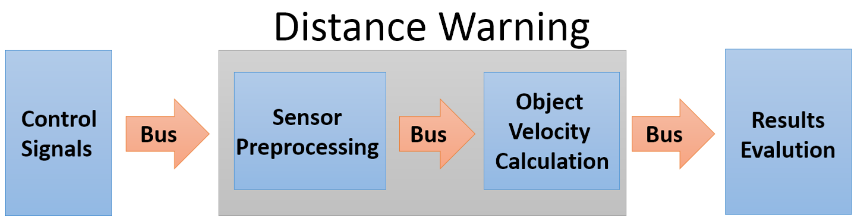


Figure 9.3: Architecture of the DistWarn Model

simulating the input signals, the actual distance warning component and some post-processing of the calculated results. Within the `Control Signals` component, the input signals are generated by a `SignalBuilder` block. Furthermore, `ManualSwitch` blocks are used to model sensor failures. The communication between all components is realized by bus signals.

Within the `DistanceWarning` component, the `Sensor Preprocessing` component analyzes the input signals for failures and performs a conversion of the velocity units first. After that, the `Object Velocity Calculation` uses radar signals to calculate the distance to some vehicles ahead. Furthermore, it calculates if there are multiple vehicles driving ahead and provides this information to its output. The calculated signals are then evaluated in the `Results Evaluation` component within a Stateflow design.

The *DistWarn* design uses model referencing and user-defined libraries as well as a number of composite blocks from the Simulink block library. It uses one small subsystem from an user-defined library and two references to a larger model (about 37 blocks), which is called *DistanceCalculation*.

Altogether, with all library links and model references resolved, the *DistWarn* model consists of 264 blocks. It is one of our largest case studies and contains floating point arithmetics, a number of multiplication operations and some Simulink constructs that make the verification challenging. Furthermore, the *DistWarn* design contains some errors that we discuss in detail in the evaluation section.

9.1.4 Additional Models for Slicing Evaluation

In addition to our three case studies for our verification approach, we also use a set of additional models that are used for the experimental evaluation of our slicing approach. Besides some models taken from the Simulink examples, we use a number of synthetic models that showcase some Simulink features and have been used as examples in the previous chapter. In the following, we briefly describe these models:

- **control_flow** The first model we use in our experimental evaluation is synthetic and features the hierarchical nesting of Conditional Execution Contexts (CECs).

Table 9.1: Overview of the Characteristics of the Case Studies

Model	Block Count	Line Count	Subsys. Count	Ref. Count	Hierarchy Depth
Models without large Feedback Loops or Bus Systems					
slicingexample	26	27	2	1	2
Benchmark	31	31	2	0	1
control_flow	57	47	8	1	3
clutch_if	103	112	11	0	4
Models with large Feedback Loops					
f14_control	64	66	5	1	3
climatecontrol	103	103	10	0	2
enginewc	114	120	13	0	3
aero_guidance	338	381	28	18	5
Models with Bus Systems					
busexample	19	18	0	0	0
Odometer	190	192	25	4	5
DistWarn	264	315	31	27	6

- **slicingexample** This is the model introduced to illustrate the original slicing technique in Section 5.3.
- **busexample** The model we have introduced in Section 5.4, to illustrate the slicing of bus systems.
- **aero_guidance** A Simulink example for a missile guidance system.
- **f14_control** A design from the Simulink examples modeling the flight control of a military aircraft.
- **climatecontrol** A Simulink example design modeling a climate control for a car.
- **clutch_if** A design from the Simulink examples representing the clutch of a car.
- **enginewc** A Simulink example modeling an engine of a car.

Table 9.1 briefly summarizes the characteristics of all 11 case studies. The model sizes range from 26 to 338 blocks. The line count describes the number of lines after all line branches are resolved. Furthermore, the layers of hierarchy introduced by subsystems varies from 0 to 6. The table also shows the number of subsystems and references in the model. Note that references are automatically replaced by the corresponding blocks from a (possibly user-defined) library or model reference. All models are loaded from the database in 8 to 40 seconds.

In Table 9.1, the case studies are partitioned into three groups whether they contain large feedback loops and bus systems or not. A model contains a feedback loop if a block depends on a value calculated in a previous simulation step. We consider a feedback loop as large if more than a third of the blocks in a model are part of a feedback loop.

In the following sections, we use the case studies presented above for the experimental evaluation of our slicing and verification approach. In the next section, we present and discuss the experiments for our slicing technique.

9.2 Slicing Evaluation

In this section, we present our experimental evaluation results for our slicing technique presented in Chapter 5. To evaluate the performance of our slicing technique, we measure the computational effort for the construction of the dependence graph and for the slicing algorithm itself. To evaluate the quality, we measure the precision of the slicing technique. As measure for the precision of our approach, we use the *average slice size* metric as introduced by Binkley et al. [BGH07]. The basic idea of the *average slice size* is to calculate the average size of all possible slices for a program. Analogously, we measure the computational effort for the slicing algorithm by using the *average slice computation time*. These measures are defined as follows:

Definition 9.1 (Average Slice Size (Simulink Model))**.** *For a Simulink model m that consists of a set of blocks $B = b_1, \dots b_n$, we define the average slice size $S_{AVG}(m)$ for m as*

$$S_{AVG}(m) = \frac{1}{|m|} \sum_{b \in B} \frac{|s(b)|}{|m|}$$

Here, $s(b)$ is the slice with the block b used as slicing criterion. $|s(b)|$ is the number of blocks within the slice and $|m|$ is the number of blocks within the original model.

Note that this measure also includes subsystems. Since we flatten the model in our intermediate representation, we use the corresponding port blocks of the subsystem as slicing criteria instead of the actual subsystem block.

Definition 9.2 (Average Slice Computation Time (Simulink Model))**.** *For a Simulink model m that consists of a set of blocks $B = b_1, \dots b_n$, we define the average slice computation time $T_{AVG}(m)$ for m as*

$$T_{AVG}(m) = \frac{1}{|m|} \sum_{b \in B} T(s(b))$$

Table 9.2: Experimental Results for the Basic Slicing Technique

Model	**MSDG (ms)**	**Forward**			**Backward**		
		T_{AVG} (ms)	**S_{AVG} (%)**	**σ (%)**	**T_{AVG} (ms)**	**S_{AVG} (%)**	**σ (%)**
No Feedback Loops, No Bus Systems							
Benchmark	< 1	<0.01	50.36	26.85	<0.01	50.36	30.54
slicingexample	< 1	<0.01	51.78	26.50	<0.01	52.37	31.74
control_flow	< 1	<0.01	33.55	25.08	<0.01	33.70	23.77
clutch_if	< 1	<0.01	69.98	30.31	0.06	68.60	32.78
Average			51.42			51.26	
With large Feedback Loops							
f14_control	< 1	<0.01	53.62	27.45	0.05	53.67	33.13
climatecontrol	< 1	0.06	57.23	24.72	0.03	57.16	37.40
enginewc	< 1	0.06	83.51	22.56	0.09	83.53	27.80
aero_guidance	< 1	0.11	82.15	31.10	0.08	82.12	20.22
Average			69.13			69.12	
With Bus Systems							
busexample	< 1	<0.01	36.57	21.50	<0.01	36.57	30.43
Odometer	< 1	0.03	31.69	24.52	<0.01	31.76	23.59
DistWarn	< 1	0.05	28.83	19.34	0.04	28.90	28.57
Average			32.36			32.41	
Total Avg.			**52.66**			**52.61**	

For the slice $s(b)$ with the block b used as slicing criterion, $T(s(b))$ is the time needed to calculate $s(b)$.

For the experimental evaluation of our slicing technique, we have calculated the above measures for each of the case studies introduced in Section 9.1. In the following sections, we first present and discuss the evaluation of our base slicing approach presented in Section 5.3. After that, we present the evaluation result for the routing-aware approach presented in Section 5.4.

9.2.1 Evaluation of Our Basic Slicing Technique

Table 9.2 shows the experimental results for our basic slicing technique for all 11 models. For each model, the table shows the time needed to construct the MATLAB/Simulink Dependence Graph (“*MSDG*”). Furthermore, it shows the average slice size ("S_{AVG}"), the average slice computation time ("T_{AVG}") and the standard deviation ("σ") for forward and backward slicing. The latter indicates the mean distance between the average slice size and the actual slice sizes.

