Information Flow Analysis of Discrete Embedded Control System Models

vorgelegt von

Marcus Mikulcak, M.Sc.
ORCID: 0000-0003-2900-2897

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Clemens Gühmann Technische Universität Berlin
Gutachterin:	Prof. Dr. rer. nat. Sabine Glesner Technische Universität Berlin
Gutachter:	Prof. Dr.-Ing. Christian Hammer Universität Potsdam
Gutachter:	Prof. Dr.-Ing. Frank Slomka Universität Ulm

Tag der wissenschaftlichen Aussprache:

06. November 2019

Berlin 2020

Abstract

Embedded systems used in safety-critical domains have to uphold strict *safety* and *security* requirements. At the same time, their complexity has been strongly increasing across application domains. To manage this rise in complexity, manufacturers have shifted towards *model-driven development* methodologies. While successful in managing the complexity in the *development* of large, interconnected systems, *analysis* and *verification* techniques for model-driven development methods and languages still have to reach a similar level of maturity as those for text-based imperative programming languages traditionally used in the development of safety-critical embedded systems. In this thesis, we present *an information flow analysis* method for discrete embedded control system models, which has the potential to identify possible violations of both safety requirements and security policies by analyzing where and under which conditions information travels through a model. The main challenges such an information flow analysis faces are to (1) consider the specific semantics of the modeling languages, which heavily rely on concurrency and a complex notion of timing, and (2) relate the strongly different semantics of signal-flow-oriented and state-machine-based components, which comprise embedded control system models.

Our major contribution is twofold: First, we provide an information flow analysis for the signal-flow-oriented components of an embedded control system model. The main idea of this analysis is that we only extract that information from an existing model which is required to analyze information flow in respect to both its timing and functionality. To this end, our technique captures *timed path conditions*, i.e., the precise control, data and timing conditions under which information flow is enabled as well as when and how these conditions are triggered. Second, we relate the inherently different semantics of the signal-flow-oriented and the state-machine-based components. To this end, we first translate the state-machine-based controller into a formally verifiable representation and, second, combine this representation with *condition observer automata* which we generate from the timed path conditions extracted in the first step of our method. This enables us to use the well-established technique of model checking to identify precisely the behavior that leads to the execution of information flow paths under analysis.

To show the practical applicability of our approach, we have implemented it as a fully automatic and modular framework for MATLAB *Simulink*/*Stateflow* and Modelica, two widely used languages from the domain of embedded control systems, and applied our information flow analysis to two industrial case studies. In these case studies, we are able to verify *integrity* by checking that no information flow is possible from a non-critical to a critical component.

Zusammenfassung

Im sicherheitskritischen Bereich unterliegen eingebettete Systeme strengen Anforderungen an Ausfall- sowie Datensicherheit. Zugleich ist die Systemkomplexität in den vergangenen Jahren in allen Anwendungsbereichen stark gestiegen. Um diesem Anstieg zu begegnen, nutzen Hersteller modellgetriebene Entwicklungsansätze. Während Ansätze dieser Art bereits erfolgreich dazu genutzt werden, komplexe, miteinander verbundene Systeme zu *entwickeln*, haben Ansätze zur *Analyse* und *Verifikation* dieser Systeme noch nicht den Stand erreicht, den ähnliche Methoden für imperative Programmiersprachen aufweisen. In dieser Arbeit stellen wir eine Informationsflussanalyse für diskrete eingebettete Kontrollsystemmodelle vor, die es ermöglicht, Verletzungen von Ausfall- sowie Datensicherheitsanforderungen zu erkennen. Unsere Analyse verfolgt, wie und unter welchen Bedingungen Informationen durch ein Modell fließen. Die Herausforderungen in der Entwicklung einer solchen Analyse liegen in (1) der Berücksichtigung der spezifischen Semantik der Modellierungssprachen, welche auf komplexen zeitlichen Abhängigkeiten und Parallelität basieren, und (2) der Verbindung der stark heterogenen Semantiken der signalflussorientierten und jener auf Zustandsautomaten basierenden Komponenten, aus denen Kontrollsystemmodelle aufgebaut sind.