For all case studies, the dependence analysis and the construction of the MSDG is done in less then one millisecond. Furthermore, in average we only need a fraction of a millisecond to calculate the slice with the reachability analysis. Hence, we can calculate slices for each model in less then 2 milliseconds.

With the basic slicing approach, we have obtained an average slice size around 52.6% for all case studies for both forward and backward slicing. This means, with our basic slicing approach we are able to reduce the models by half of their size in average. Note that the average slice size strongly depends on some model characteristics. In our case studies we have identified two factors that have a high impact on the slice size: feedback loops and control flow. Feedback loops are cycles in data flow that result in strongly connected components in the MSDG. For every block represented by a node in the strongly connected component, we obtain the same slice. Furthermore, large or many conditional execution contexts introduce more control dependences that can lead to an increase of the average slice size.

However, for our industrial case studies (*DistWarn* and *Odometer*), we are able to achieve an average reduction in the slice size of more than 68%. These results render the uses of slicing as an automatic model reduction technique prior to the verification promising. Note that we obtain this average slice size of less than a third for the industrial cases studies without resolving the bus systems. Since we require tho models for verification to be executable, we can also use our routing-aware slicing approach to obtain even more precise slices by resolving the bus signals in the models. Hence, we have evaluated our routing-aware slicing approach on these case studies and present the results in the next subsection.

9.2.2 Evaluation of the Routing-aware Slicing Technique

Since Simulink models, especially our industrial cases studies, often contain bus systems, we can obtain more precise slices using our routing-aware approach. However, our routing-aware approach requires the models under test to be executable since it uses run-time information about subsystems to construct routing-aware MSDG. Table 9.3 shows the experimental results for our routing-aware approach. Note that our routing-aware approach only differs from our basic technique if bus systems are present in the model. Hence, Table 9.3 only shows the results for the third group of Table 9.2.

With our routing-aware approach, we have been able to reduce the average slice size for the case studies containing bus systems by 9.27 percentage points for forward and 9.69 for backward slicing. This is an increase in precision of more than 28% compared to the basic approach.

Table 9.3: Experimental Results for the Routing-aware Slicing Technique

Model	MSDG	Forward			Backward		
	(ms)	T_{AVG} (ms)	S_{AVG} (%)	σ (%)	T_{AVG} (ms)	S_{AVG} (%)	σ (%)
busexample	< 1	<0.01	26.04	15.32	<0.01	24.93	19.68
Odometer	< 1	0.02	24.2	16.04	0.03	24.15	20.51
DistWarn	< 1	0.02	19.05	15.65	0.04	19.09	21.40
Average			23.10			22.72	
Total Avg.			**50.13**			**49.97**	

To assess the the average slice size calculated with our approach, we compare them to the average slices sizes reported by Binkley et al. [BGH07] in a large scale empirical study on a number of programs written in imperative programming languages. In this study, they reported an average slice size of 26.1% for forward slicing and 26.8% for backward slicing. In comparison, our approaches yield average slice sizes for all models of 52.6% for the basic and 50.13% for the routing-aware approach. However, the programs used in [BGH07] were not from the domain of controller software. Hence, it is very likely that they do not contain large feedback loops like our case studies.

Note that our routing-aware slicing technique yields an average slice size of less than 25% for the industrial case studies. Hence, it significantly reduces the complexity of the models.

9.3 Verification Evaluation

In this section, we present our experimental evaluation results for our verification approach. We have verified the three case studies *TestModel*, *Odometer* and *DistWarn* using our approach. To evaluate the performance of our approach, we have measured the computational effort for the transformation into the formal specification and for the verification of this specification with the Boogie verification framework. To evaluate the quality of our verification technique, we measure its capabilities to detect errors and to verify the absence of errors. To this end, we apply our verification process presented in Section 4.3 to each case study. For each step in our process, we measure the number of counterexamples obtained by a verification run and distinguish these into actual errors and spurious counterexamples. Furthermore, we evaluate the capability of our technique to eliminate spurious counterexamples. In the following subsections, we present our experimental results for each of our three case studies.

9.3.1 TestModel

As our first case study we use the *TestModel* design depicted in Figure 9.1. Since the original *TestModel* contains three errors, we have also used three slightly modified versions of the model (*TestModel_1_err*, *TestModel_corr* and *TestModel_lim_20*). More precisely, we use *TestModel* and *TestModel_1_err* to demonstrate the capabilities to detect errors, and *TestModel_corr* and *TestModel_lim_20* to show the absence of errors with our verification approach. The modifications for each model are discussed in detail in the subsequent paragraphs.

Table 9.4 shows the evaluation results for our first case study. For each version of *TestModel*, the table shows verification runs using inductive invariant checking, k-induction and the combination of slicing and k-induction. Inductive invariant checking is indicated in column k with a "–", for k-induction it shows the value of k. Furthermore, the table shows the lines of code ("LOC") and time required to generate the Boogie2 specification in milliseconds. The latter includes the dependence analysis to obtain the CECs, the calculation of the CFG, and the actual translation of the model into a file. The column "Checks" indicates the number of assertions generated for the error classes of interest. The number comprises all assertions that have composed into the specification. Since for k-induction an assertion is added multiple times in the base case, the number in brackets indicates the number of distinct assertions. Finally, the table shows the verification time in seconds, which includes post-processing of counterexamples, and the number of counterexamples reported by the Boogie tool. All results are averaged over five runs.

The first set of results in Table 9.4 shows three verification runs on the original model. With inductive invariant checking, we have obtained three counterexamples. Two are reported for both erroneous `MultiPortSwitch` blocks. The third is a spurious counterexample for an overflow at the `Sum` block in the counter subsystem. In the subsequent verification runs with k-induction using $k = 2$ and $k = 100$, we have only obtained one counterexample for the range violation at the "`MultiportSwitch2`" block. Since for $k = 100$ the reported error is in the base case, we can be sure that this is not a spurious counterexample and we have detected an error. Note that this error shadows other errors in the model when using k-induction since it occurs once the counter reaches 20 while the other two errors require the counter to reach 30 and 70.

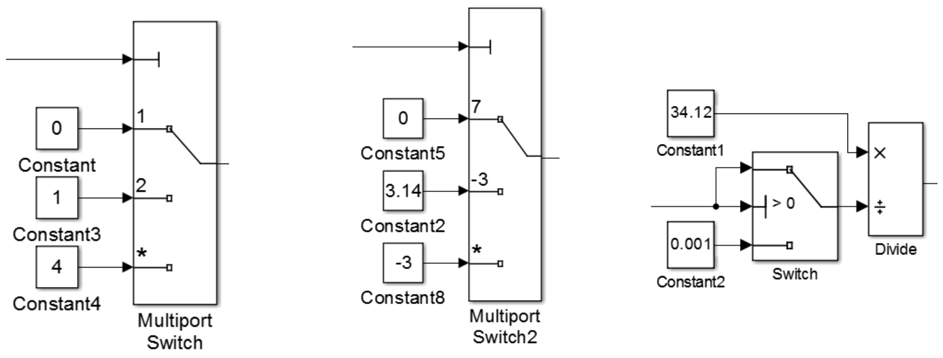
We have modified the original *TestModel* model and fixed both range violations in the model *TestModel_1_err* by modeling the default case explicitly as depicted in Figure 9.4a and Figure 9.4b. The second group of results in Table 9.4 depicts the verification runs on *TestModel_1_err*. We have obtained two counterexamples with inductive invariant checking. One is for the division-by-zero that occurs at the `Product` block and the other is, again, the spurious counterexample for the overflow at the `Sum` block. Then, with k-induction ($k = 2$), we have still obtained two counterexamples. By trying larger ks, we have been able to eliminate one spurious counterexample with $k = 49$. Since,