Die vorliegende Arbeit leistet in diesem Gebiet zwei Beiträge. Zum einen eine Informationsflussanalyse für die signalflussorientierten Komponenten eines Kontrollsystemmodells. Diese Analyse basiert auf der Idee, nur diejenigen Informationen eines Modells zu extrahieren, die für die Analyse des Informationsflusses hinsichtlich des Zeitverhaltens und der Funktionalität des Modells relevant sind. Um dies zu ermöglichen, erfasst unser Ansatz *timed path conditions*, das heißt, diejenigen Bedingungen, die präzise das Kontroll-, Zeit-, und Datenverhalten abbilden, unter denen Informationsfluss stattfindet. Zum anderen schafft unsere Arbeit eine Verbindung zwischen den stark heterogenen Semantiken der signalflussorientierten und den auf Zustandsautomaten basierenden Komponenten. Unser Ansatz ermöglicht dies durch eine Übersetzung der Automaten in eine formal verifizierbare Darstellung, und die Kombination dieser Darstellung mit *condition observer automata*, welche wir aus den im ersten Schritt extrahierten *timed path conditions* generieren. Diese Verbindung der Semantiken ermöglicht uns, eine wohlfundierte Technik wie model checking zu nutzen, um genau das Verhalten des Zustandsautomaten zu identifizieren, welches zur Ausführung eines Informationsflusspfades führt.

Um die Anwendbarkeit unseres Ansatzes unter Beweis zu stellen, präsentieren wir außerdem eine vollautomatische und modular aufgebaute Implementierung für MATLAB Simulink/Stateflow und Modelica, zwei im Bereich sicherheitskritischer eingebetteter Software weit verbreitete Sprachen. Mithilfe dieser Implementierung konnten wir unseren Ansatz zur Informationsflussanalyse auf zwei Fallstudien aus dem industriellen Bereich anwenden. In beiden Fallstudien waren wir in der Lage, die Integrität kritischer Berechnungen sicherzustellen, indem wir Informationsfluss zwischen nicht-kritischen und sicherheitskritischen Komponenten ausschließen konnten.

Danksagung

Diese Arbeit wäre ohne die Anregungen und die Unterstützung einer Vielzahl von Menschen nicht zustande gekommen.

Zuallererst möchte ich Prof. Dr. Sabine Glesner für ihre jahrelange Unterstützung meiner Arbeit sowie für die vielen Gespräche und Anregungen danken. Sie hat mir nicht nur einen wunderbaren Rahmen für meine Promotion am Fachgebiet *Software and Embedded Systems Engineering* (SESE) der Technischen Universität Berlin geboten, sondern mich auch dazu ermutigt, mit meiner Arbeit in die Welt zu treten, sie an anderen Orten vorzustellen und dadurch voranzubringen. Dank ihrer Vermittlung konnte ich meine Arbeit auch an den Lehrstühlen von Prof. Dr.-Ing. Christian Hammer und Prof. Dr.-Ing. Frank Slomka vorstellen. Beide haben meine Arbeit im Anschluss daran als Gutachter betreut und begleitet. Die hilfreichen Anregungen und Hinweise beider haben diese Arbeit enorm verbessert und die Offenheit und Gründlichkeit beider im Umgang mit meiner Arbeit haben mich beeindruckt und geehrt.

Prof. Glesner hat mich auch darin unterstützt, sechs Monate als Visiting Research Collaborator in der Gruppe von Prof. Sharad Malik an der Princeton University zu verbringen. Die Gespräche und die gemeinsame Arbeit mit ihm, Prof. Aarti Gupta und Zhixing Xu haben meine Forschung extrem bereichert und mir neue Perspektiven eröffnet. Für Prof. Glesners Unterstützung in Planung und Durchführung dieser unvergesslichen Zeit bin ich ihr – sowie dem Deutschen Akademischen Austauschdienst – sehr dankbar.

Sehr dankbar bin ich auch für die Unterstützung durch das *Bundesministerium für Bildung und Forschung* (BMBF) sowie durch unseren Industriepartner *Model Engineering Solutions GmbH* (MES), die meine Forschungsarbeit im Rahmen der Projekte *Change Impact Analyses for Software Models* (CISMo) und *Effective Complexity of Software Models* (ECoSMo) förderten. Auf Seiten von MES war Dr. Heiko Dörr zu jeder Zeit ein sehr hilfsbereiter und engagierter Gesprächspartner für mich, der meine Forschung nicht nur durch das Zurverfügungstellen zahlreicher Anwendungsbeispiele maßgeblich vorangebracht und motiviert hat.

Der Softwareprototyp, der im Rahmen meiner Arbeit und Forschung entstanden ist, hat von der langjährigen Mitarbeit meiner studentischen Hilfskräfte Moritz Lummerzheim, Feras Fattohi und Umar Ahmad unheimlich profitiert. Ich bin sehr froh, dass ihr mir geholfen habt, meine Ideen Realität werden zu lassen.

Diese Arbeit wäre ohne die kontinuierliche Ermutigung, die klugen Kommentare und nicht zuletzt die immer tatkräftige Unterstützung von Prof. Dr.-Ing. Paula Herber und Dr.-Ing. Thomas Göthel nicht zustande gekommen. Sie haben diese Arbeit maßgeblich geprägt und vom ersten Satz bis zur letzten Seite begleitet. Dass die beiden nicht nur bewunderte Postdocs blieben, sondern mir auch zu Freunden wurden, ist ein großes Glück. Dafür, aber auch für ihre unermüdliche Unterstützung und ihr Interesse an meiner Arbeit werde ich immer dankbar sein.

Meine Promotionszeit wäre nicht die gleiche gewesen ohne meine Kolleginnen und Kollegen am Fachgebiet SESE. Die Gespräche während all der gemeinsamen Kaffeepausen und Mittagessen haben die Arbeit an meiner Dissertation nicht nur sehr viel schöner, sondern auch besser gemacht. Ganz besonders froh und dankbar bin ich über die Freundschaften, die sich mit Dr.-Ing. Sebastian Schlesinger und Dr. Nils Berg entwickelt haben. Die Möglichkeit, mein Leid mit euch zu teilen, hat mir auch über Momente hinweggeholfen, in denen ich sicher war, diese Arbeit nie zu Ende bringen zu können.

Meinen Eltern danke ich für ihre große Unterstützung – nicht nur während meiner Promotionszeit, sondern auch in all den Jahren davor. Sie haben mich früh dazu ermutigt, meinen eigenen Weg zu gehen, haben mir an jedem Ort, an den es mich auf diesem Weg verschlagen hat, geholfen, ein Zuhause aufzubauen, und mir nicht zuletzt sehr früh ermöglicht, mich an meinem eigenen 486er auszuprobieren.

Mein innigster Dank gilt meiner Frau Laetitia Lenel. Ich bin nicht in der Lage, ihr genug für ihr Vertrauen in mich und meine Fähigkeiten zu danken, für unsere Gespräche und Diskussionen, ihre unendlich klugen Kommentare und Einsichten, sowie für ihre unermüdliche Ermunterung gegen meinen erbitterten Widerstand. Das gemeinsame Leben mit ihr und unserer wunderbaren Tochter Mathilda hat mir erst die Kraft, die Ruhe und den Mut gegeben, diese Arbeit fertigzustellen.

Contents

1 Introduction

Embedded systems are shifting from unconnected to increasingly interconnected functionality. The connection of these *cyber-physical systems* to the internet and to each other poses severe threats to confidentiality, integrity and availability. In 2015, this was prominently demonstrated by software security researchers Miller and Valasek as they remotely exploited a vulnerability in the internet-connected entertainment system of a Jeep Cherokee, gaining control over vital functions such as acceleration, brakes and steering [Miller and Valasek 2015]. The attack was made possible by a combination of two factors: (1) the existence of the on-board *Controller Area Network* (CAN) bus [ISO 1993] which connects most major components in the car in order for them to receive commands and exchange information, and (2) a range of common vulnerabilities in the internet–connected entertainment system of the car [Koscher et al. 2010b; Checkoway et al. 2011; Mazloom et al. 2016; Choi and Jin 2019].

For safety-critical systems, correct operation at all times is of the utmost importance. To protect cars from attacks, i.e., to protect the safety-critical components from remote interference, every component in itself needs to adhere to strict security standards. Furthermore, as all modules are able to communicate with each other freely, thus, there is a risk of security violations as well as component failures traveling through the system without constraint. To overcome this problem, the state-of-the-art safety standard defined by the *International Organization for Standardization* (ISO) recommends partitioning of the software [ISO 2009]. The aim of this method is to safely break down the system into smaller parts in order for them to be analyzable individually without having to consider the complete system. The most basic partitioning method is to completely refrain from utilizing interconnected systems. If components are not able to communicate with each other, security violations and safety failures never travel between them. Modern

embedded systems, however, rely heavily on interconnections. Thus, more advanced analysis methods for connected systems are required, which are able to safely identify the flow of information between components in heavily interconnected systems.

Especially in the automotive domain, the complexity of embedded software systems has increased dramatically in the past. While in 2007, the binary code of the software in an upper-class vehicle amounted to approximately 65 MB, recent generations contain more than 1 GB of binary code [Pretschner et al. 2007; Braun et al. 2014]. Developers have therefore shifted towards MDD [Hailpern and Tarr 2006; Navet and Simonot-Lion 2009], which enables system design on a high level of abstraction and early simulation. Additionally, it allows for the automated generation of implementation-specific software, a highly error-prone process when performed manually. While model-driven development enables a better understanding of complex systems, further technique are needed to prove adherence to safety and security standards [ISO 2009]. However, due to the strongly differing semantics of models and code, classic analysis techniques are not applicable to MDD [Tanković et al. 2012]. It is therefore imperative to develop novel analysis approaches that are able to perform, e.g., information flow analysis or security policy compliance checks directly on the models that are used as the main development artifacts.