Table 9.4: Evaluation Results for the *Testmodel* Model

Model	k	LOC	Trans. Time (ms)	Checks	Verif. Time (s)	Counter-examples
TestModel	–	301	15	7(7)	0.99	3
TestModel	2	901	15	21(7)	0.97	1
TestModel	100	27361	221	707(7)	4.56	1
TestModel_1_err	–	395	16	5(5)	0.90	2
TestModel_1_err	2	871	16	15(5)	0.97	2
TestModel_1_err	30	8,095	65	155(5)	25.17	2
TestModel_1_err	49	12,997	94	250(5)	118.04	1
TestModel_1_err Slice: 9/31 Blocks	49	4,107	31	50(1)	1.22	1
TestModel_1_err Slice: 9/31 Blocks	57	4,747	31	58(1)	1.16	0
TestModel_corr	–	319	16	5(5)	0.90	1
TestModel_corr	2	969	15	15(5)	0.99	1
TestModel_corr	57	16,919	124	305(5)	32.94	0
TestModel_corr Slice: 9/33 Blocks	57	4,747	31	58(1)	1,16	0
TestModel_lim_20	–	301	16	7(7)	1.21	3
TestModel_lim_20	2	901	16	21(7)	0.86	0

a range of the counting variable between 70 and 79 leads to the division-by-zero, $k = 49$ is suitable to eliminate the counterexample for the overflow $(79+49 = 128)$ due to shadowing. However, this verification run takes almost 2 minutes. Instead of verifying the whole model with $k = 57$, we can also show that the overflow error is spurious in less than 2 seconds by verifying only the slice for the corresponding `Sum` block with $k = 57$. Note that for the verification of the slice with $k = 49$ the spurious counterexample is still reported since the division-by-zero error does not shadow it any more. However, it can be useful to advance to the slicing step in our verification process earlier to eliminate spurious counterexamples with less computational effort.

For the model *TestModel_corr*, we have eliminated the division-by-zero from *TestModel_1_err* by adding a `Switch` and a `Constant` block such that a constant value is used if the `CascadingSwitches` subsystem outputs a 0 (see Figure 9.4c). Although the model is now free of errors w. r. t. the error classes



(a) Fixed "`MultiportSwitch`" **(b)** Fixed "`MultiportSwitch2`" **(c)** Fix to avoid Division-by-Zero

Figure 9.4: Changed Model Elements for *TestModel_1_err* and *TestModel_corr*

of interest, the spurious counterexample for the overflow is still reported for inductive invariant checking and k-induction ($k = 2$). However, we can verify the absence of errors with $k = 57$ in less than 35 seconds. Again, the use of slicing can greatly reduce the computation time to eliminate the spurious counterexample.

For the model *TestModel_lim_20*, we have modified the `Constant` block that provides the limit to the counter subsystem and set it to 20 instead of 71. With this configuration, neither the range violations nor the division-by-zero can occur. When attempting to verify the model *TestModel_lim_20* with inductive invariant checking, three spurious counterexamples are reported for the `Sum` block and both `MultiPortSwitch` blocks. However, with k-induction ($k = 2$) we can show the absence of errors for this model in less than one second.

Discussion

Before we present the experimental evaluation for the industrial case studies, we discuss results for all versions of the *TestModel* model. In all verification runs, the formal specification has been generated in less than one second. The computational effort for the verification increases with the size of the specification. However, applying our slicing technique to the models has greatly reduced the computational effort especially for the elimination of spurious counterexamples.

Note that the counter subsystem in the *TestModel* model is intentionally modeled in a way that stresses the verification with our inductive techniques by using an equality to reset the counter. Since we only generate invariants from the data type bounds, the arbitrary start state for the verification includes assignments to the counter variable that are larger than the actual limit of the counter and would lead to an overflow. To verify that no overflow can occur, we have to choose k such that we reach the assignment to the counter variable that would cause the overflow in k steps from the limit of the counter. Since the `Sum` block has the data type *int8* and counts up to 71, we need at

Table 9.5: Evaluation Results for the *Odometer* Model

Model	k	LOC	Trans. Time (ms)	Checks	Verif. Time (s)	Counter-examples
Odometer	–	1,042	59	8(8)	1.37	1
Odometer	2	3,778	75	24(8)	1.47	0

least a k of 57 ($71+57=128$) to verify that no overflow is possible. However, a manually added invariant that the counter variable is always smaller than the limit can easily eliminate the spurious counterexample. Furthermore, if $\leq$, $\geq$, $<$ or $>$ are used instead of equality and inequality, it is possible to show that the counterexample is spurious with $k=2$ since every assignment to the counter variable greater than the limit is reset in the first step.

Note that for *TestModel_lim_20* no spurious counterexamples are reported for k-induction with $k=2$. This is the case since the ranges for valid control inputs of the "`MultiportSwitch`" block are propagated to the counter variable.

9.3.2 Odometer

The second case study we have used for the evaluation of our approach is the *Odometer* model. Table 9.5 shows the evaluation results obtained by applying our verification approach to the model. The *Odometer* model does not contain any `MultiPortSwitch` blocks. All of the 8 checks are generated either for potential over- and underflows (6) or for potential division-by-zero errors (2).

As shown in Table 9.5, we have obtained one counterexample[2] during the first verification run with inductive invariant checking. The counterexample has been reported for an *invariant maintenance error* for a `UnitDelay` block within a referenced block from a library. The cause for this error is a type mismatch for a signal connected to the library block, which is an unsigned integer of width 32, and output of the `UnitDelay` block within the library, which is a signed 32 bit integer. However, this counterexample is spurious since the signal connected to the library block can never exceed a certain bound (10000) that is specified in the configuration file and is always positive. Hence, we have performed a verification run using k-induction with $k=2$. With that, we have been able to verify the absence of errors w. r. t. the run-time errors of interest in less than two seconds for the *Odometer* model.

[2]Note that this is auctually the error displayed in Figure 8.3.

Table 9.6: Evaluation Results for the *DistWarn* Model

Model	k	LOC	Trans. Time (ms)	Checks	Verif. Time (s)	Counter-examples
DistWarn	–	1,919	109	20(20)	3.32	17
DistWarn	2	7,437	150	60(20)	11.64	15
DistWarn_corr	–	1,927	94	20(20)	1.85	8
DistWarn_corr	2	7,716	146	60(20)	9.30	6
DistWarn_corr	4	11,801	203	100(20)	171.35	6
DistWarn_corr	8	20,649	275	180(20)	1,013.03	6
DistWarn_corr	9	22,861	283	200(20)	15,298.60	6
DistWarn_corr	10	25,073	320	220(20)	> 12h	——
DistWarn_corr *Slice*$_1$: 87/264 Bl.	11	14,677	227	72(6)	152.56	0
DistWarn_corr *Slice*$_2$: 87/264 Bl.	11	14,677	224	72(6)	357.78	0
DistWarn_corr *Slice*$_3$: 87/264 Bl.	21	34,981	485	220(10)	> 12h	——
DistWarn_corr (4 Invariants)	–	1,939	97	20(20)	1.95	10
	2	7,437	150	60(20)	10.53	0
Distance-Calculation	11	4,009	41	39(3)	1.47	0

9.3.3 DistWarn

The third case study we have used to evaluate our verification approach is the *DistWarn* system. Although it is the largest of the three case studies, it neither contains `MultiPortSwitch` nor `Product` blocks. However, it contains a number of `Sum` and `DiscreteIntegrator` blocks, where over- and underflows can occur. Since the original *DistWarn* model contains a number of errors, we have also used a modified version *DistWarn_corr* for our evaluation. In the latter, we have fixed the errors from the original *DistWarn* model.

Table 9.6 shows the experimental results obtained for the *DistWarn* model. First, we have verified the original model (*DistWarn*) with inductive invari-

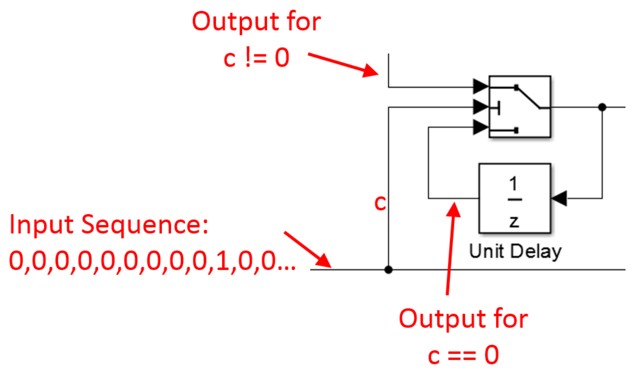


Figure 9.5: Source for the spurious Counterexamples

ant checking. With that, we have obtained 17 counterexamples for over- and underflows at 11 different blocks. With a subsequent verification run using k-induction, we have been able to eliminate two spurious counterexamples for two blocks.