The challenges we face when developing novel static analysis techniques for model-based development can be summarized as follows: (1) The semantics of the most widely-used modeling languages for discrete embedded controllers are based on the concept of *signal-flow graph*s (SFGs), which are a common abstraction used to model physical systems as well as their controllers. They naturally express data-intensive computations with local evaluation of data, as commonly found in control systems [Bonchi et al. 2017a]. This *signal-flow semantics* [Misra 2004] introduces characteristics not present in the semantics of traditional imperative programming languages which are utilized in the domain of safety-critical software. One of the strongest differences is the utilization of time-dependent, stateful modeling elements, extending the execution semantics of the model by a complex notion of timing. Additionally, due to the signal-flow nature of the models, they inherently utilize concurrency, further increasing the complexity of analyses of their execution behavior. (2) Modeling languages allow the developer to utilize development patterns with strongly differing semantics in the same model. To combine, e.g., data manipulation on signals with complex control flow mechanics, industrially used modeling languages, such as MATLAB Simulink/Stateflow and Modelica,

allow for the integration of state machines into signal-flow models. Thus, new analysis techniques that support the simultaneous extraction of information from both modeling styles are highly desirable.

The aim of this thesis is to establish a framework for the automated extraction and analysis of information flow in discrete embedded control system models. In the following, we list the criteria our approach should fulfill:

1. **Signal-Flow Semantics:** The proposed methodolody must be able to cope with the characteristics specific to discrete embedded control system models, i.e., with complex timing behavior as well as the concurrency inherent to these models.

2. **Combined Analysis:** The proposed approach has to be able to extract information flow from models that combine signal-flow and state-machine-based components.

3. **Language Support:** We require our methodology to be applicable to industrially used languages for the model-based design of discrete embedded control systems. Additionally, our approach should cover a broad range of modeling elements frequently used in the design of such systems and should facilitate easy extension of the set of supported elements.

4. **Automation:** To be integrated into existing quality assurance processes for MDD, the proposed methodology should be applicable fully automatically. This means that the analysis must not require annotations or user input.

5. **Applicability:** Finally, we require our methodology to have acceptable analysis effort. This should be demonstrated by applying our novel analysis technique to industrial case studies from the automotive domain.

To meet these criteria, we propose a novel methodology that provides a method to identify and analyze information flow relations in discrete embedded control system models containing complex control flow and timing behavior. To demonstrate the industrial applicability of our approach, we have fully implemented our methodology for two of the most widely-used modeling languages for the development of embedded controller software: MATLAB Simulink/Stateflow by The MathWorks and Modelica by the Modelica Association [Schroeder et al. 2015; Sutherland et al. 2016]. Both languages implement a graphical development front-end to develop models employing signal-flow semantics. As our approach targets the timing behavior of signal-flow-based semantics,

we are confident that it can be applied to further simulation languages with similar semantics. To facilitate the extensibility of our approach, we have implemented it as a modular framework.

The main contributions of this thesis are:

1. We have developed an **information flow analysis approach for MATLAB Simulink models**. Our method adapts the concept of path conditions to the domain of model-based development, using MATLAB Simulink as a modeling language widely-used in the development of discrete embedded system controllers. Our extension, which we call *timed path conditions* (TPCs), is able to express the data as well as the time-dependent execution behavior of information flow paths through a signal-flow-oriented model. To identify non-interference between model elements, we use timed path conditions to formulate a *constraint satisfaction problem* and solve it using a constraint solver. If no solution can be found, we have shown non-interference for the model elements under analysis.

2. As industrial embedded system models most often contain complex control logic, our technique supports the **analysis of information flow in combined MATLAB Simulink/Stateflow models** consisting of signal-flow-oriented components modeled in Simulink and state-machine-based controllers implemented using Stateflow. To enable a combined analysis, we make use of an existing technique to translate one of the most common modeling languages for embedded controllers, Stateflow, into formally well-defined UPPAAL *timed automata*. Using this translation, we merge the two strongly differing execution semantics of embedded system controllers developed in MATLAB Simulink/Stateflow and are able to identify non-interference on paths through such combined models.

3. To demonstrate the applicability of our method to other modeling languages that are based on signal-flow graphs, we have developed an **information flow analysis for discrete embedded control systems developed using Modelica**.

4. We have developed a **fully automatic framework to analyze the information flow in discrete embedded systems**. Using this framework, we show that the extraction of *timed path conditions* from the signal-flow-oriented model components as well as the translation of the state-machine-based components to UPPAAL timed automata can be performed fully automatically.

5. The **experimental results** that we have obtained using our methodology show that the extraction and analysis of information flow from synthetic as well as **industrial case studies** from the automotive domain can be performed with **acceptable effort**.

We have published our work as follows: In Mikulcak et al. [2017], we have introduced the concept of timed path conditions and have shown how they can be utilized to identify information flow in a MATLAB Simulink model. In Mikulcak et al. [2016], we have extended our information flow analysis of MATLAB Simulink models to additionally support the extraction of flow data in the face of complex control behavior modeled in Stateflow. In Mikulcak et al. [2018], we have introduced a fully automatic solution based on a translation to UPPAAL timed automata that does not pose any restrictions on the controller implementation. Further, in this work we have presented experimental results from an industrial case study provided by our partners from the automotive industry to demonstrate the practical applicability of our approach. Finally, in [Mikulcak et al. 2019], we have published a detailed explanation of our method and its components.

This thesis is structured as follows: In Chapter 2, we introduce a number of concepts that are necessary for the understanding of this thesis. In Chapter 3, we provide a discussion of related work. Chapter 4 presents an overview of our system to identify information flow in discretely-timed embedded control system models and discusses requirements that models have to fulfill in order to be analyzable by our system. Chapter 5 and 6 explain the details of our system. In Chapter 5, we present our concept of timed path conditions and provide a detailed description of our technique to infer non-interference between arbitrary model elements using constraint solving. Chapter 6 presents our technique to extract information flow from models containing both signal-flow-oriented and state-machine-based components. In Chapter 7, we provide details on the implementation of our approach as a fully automatic and extensible framework, present case studies from the automotive domain and discuss experimental results. Chapter 8 closes this thesis with a conclusion and a discussion of future work.

2 Background

In this chapter, we provide preliminaries that are the foundation of this thesis. First, we give a brief introduction to the domain of model-driven development. Subsequently, we present signal-flow-based modeling languages, namely MATLAB Simulink/Stateflow and Modelica, after which we introduce the concept of information flow analysis. We close with an introduction to constraint satisfaction problems and their solvers as well as to system verification using model checking.

2.1 Model-Driven Development of Embedded Software

Models provide abstractions by forming a representation of the essential components and functionalities of a system. Consequently, models are less complex and they are easier to grasp, maintain, and debug, not only by the model developers, but also by specialists from other domains involved in the development process, e.g., electrical or mechanical engineers. The three major levels of abstraction commonly utilized in model-driven development are:

***Computation-Independent Model* (CIM)** The CIM defines what the system is expected to do while hiding all information-technology-related specifications, such as the algorithmic implementation [Truyen 2006].

***Platform-Independent Model* (PIM)** The PIM captures the detailed functionality of the system, defines algorithmic details as well as timing behavior.

***Platform-Specific Model* (PSM)** The PSM contains enough details about the functionality of the system as well as about the target platform to produce an implementation that can be deployed.

As part of the MDD approach standard defined in Kleppe et al. [2003], a PIM, combined with a *platform model*, contains sufficient information about the functionality of a system to enable the *automatic* generation of a PSM, i.e., enabling the *automated* generation of non-essential implementation-specific details.

In this thesis, we focus on *model-centric* approaches, as this style is the one most adopted in industrial software development processes for safety-critical systems [Albers et al. 2006; Frevert et al. 2006]. *Model-centric techniquees* describe a development style in which the development is solely focused on models. Models form the main development artifact and need to contain sufficient detail to, given a platform model, enable the automatic generation of a system implementation. This technique eliminates the need to feed back source code changes into the model, but introduces the necessity to utilize modeling languages expressive enough to capture all required details of the functionality as well as the timing of the system. However, as a wide range of modeling languages supports the design of systems on various levels of abstraction, the model focus can be maintained.

2.1.1 Model-Driven Development in Software Development Guidelines

Due to the advantages of model-driven development and automatic generation of code from abstract models, the utilization of MDD techniques is recommended as *state of the art* by international standards for the design of safety-critical electronic systems. While IEC [2010] as an overarching standard for general safety-related electronic systems defines generic requirements for structuring and testing of software systems, ISO [2009] as the standard for automotive systems requires developers to utilize MDD-based methods to manage the complexity of the developed software and to increase maintainability [Bell 2006; ISO 2009; IEC 2010].