When inspecting the counterexamples, we discovered that many of them correspond to actual errors in the *DistWarn* model. More precisely, the model contains four `DiscreteIntegrator` blocks that are neither reseted nor limited. Hence, over- or underflows can (and will) occur at these blocks. This accounts for six of the 15 counterexamples in Row 2 of Table 9.6. Furthermore, two `Sum` blocks can overflow, since they use output values of two of the `DiscreteInte-grator` blocks. This accounts for three of the 15 counterexamples. However, the remaining six counterexamples are spurious and occur at three blocks in the model.

To investigate the spurious counterexamples, we have modified the four `DiscreteIntegrator` blocks of the original *DistWarn* model. To this end, we have added a saturation to $[-1000, 1000]$ to the output values for all integrator blocks. The evaluation results for the modified model (*DistWarn_corr*) are shown in Table 9.6. With inductive invariant checking, we have obtained 8 counterexamples for over- or underflows for 6 blocks in the model. With k-induction ($k = 2$) we have been able to eliminate two spurious counterexamples. However, our tool still reports 6 counterexamples for over- and underflows at 3 blocks, When inspecting the counterexamples, we discovered that all counterexamples are spurious.

The source of the spurious counterexamples is a certain combination of blocks in the *DistanceCaluclation* model, which is referenced twice in *DistanceWarning*. Figure 9.5 depicts an excerpt of the *DistanceCaluclation* model, which is the source of four spurious counterexamples. Here, the state of the `UnitDelay` is only changed every 10 simulation steps. This means for our verification approach that we have to unroll the model with k-induction such that the `UnitDelay` is at least updated once. Since the automatically generated invariants only limit the state variable of the `UnitDelay` block to the data type bounds (similar to the counter subsystem in our *TestModel* design), a spurious counterexample is reported otherwise. From the counterexamples reported for *DistWarn_corr*, four are a result of calculations that directly follow the blocks from Figure 9.5. However, due to another `UnitDelay` block being involved in the calculation, we need $k = 11$ to eliminate these spurious counterexamples. Furthermore, in the actual `DistanceCaluclation` model, the construct from Figure 9.5 is used again after those calculations. Hence, we need $k = 21$ to eliminate the other two spurious counterexamples.

Note that the entire *DistWarn_corr* model is too complex to be verified for $k = 21$ (and even for $k = 11$) in reasonable time. We have not been able to finish the verification run for $k = 10$ in 12 hours. However, with our slicing technique and k-induction, we have been able to eliminate four of the six spurious counterexamples. To this end, we have used our slicing technique to calculate slices for two of the blocks that introduce spurious counterexamples, each contained in one of the two model references to the `DistanceCaluclation` model. Note that both slices have the same number of blocks. Although each reference to the `DistanceCaluclation` model uses a different sensor, the preprocessing for both values is the same. With slicing, we were able to eliminate four spurious counterexamples (2 with $Slice_1$ and 2 with $Slice_2$) in less than three minutes for the first model reference to `DistanceCaluclation` and in less than six minutes for the second reference. However, we were not able to eliminate the two remaining counterexamples in a reasonable amount of time with fully automatic techniques (see the $Sice_3$ with slicing for the block that causes the counterexamples and $k = 21$).

To show that no further errors exist in our *DistWarn_corr* model, we have added some invariants manually. More precisely, we have added an invariant for each `UnitDelay` block in each occurrence of the above discussed model elements. The invariants limit the state variables for the `UnitDelay` blocks to more precise bounds corresponding to the limits of the `DiscreteIntegra-tor` block. When attempting to verify the model using these invariants and inductive invariant checking, we have obtained 11 counterexamples. The two additional counterexamples are *invariant maintenance errors* reported for two of the manually added invariants. However, with k-induction and $k = 2$ we have been able to show the absence of errors w. r. t. the error classes of interest for the *DistWarn_corr* model in less then eleven seconds.

Note that we can eliminate the spurious counterexamples in the *Distance-Calculation* in less than 2 second by applying our technique directly to the referenced model. Although this does not help for elimination of the remaining spurious counterexamples in the *DistWarn_corr* model, it may generally indicate a starting point for future work using forward slicing or contract-based verification. We discuss this in more detail in Chapter 10.

9.4 Summary

The experimental evaluation results presented in this chapter demonstrate the performance, the capabilities to detect and to show the absence of errors w. r. t.

the class of common run-time errors, and the practical applicability of our verification approach.

For our slicing technique, the performance evaluation shows that the computational effort for the dependence analysis and the construction of the MSDG is low. Even for the largest of our case studies, a slice is calculated in a fraction of a second. We expect the slicing technique to scale well with larger models, since the effort will increase linearly with the size of the models, the number of bus signals and propagated CECs. Note that the set of models we have used for the experimental evaluation of our slicing technique might not be representative since the set is rather small and contains models with few blocks and synthetic data. Furthermore, most case studies are taken from the Simulink examples and are rather showcases for the capabilities of the Simulink tool suite than real world (controller) models. However, based on the results for our two industrial case studies we would rather expect a decrease in the average slice size in a larger scale empirical evaluation with industrial case studies.

The results for the computational effort of our verification technique meets our expectations. Although the worst-case complexity of our scheduling algorithm is exponential with the number of blocks, the transformation time scales well for the case studies. All case studies have been transformed in less than a half second. Regarding the verification of the translated models, our performance evaluation indicates that the computational effort exponentially increases as expected with the size of the specification and, hence, the complexity of the verification condition. However, with the application of our slicing technique, we were able to reduce the complexity and to obtain verification results in considerably less time. The evaluation demonstrates that we are able to detect and to show the absence of common run-time errors. We have detected all errors in our synthetic *TestModel* design and even found some design flaws in the *DistWarn* model. Furthermore, we have shown the absence of errors for the *Odometer* model and that k-induction is well suited for the elimination of spurious counterexamples. However, it may require high *ks* to compensate for weak invariants.

We have demonstrated the practical applicability of our verification framework by applying our proposed verification process to the case studies. We were able to fully automatically translate and verify two of the case studies and have needed only little manual intervention for the third. We have demonstrated that our slicing technique, which provides an average reduction of the models of 50%, improves the scalability of our approach by greatly reducing the time needed to show certain properties. Finally, we have demonstrated that for larger industrial models like our third case study, where a fully automatic translation and verification is not possible, our approach can be applied with only a little manual intervention.

10 Conclusion and Future Work

In this chapter, we give a conclusion of the work presented in this thesis. We first summarize the developed techniques and obtained results for our verification approach for Matlab/Simulink models in Section 10.1. Then, in Section 10.2, we discuss the results of this thesis with respect to the objectives we have defined in the introduction of this thesis (Section 1.2). Finally, we give an outlook on future work in Section 10.3.

10.1 Results

In this thesis, we have presented our approach for the automatic verification of discrete-time Matlab/Simulink models. Our approach enables the fully automatic verification of embedded controllers modeled in Matlab/Simulink for the absence of a number of important classes of run-time errors. It is based on a translation of Matlab/Simulink models into a formal representation, the Boogie2 intermediate verification language, which enables the use of the Boogie verification framework for Matlab/Simulink models. The basic idea of our approach is to use a combination of different inductive verification strategies and automatic model reduction techniques to obtain a fully automatic and comparatively scalable verification framework for Matlab/Simulink.

A prerequisite for the application of formal verification techniques is to obtain a formal semantics for Matlab/Simulink models. Since the semantics for Simulink is only given informally by the Matlab/Simulink documentation, we have presented a mapping to the formally well defined Boogie2 intermediate verification language. We use the sequential simulation semantics that is given to the models by the discrete, fixed-step simulation engine (solver) to translate the models into a semantically equivalent Boogie2 program. The basic idea of this mapping is to represent the simulation loop of the solver as a loop within the Boogie2 specification. Furthermore, we determine the execution order of all blocks in a given model according to the scheduling rules defined in the Matlab/Simulink documentation. A main advantage of our mapping is the preservation of mechanisms for the conditional execution of blocks within Simulink models. For this mapping, we have presented a fully automatic translation for a subset of basic Simulink blocks into the corresponding Boogie2 statements. The translation is modular and can easily be extended by further block types in the future.