Software Safety Risk Classification Schemes

To define a common basis for comparison, IEC [2010] defines *Safety Integrity Level*s (SILs) that help developers in determining the necessary measures that need to be taken to ensure the correct operation of software systems. According to the dangers arising from a possible failure of a component, one of four SILs is assigned, ranging from the lowest level SIL 1 to the highest SIL 4. The higher the SIL assigned to a component, the more rigorous the testing and verification process needs to be in the development phase until a tolerable level of failure probability is reached [Bell 2006; IEC 2010]. ISO [2009]

builds on this concept by introducing *Automotive Safety Integrity Levels* (ASILs), ranging from the lowest ASIL A to the highest ASIL D for safety-critical components with the lowest to the highest automotive hazard. Additionally, a level *Quality Management* (QM) is introduced, which defines a base level for quality management that is necessary to be administered. In contrast to SILs, ASILs are not defined quantitatively, but qualitatively, i.e., a component level is determined via:

$$\begin{aligned} &\text{Risk} = \text{Expected severity in case of failure} \times \text{Probability of failure}, \quad \text{or} \\ &\text{ASIL} = \text{Severity} \times \text{Exposure} \times \text{Controllability} \end{aligned}$$

For every level and every step in the development process, ISO [ibid.] defines mechanisms required to be applied to mitigate the respective risk of failure of each component and the severity in case of such a failure. The highest levels, ASIL C and D, for example, require the use of unambiguous graphical representations of the software as well as the utilization of formal analysis techniques to gather information about data and control flow through the software. In addition to the assignment of an ASIL to each individual component, a connection between multiple components, i.e., the possibility for information flow between them, infers the requirement to develop and maintain all components on the level of the component with the highest ASIL. Only if a safe partitioning between components can be shown, ISO [ibid.] allows this requirement to be relaxed.

2.2 Signal-Flow-Oriented Modeling Languages

The largest number of embedded systems and internet-connected cyber physical systems are *reactive systems* [Berry 1989; Sander 2003]. Contrary to *interactive systems*, such as browsers or word processors, these reactive systems continuously react to their environment at the speed of the environment. In addition to the aspect of reactivity, embedded systems, such as, e.g., automotive control systems, are required to continuously perform computations on numerous incoming signals concurrently. In the design of safety-critical embedded systems, the *signal-flow-oriented* programming paradigm [Misra 2004; Kuo and Golnaraghi 2009], which naturally combines both aspects, has become the main development style. In the following, we first give a short introduction to the basic concept of signal-flow graphs. Subsequently, we present two modeling languages based on signal-flow-oriented semantics, MATLAB Simulink/Stateflow and Modelica, which we use as

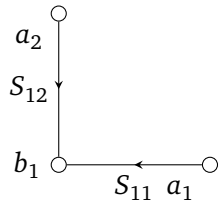


Figure 2.1: Signal-flow graph for Equation (2.1)

example languages in this thesis to present our information flow analysis method. Both languages are widely used in the domain of safety-critical systems and offer graph-based, signal-flow-oriented modeling of software systems.

2.2.1 Signal-Flow Graphs

SFGs are graphical representations of signal processing algorithms that display the relationship between variables in sets of linear algebraic equations and consist of *nodes* interconnected by directed branches [Mason 1953; Abrahams 1965]. The nodes represent variables or parameters of the differential equations, the branches act as coefficients connecting the variables. Figure 2.1 shows an example of an SFG that implements the equation

$$b_1 = S_{11}a_1 + S_{12}a_2 \tag{2.1}$$

The corresponding SFG is comprised of three nodes (b_1, a_1, a_2) and two branches. The arrows on the two branches are directed from the *causes* a_1 and a_2 towards the *effect* b_1, modified by the coefficients S_{11} and S_{12}. As is shown there, the operations performed at nodes and branches are implicit: a multiplication with the corresponding coefficients on the branches, and a summing operation on a unification of multiple branches at a node.

2.2.2 Syntax of Signal-Flow-Oriented Modeling Languages

Signal-flow-oriented modeling languages employ *blocks* which are connected using *signals*. Additionally, each block and signal is assigned a set of *parameters*. When relating a signal-flow-oriented *program* to traditional text-based programming languages, the signals connecting blocks are variables. The values of these variables are determined by the

blocks they are connected to, which represent functions defined over a continuum, i.e., the output value of a block is not only dependent on its current inputs but also on an internal state [Lee and Neuendorffer 2005].

A signal-flow-oriented model is defined, analogously to Boström and Morel [2007], Boström et al. [2007], and Zander-Nowicka [2009], as a tuple

$$\mathscr{M} = (B, root, sub_h, P, rlt, sig, subi, subo, C), \text{where :}$$

B represents the set of blocks in the system. Depending on the functionality of a block, they are part of one of the following categories: B^s for subsystem blocks, B^i for inports to a subsystem, B^o for subsystem outports, merge blocks B^m, blocks that contain an internal state B^{mem}, and basic, or *direct feed-through*, blocks B^b. Additionally, every subsystem is either *virtual*, B^{vs} or *non-virtual*, B^{ns}, such that $B^s = B^{vs} \cup B^{ns}$. Virtual blocks, such as inports to a subsystem or subsystems themselves, do not influence the simulation behavior of a signal-flow-oriented model, while non-virtual blocks do. Virtual blocks are purely used to structure the model and do not influence the behavioral semantics;