For the verification of the transformed models, we have presented the adaption of two inductive verification strategies for MATLAB/Simulink: inductive invariant checking and k-induction. While inductive invariant checking scales better, k-induction provides a higher degree of automation since it requires less strong invariants at the cost of more computational effort. For both strategies, verification goals for the run-time errors of interest and loop invariants are fully automatically generated during the translation. Depending on the selected verification strategy, we automatically create a corresponding Boogie2 program that comprises the formal model obtained from the automatic translation, the automatically generated verification goals, and the automatically generated invariants. The resulting Boogie2 programs are used as input to the Boogie tool to, for example, show the absence of errors for a Simulink model or to obtain counterexamples for potential errors.

To address the problem of increasing computational effort with k-induction, we have presented an automatic model reduction technique based on slicing. To this end, we have introduced a novel slicing technique for Simulink models. This technique is based on a dependence analysis for MATLAB/Simulink models according to the simulation semantics to determine data and control dependences. The calculated dependences are used to compute a slice for the model for a given block as slicing criterion. Our slicing technique automatically calculates static slices that respect all possible executions that can affect a criterion. Hence, the slices calculated with our approach can be verified by applying our automatic translation directly to the slice. The resulting slices are significantly smaller than the complete model. Thus, the computational effort of formal verification is also noticeably reduced.

In our approach, we have presented a process that assists the efficient use of the particular verification strategies and abstraction techniques. The general idea of this process is to start with the best scaling strategy (inductive invariant checking). Subsequently, if the automatically generated invariants are too weak, k-induction is used with increasing k. Finally, if the computational effort for the verification of the entire model is too big, slicing is used to reduce the complexity of the verification for a specific block of interest in the model.

To show the practical applicability of our approach, we have implemented all techniques presented in this thesis in our *MeMo* tool suite. The *MeMo* tool suite is based on the Eclipse Rich Client Platform (Eclipse RCP) and provides a graphical user interface that guides the developer through our verification process. To evaluate the applicability, we have used three case studies: a simple model carefully designed to contain a number of run-time errors and model elements to stress the verification approach (*TestModel*), and two industrial case studies. The first is a model for an *Odometer* and the second industrial design models a distance warning system (*DistWarn*) that tracks the distance to vehicles driving ahead. The evaluation results show that we have been able to translate and verify all models fully automatically. With our approach, we have been able to detect run-time errors in the *TestModel* and *DistWarn* design and to show the absence of errors for the *Odometer*. Furthermore, we have used a number of modified versions of *TestModel* and *DistWarn* where we have successively removed errors in the design to demonstrate the capabilities of our process to automatically eliminate spurious counterexamples. We have been able to fully automatically eliminate all spurious counterexamples except for two in the *DistWarn* design. To show the absence of errors for the complete corrected version (*DistWarn_corr*), we had to add only four invariants manually. Overall, the degree of automation in our verification framework is very high and scalable compared to other approaches for the formal verification of Matlab/Simulink models.

10.2 Discussion

In this section, we discuss our approach w. r. t. the objectives that we have defined in Section 1.2.

The aim of this thesis was to provide an automated formal verification framework for Matlab/Simulink models. To apply formal verification, a formal semantics for the only informally defined semantics of Matlab/Simulink models was required. In our framework, we obtain the formal semantics with an automatic translation of a relevant subset of blocks for discrete-time Matlab/Simulink models into a formal intermediate representation. More precisely, we translate the models according to the informally specified simulation semantics into the formally well defined Boogie2 intermediate verification language. To this end, we translate the instances of each supported block type according to their discrete, fixed-step simulation semantics and their parameter configuration. Our translation respects the mechanisms for conditional execution of blocks and, hence, is more precise w. r. t. control flow than other existing approaches.

Furthermore, we have defined the following criteria for our framework:

- **Automation** The first criterion for our framework was to reach a high degree of automation for the verification approach such that no or only little user (inter)action is required to verify a model. In our framework, we have presented a fully automatic translation of Matlab/Simulink models into the Boogie2 language that does not require any user interaction or annotations. Furthermore, verification goals and invariants are generated fully automatically for the models. The resulting formal Boogie2 program is verified automatically with the Boogie verification framework. Our experimental results have shown that we were able to verify two of our three case studies fully automatically. In our verification process, user interaction is only required for selecting a k, for selecting a slicing criterion and, in some cases, to specify manual invariants. The latter are automatically integrated during the automatic generation of the Boogie2 program no matter which verification strategy or k is chosen.

- **Coverage** The second criterion for our approach was the coverage of a substantial subset of the blocks from the Simulink block library. In our framework, we have presented translation rules for more than 40 frequently used *basic* blocks. These basic blocks are sufficient to translate all of our case studies. Furthermore, our framework automatically integrates model and library references. Hence, we are able to translate further composite blocks from the Simulink block library like `Signal Builder`, `Counter (limited)`, `Difference` or `Wrap to Zero` blocks, which are all used by our case studies. To provide extensibility, our approach uses a modular block-by-block translation such that rules for further block types can easily be added in the future.
- **Verification Goals** Besides an automatic transformation, we also required our framework to automatically generate verification goals for important run-time errors. In our framework, we automatically generate checks for such errors during the translation of blocks. These checks are used to detect or to show the absence of certain errors. The generated checks depend on whether a certain type of error may possibly occur at a specific block type. Our framework currently supports the automatic generation of checks for the following error classes of interest: *division-by-zero*, *range violations* and *over-* and *underflows*. The checks are automatically weaved into the formal representation for a given Simulink model.
- **Scalability** We required our translation and verification approach to scale well with the size of the models and their state space. Although the worst case complexity of our scheduling algorithm is exponential in the number of blocks in an execution context, this worst case hardly occurs in practice. Moreover, the complexity of the other algorithms used in the translation and slicing are linear in the number of blocks, of resolved signals, of execution contexts, or a combination of these. The evaluation results show that the impact of the scheduling algorithm on the translation time is negligible. Hence, we expect the translation time to scale well in most practical cases. To achieve scalability of the verification, we use inductive techniques together with SMT-solving. While inductive verification techniques scale well compared to fully automatic techniques like model checking, they are generally less automated. To overcome this problem, we increase the degree of automation by automatically generating the invariants to verify models for the error classes of interest and by using k-induction. Furthermore, our framework provides a slicing technique to automatically reduce the complexity of models w. r. t. a specific point in the model. With this technique we are able to split the verification task into a number of subtasks with less computational effort.
- **Comprehensibility** In addition to scalability, we required our framework and especially the verification results to be comprehensible. We

fulfill this requirement since our translation encodes the block information directly into the variable names. Every model element is traceable in our framework by an unique identifier assigned during the parsing process. Furthermore, we make use of the Boogie2 *goto*-blocks to encode information about the automatically generated checks and invariants into the Boogie2 program. In case of an error report by the Boogie verification framework, we post-process the reported traces and counterexamples and present it in our graphical user interface in a human-readable way. Moreover, we have also presented a verification process that recommends how the user should use our framework. This process is supported by the graphical user interface of our implementation.

- **Practicability** Finally, we required our framework to be applicable to real world industrial models. We have shown the applicability of our framework by automatically verifying the MATLAB/Simulink designs provided by our industrial partners in our experimental evaluation. The results demonstrate that we are able to automatically parse, translate and verify industrial models (including referenced models and configuration files) in a reasonable amount of time.

Overall, our verification framework for discrete-time MATLAB/Simulink models meets all the requirements given in Section 1.2. It enables the automatic verification of discrete time MATLAB/Simulink models for the absence of a number of important run-time errors. In the next section, we discuss possible extensions to our framework that tackle open questions and motivate future work.

10.3 Outlook

In this section, we discuss possible improvements and extensions of our verification framework for discrete-time MATLAB/Simulink models presented in this thesis.

Optimizations of the Framework While the experimental evaluation of our framework is promising, there is still room for optimizations to achieve even better computation times for the verification and to increase the automation. In this paragraph, we discuss some of these short-term goals.

First, the complexity of the scheduling algorithm used in our approach is exponential in the worst case. However, it was not in the scope of this thesis to develop an efficient scheduling algorithm. Hence, a more sophisticated (and less brute force) algorithm may improve the translation time for a given model. Second, the use of the *Z3* theorem prover within the Boogie verification framework does not enable the full power of the *Z3* since the Boogie tool disables the automatic configuration[1] of *Z3*. This leads to situations where a Z3 instance

[1]The Z3 tool uses heuristics to select suitable theory solvers and to perform a number of simplification and rewriting steps.

started by Boogie is not able to solve a problem in hours, while a manually started instance solves the problem in seconds. Note that some configuration parameters are required for the Boogie tool to work correctly. Since we were able to achieve a huge performance increase for our framework by changing only one parameter, a thorough investigation of the Z3 parameters may lead to a substantial performance increase. Third, the abstract interpretation performed by Boogie is done w. r. t. arithmetic operations directly supported by the Boogie tool. Since we use built-in functions of the Z3 in our formal model and a number of optimizations (e. g., the use of `ite`), it could be useful to generate further invariants automatically by extracting them from an interval analysis performed on the intermediate representation of the model w. r. t. particular block parameters. Finally, the comprehensibility of the verification results could be increased by displaying them directly in the corresponding Simulink models.

Extension of the Supported Block Set and Arithmetics While the set of blocks supported by our framework is suitable to translate and verify our industrial case studies, it could be extended by further blocks from the standard library, e. g., the blocks from the *Lookup Tables* block set. Furthermore, the supported arithmetics could be extended in future. While the support of floating point values in our framework generally enables the verification of Simulink models in earlier design stages, the mapping to the *real* type is not sound w. r. t. rounding errors in floating point arithmetics. This means that although our technique reports the absence of errors in a model containing floating point values, there is a small chance that an error can occur because of floating point rounding. Although our technique is perfectly suitable to discover flaws in the general design of the models, it could be extended to use the theory of floats (QF_FPBV and QF_BV) once it has been fully added to the Z3 theorem prover and to the Boogie verification framework. Note that these are quantifier free theories and, hence, the translation for some block types has to be adjusted to use macros instead of functions (i. e., all Boogie2 functions need to be inlined with `{:inline}`). Furthermore, our framework currently does not support fixed-point data types as often used in embedded controllers. We already have done some work on a formalization of fixed-point types based on a mapping to the bit vector type in Boogie2. However, when using bit vectors, Boogie and Z3 do currently not meet our requirements w. r. t. performance. Hence, it would be an interesting question whether a formalization based on a mapping to the integer type performs significantly better.

Multirate Blocks In our framework, we currently do not support blocks that have a different sample time than the parent model. In future, it would be useful to add support for multirate blocks. However, models with multirate blocks may require stronger invariants and a higher value of k for the verification using k-induction since they do not calculate outputs in every global sampling step. Hence, adding support for multirate blocks may require the development of further techniques to automatically derive stronger invariants.

Stateflow A long term goal for our verification framework would be adding support for Stateflow models to enable a comprehensive verification of Simulink/Stateflow models. This would require the development of a slicing and a verification technique for Stateflow models. Since the simulation semantics of Stateflow is strictly sequential and, in contrast to other statechart dialects, deterministic, a mapping of Stateflow models to Boogie2 should generally be possible. However, to translate Stateflow models, a thorough analysis of the semantics of Stateflow models has to be done. Note that there is already some work on the semantics of Stateflow by Hamon and Rushby [Ham05, HR04]. We have already done some work on slicing of Stateflow models. To this end, we have applied the technique of Wang et al. [WDQ02] to Stateflow models. However, the granularity of this approach are composite states and, hence, it is rather imprecise. Especially w. r. t. the verification of Stateflow models, a more precise technique would be desirable.

Abstraction by Slicing In our framework, we currently use our slicing technique only to calculate static backward slices to perform a verification run on a partial model that contains all blocks that directly or transitively influence the calculation at a particular block. It is an interesting question whether our slicing technique can also be used as an abstraction technique. Therefore, forward slicing could be used to calculate a slice of all blocks that are affected by a particular block. For example, we could use limited `DiscreteIntegrator` blocks from the *DistWarn_corr* case study as slicing criterion. Then, a subsequent verification of the slice while assuming that the outputs of the block are arbitrary values within the limits would possibly render a fully automatic elimination of the spurious counterexamples. Furthermore, the development of conditioned slicing and chopping techniques for Matlab/Simulink to calculate abstractions for a model could be an interesting research topic.

Contracts Another possible extension of our framework is the integration of contract-based verification to improve the scalability of our framework w. r. t. large scale industrial controller models with thousands of blocks. However, contracts decrease the automation significantly. In contrast to Boström [Bos11], we would not recommend to add contracts to every atomic subsystem. Conditional executed subsystems are also atomic subsystems but may propagate their execution contexts outside the subsystem. Instead, we would recommend the use of contracts only for model references, atomic subsystems that model a controller function and Stateflow blocks since it is likely that a specification exists for these kind of blocks. This specification may be used to derive the contract or at least parts of it automatically. Note that although the Boogie verification framework naturally supports contract-based verification, adjustments in the translation are required to enable k-induction. More precisely, each atomic subsystem needs to be modeled within its own simulation loop.

List of Figures

2.1 The MDD Process 23

2.2 The MATLAB Architecture 25

2.3 The MDD Process in Simulink 26

2.4 A Example Simulink Model 28

2.5 The Simulation Phases in Simulink 31

2.6 Example Program and its Weakest Precondition 40

2.7 Example Program and CFG with Loop-Cutting 43

2.8 A Example Program and its CFG 56

2.9 PDG and Slice for Line **10** for Fig. 2.8 57

4.1 The General Verification Idea 68

4.2 The Architecture of the Verification Framework 70

4.3 The Verification Process 71

5.1 An `Enabled Subsystem` and its contents 77

5.2 A MultiPortSwitch 78

5.3 CFG for an Arbitrary Model with Nested Control Flow 80

5.4 Propagated ECs for a `MultiPortSwitch` 82

5.5 An Example Simulink Model and its MSDG, the Marked MSDG and the resulting Slice for the Block `write(mul)` 89

5.6 Example MATLAB/Simulink model with Bus Systems 90

5.7 Slice for “`Display ?`” (Original Algorithm) 91

5.8 Model with nested Bus Systems and a `BusAssigment` block . . . 93

5.9 Routing-aware MSDG with Slice for “`Display ?`” 96

5.10 Routing-aware Slice for “`Display ?`” 97

6.1 The Translation Process 100
6.2 Example Model with Control Flow 101
6.3 The Structure of the Formal Model 104
6.4 An Excerpt from the Example Simulink Model (Figure 5.5) and the Output and Update Functions 106
6.5 Over-approximation of *sqrt* with Linear Functions 129

7.1 Formal Specification for Inductive Invariant Checking 135
7.2 Formal Specification for K-Induction 136
7.3 Slicing Procedure for K-induction 139

8.1 The MeMo Tool Suite 147
8.2 The Parse Process for a Simulink Model 149
8.3 The MeMo Tool 153

9.1 Structure of TestModel 157
9.2 Architecture of the Odometer Model 158
9.3 Architecture of the DistWarn Model 159
9.4 Changed Model Elements for *TestModel_1_err* and *TestModel_corr* 167
9.5 Source for the spurious Counterexamples 170

List of Algorithms

5.1 Search for all Execution Contexts in a Model 84
5.2 Propagation of Conditional Execution Contexts (Backward) . . . 85
5.3 Slicing Algorithm 88

List of Tables

2.1 Primitive Types in Boogie2 49

6.1 Mapping of data types 108

9.1 Overview of the Characteristics of the Case Studies 160

9.2 Experimental Results for the Basic Slicing Technique 162

9.3 Experimental Results for the Routing-aware Slicing Technique . 164

9.4 Evaluation Results for the *Testmodel* Model 166

9.5 Evaluation Results for the *Odometer* Model 168

9.6 Evaluation Results for the *DistWarn* Model 169

List of Acronyms

CFG	Control Flow Graph	41
MDD	Model Driven Development	23
MDA	Model Driven Architecture	21
MDE	Model Driven Engineering	21
OMG	Object Management Group	21
UML	Unified Modeling Language	22
PIM	Platform Independent Model	22
CIM	Computation Independent Model	22
PSM	Platform Specific Model	22
DSML	Domain-specific Modeling Language	22
M-code	MATLAB Code	25
FOL	First Order Logic	35
VC	Verification Condition	36
WP	Weakest Precondition	35
WLP	Weakest Liberal Precondition	37
SMT	Satisfiability Modulo Theories	46
ESC	Extended Static Checking	47
PDG	Program Dependence Graph	54
MSDG	MATLAB/Simulink Dependence Graph	86
EC	Execution Context	30
CEC	Conditional Execution Context	79
Eclipse RCP	Eclipse Rich Client Platform	148
AST	Abstract Syntax Tree	149
EFSM	Extended Finite State Machine	64
STPDG	Synchronous Threaded Program Dependence Graph	65

Bibliography

References

[ABC$^+$11] Kelly Androutsopoulos, David Binkley, David Clark, Nicolas Gold, Mark Harman, Kevin Lano, and Zheng Li. Model projection: simplifying models in response to restricting the environment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 291–300, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0445-0.
(Referenced on pages 62 and 64.)

[ACH$^+$09] Kelly Androutsopoulos, David Clark, Mark Harman, Zheng Li, and Laurence Tratt. Control dependence for extended finite state machines. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 216–230, 2009.
(Referenced on pages 62 and 64.)

[ACH$^+$13] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based model slicing: A survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, August 2013.
(Referenced on page 63.)

[ACOS00] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: control laws in Z. In *Third IEEE International Conference on Formal Engineering Methods (ICFEM 2000).*, pages 169–176, 2000.
(Referenced on page 60.)

[AGH$^+$09] K. Androutsopoulos, N. Gold, M. Harman, Zheng Li, and L. Tratt. A theoretical and empirical study of EFSM dependence. In *IEEE International Conference on Software Maintenance (ICSM 2009).*, pages 287–296. IEEE, Sept 2009. ISSN: 1063-6773.
(Referenced on pages 62 and 64.)

[ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, December 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).
(Referenced on page 59.)

[BC12] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for Simulink's simulation engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 129–138, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1212-7.
(Referenced on page 60.)

[BCD+06] Mike Barnett, Bor-yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006. ISBN: 978-3-540-36749-9.
(Referenced on pages 15 and 47.)

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool, 2012. ISBN: 9781608458820.
(Referenced on page 23.)

[BDdM+13] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.
(Referenced on page 46.)

[BGH07] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2), April 2007.
(Referenced on pages 161 and 164.)

[BHH+14] Pontus Boström, Mikko Heikkilä, Mikko Huova, Marina Waldén, and Matti Linjama. Verification and validation of a pressure control unit for hydraulic systems. In István Majzik and Marco Vieira, editors, *Software Engineering for Resilient Systems*, volume 8785 of *Lecture Notes in Computer Science*, pages 101–115. Springer International Publishing, 2014. ISBN: 978-3-319-12240-3.
(Referenced on page 61.)

[BHL$^+$10] Thomas Ball, Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards scalable modular checking of user-defined properties. In Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-15056-2.
(Referenced on page 47.)

[BLS05] Mike Barnett, K.RustanM. Leino, and Wolfram Schulte. The spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-24287-1.
(Referenced on pages 41 and 47.)

[BM07] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification.* Springer Berlin Heidelberg,, Berlin, Heidelberg, 2007. ISBN: 978-3-540-74112-1.
(Referenced on page 33.)

[Bos11] Pontus Boström. Contract-based verification of simulink models. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 291–306. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-24558-9.
(Referenced on pages 61 and 179.)

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
(Referenced on page 46.)

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB standard: Version 2.0*, 2010.
(Referenced on page 53.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.
(Referenced on page 53.)

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*,

2(4):410–425, 2000.
(Referenced on page 60.)

[CCM$^+$03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time Simulink to Lustre. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99. Springer Berlin / Heidelberg, 2003.
(Referenced on pages 60 and 66.)

[CCO05] Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. Control law diagrams in Circus. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 253–268. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-27882-5.
(Referenced on page 61.)

[CDC00] Cécile Canovas-Dumas and Paul Caspi. A PVS proof obligation generator for Lustre programs. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 179–188. Springer Berlin Heidelberg, 2000. ISBN: 978-3-540-41285-4.
(Referenced on page 60.)

[CDH$^+$09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-03358-2.
(Referenced on page 47.)

[CFR$^+$99] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 298–313. Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-66559-5.
(Referenced on page 65.)

[CMW13] Ana Cavalcanti, Alexandre Mota, and Jim Woodcock. Simulink timed models for program verification. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *Lecture Notes in Computer Science*, pages 82–99. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-39697-7.
(Referenced on page 61.)

[Det96] D.L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, 1996.
(Referenced on page 47.)

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *LNCS*, pages 351–368. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23701-0.
(Referenced on pages 43, 134, and 136.)

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
(Referenced on page 35.)

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.
(Referenced on page 46.)

[DLNS98] D.L. Detlefs, K. Rustan M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. In *SRC Research Report 159, Compaq Systems Research Center*, 1998.
(Referenced on page 47.)

[dMB08] Leonardo de Moura and Nikolaj Börner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-78799-0.
(Referenced on page 52.)

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
(Referenced on page 41.)

[FL06] Chris Fox and Arthorn Luangsodsai. And-or dependence graphs for slicing statecharts. 2006.
(Referenced on page 64.)

[FLGD$^+$11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-22109-5.
(Referenced on page 60.)

[FLL$^+$02] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. In *ACM Sigplan Notices*, volume 37, pages 234–245. ACM, 2002.
(Referenced on page 47.)

[Flo67] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.
(Referenced on page 34.)

[FOW84] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 125–132. Springer Berlin Heidelberg, New York, NY, USA, 1984. ISBN: 978-3-540-12925-7.
(Referenced on page 54.)

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM. ISBN: 1-58113-336-7.
(Referenced on page 40.)

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs. *Electronic Notes in Theoretical Computer Science*, 65(5):50 – 64, 2002. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002).
(Referenced on pages 62 and 63.)

[Ham05] Grégoire Hamon. A denotational semantics for Stateflow. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT '05)*, pages 164–172, New York, NY, USA, 2005. ACM. ISBN: 1-59593-091-4.
(Referenced on page 179.)

[HHWT97] ThomasA. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-63166-8.
(Referenced on page 60.)

[HJWW09] David Hardin, D. Randolph Johnson, Lucas Wagner, and Michael W. Whalen. Development of security software: A high-assurance methodology. In *International Conference of Formal Engineering Methods (ICFEM 2009)*. Springer Verlag, 2009.
(Referenced on page 62.)

[HLW12] Wei Hu, T. Loeffler, and J. Wegener. Quality model based on ISO/IEC 9126 for internal quality of MATLAB/Simulink/Stateflow models. In *2012 IEEE International Conference on Industrial Technology (ICIT)*, pages 325–330, March 2012.
(Referenced on pages 18 and 74.)

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
(Referenced on page 34.)

[HR04] Grégoire Hamon and John M. Rushby. An operational semantics for Stateflow. 2984:229–243, 2004.
(Referenced on page 179.)

[HRB90] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
(Referenced on page 57.)

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.
(Referenced on page 60.)

[JH05] Anjali Joshi and Mats P.E. Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *Computer Safety, Reliability, and Security*, volume 3688 of *LNCS*, pages 122–135. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-29200-5.
(Referenced on page 62.)

[Ken02] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-43703-1.
(Referenced on pages 21 and 22.)

[KR03] Aditya Rajeev Kulkarni and S. Ramesh. Static slicing of reactive programs. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 98–107, 2003.
(Referenced on page 65.)

[Kri98] Jens Krinke. Static slicing of threaded programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, New York, NY, USA, 1998. ACM. ISBN: 1-58113-055-4.
(Referenced on page 65.)

[Kro11] B.H. Krogh. sliceMDL. Private Communication, February 2011.
(Referenced on pages 62 and 66.)

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures*, volume 5. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-74104-6.
(Referenced on pages 33, 45, and 46.)

[KSTV03] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 34 – 43, sept. 2003. ISSN: 1063-6773.
(Referenced on pages 62 and 64.)

[Lan09] Kevin Lano. Slicing of UML state machines. In *Proceedings of the 9th WSEAS international conference on Applied informatics and communications*, pages 63–69. World Scientific and Engineering Academy and Society (WSEAS), 2009.
(Referenced on pages 62, 63, and 66.)

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281 – 288, 2005.
(Referenced on page 40.)

[Lei08] K. Rustan M. Leino. This is Boogie 2. Unpublished Technical Report, 2008.
(Referenced on pages 48 and 49.)

[LF10] A. Luangsodsai and C. Fox. Concurrent statechart slicing. In *2nd Computer Science and Electronic Engineering Conference (CEEC)*, pages 1–7, Sept 2010.
(Referenced on page 64.)

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-05088-6.
(Referenced on page 47.)

[Mata] MathWorks. *MATLAB Simulink*. The MathWorks Inc, http://www.mathworks.com/products/simulink/.
(Referenced on page 26.)

[Matb] MathWorks. *Simulink getting started guide*. The MathWorks Inc, http://www.mathworks.com/help/pdf_doc/simulink/sl_gs.pdf.
(Referenced on page 26.)

[Mat07] MathWorks. Code verification and run-time error detection through abstract interpretation. Whitepaper, 2007.
(Referenced on page 61.)

[Mat14a] The MathWorks. *MATLAB Documentation*. The MathWorks, Inc., 2014. `http://www.mathworks.de/de/help/matlab/index.html`.

(Referenced on page 13.)

[Mat14b] The MathWorks. *Simulink Documentation.* The MathWorks, Inc., 2014. http://www.mathworks.de/de/help/simulink/index.html.
(Referenced on pages 14, 15, 82, 99, and 110.)

[MBR06] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. Tool for translating Simulink models into input language of a model checker. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 606–620. Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-47460-9.
(Referenced on page 60.)

[MM$^+$01] Joaquin Miller, Jishnu Mukerji, et al. Model driven architecture (MDA), 2001. Object Management Group, Draft Specification ormsc/2001-07-01.
(Referenced on page 21.)

[MWC10] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, February 2010.
(Referenced on page 62.)

[MZC12] Chris Marriott, Frank Zeyda, and Ana Cavalcanti. A tool chain for the automatic generation of Circus specifications of Simulink diagrams. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 294–307. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-30884-0.
(Referenced on pages 61 and 66.)

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.
(Referenced on pages 56 and 57.)

[RHR91] Christophe Ratel, Nicolas Halbwachs, and Pascal Raymond. Programming and verifying critical systems by means of the synchronous data-flow language Lustre. *SIGSOFT Softw. Eng. Notes*, 16(5):112–119, September 1991.
(Referenced on page 60.)

[RKK04] S. Ramesh, A. Kulkarni, and V. Kamat. Slicing tools for synchronous reactive programs. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 217–220, New York, NY, USA, 2004. ACM.

ISBN: 1-58113-820-2.
(Referenced on page 65.)

[RS09] Michael Ryabtsev and Ofer Strichman. Translation validation: From Simulink to C. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 696–701. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-02657-7.
(Referenced on pages 61 and 66.)

[RS10] Pritam Roy and Natarajan Shankar. SimCheck: An expressive type system for Simulink. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 149–160, April 2010. Washington D.C.
(Referenced on pages 61 and 66.)

[RW10] Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.
(Referenced on page 151.)

[SA13] Rupinder Singh and Vinay Arora. Literature analysis on model based slicing. *International Journal of Computer Applications*, 70(16):45–51, 2013.
(Referenced on page 62.)

[Sch06] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.
(Referenced on page 21.)

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
(Referenced on page 24.)

[Sha00] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In Catuscia Palamidessi, editor, *CONCUR 2000 – Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2000. ISBN: 978-3-540-67897-7.
(Referenced on page 60.)

[Sil12] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)*, 44(3):12:1–12:41, June 2012.
(Referenced on pages 54 and 57.)

[SK00] B Izaias Silva and Bruce H Krogh. Formal verification of hybrid systems using CheckMate: a case study. In *Proceedings of the American Control Conference*, volume 3, pages 1679–1683, 2000.
(Referenced on pages 59 and 60.)

[SSC$^+$04] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a ”safe” subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT ’04, pages 259–268, New York, NY, USA, 2004. ACM. ISBN: 1-58113-860-1.
(Referenced on pages 60 and 66.)

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Jr. Hunt, WarrenA. and StevenD. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000. ISBN: 978-3-540-41219-9.
(Referenced on page 43.)

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
(Referenced on pages 54 and 57.)

[Tiw02] Ashish Tiwari. Formal semantics and analysis methods for Simulink Stateflow models, 2002. Unpublished report, SRI International.
(Referenced on page 60.)

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.
(Referenced on pages 60 and 66.)

[VLH07] Sara Van Langenhove and Albert Hoogewijs. SV$_t$L: System verification through logic tool support for verifying sliced hierarchical statecharts. 4409:142–155, 2007.
(Referenced on pages 62, 63, and 66.)

[WB14] Jonatan Wiik and Pontus Boström. Contract-based verification of MATLAB and Simulink matrix-manipulating code. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering*, volume 8829 of *Lecture Notes in Computer Science*, pages 396–412. Springer International Publishing, 2014. ISBN: 978-3-319-11736-2.
(Referenced on page 61.)

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN: 0-13-948472-8.
(Referenced on page 60.)

[WDQ02] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Meth-*

ods: Formal Methods and Software Engineering, pages 435–446, London, UK, UK, 2002. Springer-Verlag. ISBN: 3-540-00029-1.
(Referenced on pages 62, 63, 66, and 179.)

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN: 0-89791-146-6.
(Referenced on page 54.)

[WY04] Fangjun Wu and Tong Yi. Dependence analysis for UML class diagrams. *Journal of Electronics (China)*, 21(3):249–254, 2004.
(Referenced on page 62.)

[ZC09] Frank Zeyda and Ana Cavalcanti. Mechanised translation of control law diagrams into Circus. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-00254-0.
(Referenced on pages 61 and 66.)

[Zha98] Jianjun Zhao. Applying slicing technique to software architectures. In *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '98).*, pages 87–98, Aug 1998.
(Referenced on page 62.)

[Zha04] Y. Zhang. Test-driven modeling for model-driven development. *Software, IEEE*, 21(5):80–86, Sept 2004.
(Referenced on page 24.)

[ZZW$^+$13] Liang Zou, Naijun Zhan, Shuling Wang, Martin Fränzle, and Shengchao Qin. Verifying Simulink diagrams via a hybrid hoare logic prover. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 9:1–9:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN: 978-1-4799-1443-2.
(Referenced on page 60.)

Publications of Robert Reicherdt

[HRB13] Paula Herber, Robert Reicherdt, and Patrick Bittner. Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 8:1–8:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN: 978-1-4799-1443-2.

[HWS$^+$11] Wei Hu, Joachim Wegener, Ingo Stürmer, Robert Reicherdt, Elke Salecker, and Sabine Glesner. MeMo - methods of model quality. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 127–132, 2011.

[RG12] Robert Reicherdt and Sabine Glesner. Slicing MATLAB Simulink models. In *ACM/IEEE 34th International Conference on Software Engineering (ICSE'12)*, pages 551–561. IEEE, 2012.

[RG14a] Robert Reicherdt and Sabine Glesner. Formal verification of discrete-time MATLAB/Simulink models using Boogie. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, volume 8702 of *Lecture Notes in Computer Science*, pages 190–204. Springer International Publishing, 2014. ISBN: 978-3-319-10430-0.

[RG14b] Robert Reicherdt and Sabine Glesner. Methods of model quality in the automotive area. pages 73–74. Köllen, 2014. ISBN: 978-388579-621-3.

Supervised Theses

[Bit13] Patrick Bittner. Automatische Transformation und Verifikation von MATLAB/Simulink Modellen mit UCLID. Master Thesis, 2013. TU Berlin.

[Li14] Yi Li. Entwicklung eines CAN Bootloaders. Bachelor Thesis, 2014. TU Berlin.

[Sta14] Georgios Stavrakakis. 3D-Visualisierung von Metrikergebnissen für MATLAB/Simulink. Bachelor Thesis, 2014. TU Berlin.