$root \in B^{vs}$ represents the root subsystem;

$sub_h : B \rightarrow B^s$ defines a function representing the subsystem hierarchy of the system. For every block $b \in B$, it returns the subsystem block $b^s \in B^s$ that it is structured into;

P is the set of ports that input, $P^i \subseteq P$, and output, $P^o \subseteq P$, data to and from blocks, $P^i \cup P^o = P$;

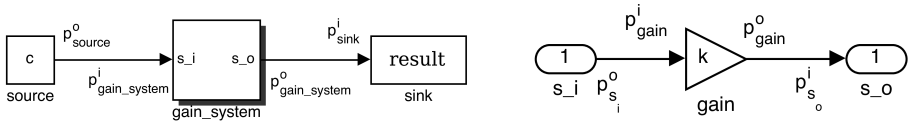
$rlt : P^i \rightarrow P^o$ is the function that maps every port to its corresponding block;

$sig : P^o \rightarrow P^i$ is a relation that maps every outgoing port to the in-going port it is connected to via a signal line;

$sub_i : B^s \rightarrow P \rightarrow \rho(P^i)$ is a partial function that maps the virtual inports of a subsystem to the non-virtual block driving it;

$sub_o : B^s \rightarrow P \rightarrow \rho(P^i)$ describes a partial function that maps the virtual outports of a subsystem to the non-virtual block that it drives;

C is the set of simulation parameters of the model.



(a) Subsystem example **(b)** Example subsystem content

Figure 2.2: Simulink example model.

2.2.3 MATLAB Simulink/Stateflow

MATLAB by The MathWorks [The MathWorks 2017b] is a multi-purpose numerical computing environment. In addition to this core functionality, MATLAB offers functionalities in the areas of, among others, data analysis and visualization, signal and image processing, financial modeling, or computational biology.

Simulink

Simulink is an add-on to MATLAB that enables multi-domain modeling and simulation of reactive control systems, offering an interactive graphical development environment for the model-based development of dynamic control systems. Simulink employs the syntax presented in Section 2.2.2.

Note that on the relations and functions defined in Section 2.2.2, there are several restrictions in order for a model to be considered a valid Simulink model. These restrict, e.g., the subsystem hierarchy and the signal connections over subsystem boundaries. In this thesis, we restrict our analysis to only support valid Simulink models, i.e., models that can be drawn using the graphical development environment and subsequently simulated. Invalid models that can be drawn but not simulated contain, e.g., unconnected signal lines.

Consider the example shown in Figure 2.2. The blocks are defined by

$$B \triangleq \{\text{source}, \text{gain\_system}, s_i, \text{gain}, s_o, \text{sink}\}.$$

The set of subsystems is given as

$$B^{vs} \triangleq \{\text{gain\_system}, root\}, \quad \text{and} \quad B^{ns} \triangleq \emptyset.$$

with the hierarchy definition

$$sub_h \triangleq \{(\text{gain}, \text{gain\_system}), (\text{gain\_system}, root), (\text{source}, root), \dots\}.$$

When displaying Simulink diagrams, port names are usually omitted. Here, they are given as

$$P = \left\{ p^o_{\text{source}}, p^i_{\text{sink}}, p^i_{\text{gain\_system}}, p^o_{\text{gain\_system}}, \dots \right\}.$$

The function describing which port is part of which block is given as:

$$blk \triangleq \left\{ (p^o_{\text{source}}, \text{source}), (p^i_{\text{gain}}, \text{gain}), (p^o_{\text{gain}}, \text{gain}), (p^i_{\text{gain\_system}}, \text{gain\_subsystem}), \dots \right\}.$$

The connections between the ports is defined as

$$sig \triangleq \left\{ (p^i_{\text{gain\_system}}, p^o_{\text{source}}), (p^i_{\text{gain}}, p^o_{s_i}), \dots \right\}.$$

The relations describing how ports in inports and outports correspond to ports of subsystems are given by sub_i and sub_o. The inport of the subsystem is related to the outport of the in-block:

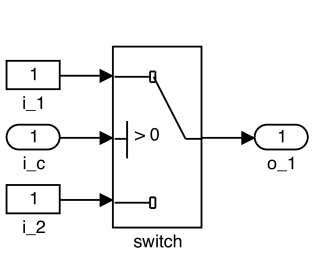
$$sub_i \triangleq \left\{ \left(\text{gain\_subsystem}, p^o_{s_i}, \{ p^i_{\text{gain\_system}} \} \right), \dots \right\}.$$

The definition of the outputs of the subsystem is similar:

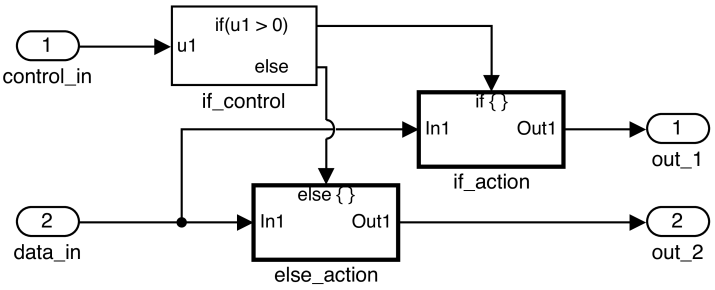
$$sub_o \triangleq \left\{ \left(\text{gain\_subsystem}, p^o_{\text{gain\_system}}, \{ p^i_{s_o} \} \right), \dots \right\}.$$

Signal Flow. Signal flow in Simulink is modeled using signals, or *signal lines*, connecting blocks. Signals carry the current value on the outports of blocks to inports of other blocks. Signals in Simulink are able to carry more than a single primitive value. According to the design of the model, signals carry either scalars, vectors, matrices, or even hierarchies of values. The precise type is not set statically, but is inferred by the underlying simulator at run-time according to the blocks driving the signals.

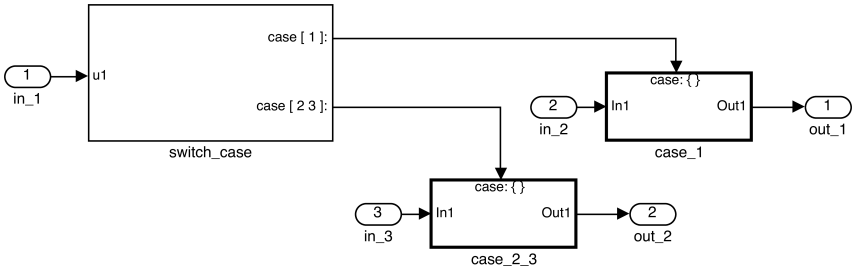
Signal flow through hierarchical Simulink subsystems is modeled using dedicated *virtual* blocks. While these blocks of types `InPort` and `OutPort` do not modify the signals connected to them, they serve as connectors in the model which connect levels in the model hierarchy to each other. Signals leaving a subsystem are connected to `OutPorts`, which consequently do not have out-going ports. For each `OutPort` block, an out-going port is added to the subsystem, which in turn is connected via an `InPort` to the surrounding model environment. Figure 2.2b shows one `InPort` and `OutPort`, respectively, connecting the subsystem to its parent hierarchy level.



(a) Example model containing a `Switch` block to route signals



(b) Example model containing a set of conditionally executed subsystems implementing an `if-then-else` structure



(c) Control flow modeling using `Switch-Case` subsystems

Figure 2.3: Examples of control flow modeling in Simulink

Control Flow. Control flow in Simulink is modeled using two different concepts. The first is the utilization of routing blocks B^R, such as `Switch` and `MultiPortSwitch`, which route one signal from a set of signals connected to the in-going ports of the routing block to a single out-going signal, depending on the current value of the signal connected to an in-going control port. Figure 2.3a shows an example model consisting of a `Switch` block that connects s_{i_1} or s_{i_2} to its out-going signal s_o if $s_c > 0$ or $s_c \leq 0$, respectively. The second concept to model control flow is conditionally executed `If-Action` and `Switch-Case` subsystems. Control flow utilizing `If` type subsystems consists of at least two `If-Action` subsystems connected to a *control block*, as shown in the example in Figure 2.3b. As is shown there, according to conditions encoded in the control block, the execution of exactly one connected `If-Action` subsystem is triggered. A model utilizing the `Switch-Case` subsystems is modeled similarly, with the control block implementing a *switch* semantics based on a single in-going signal and triggering the execution of corresponding connected subsystems. Due to the underlying signal-flow semantics, the simulator has to calculate a value for every block and output at every simulation step.

In case of `If-Action` subsystems not triggered during the current simulation step, their output is reset to 0. In case of Figure 2.3b, if the subsystem `if_action` is triggered, the value of outport `out_2` is equal to 0.

Simulation MATLAB Simulink Models. Simulation of a Simulink model is performed using *solvers*, which approximate the output of each block according to its semantics. Solvers for *time-discrete* as well as *time-continuous* interpretations of the block diagram exist. While the former compute the output for each simulation step based only on the current state of each block, the latter utilize numerical techniques to compute the state of each block based on multiple states and their derivatives. Further, solvers can be classified by the type of step size used in their calculation of the simulation state: *Variable-step* solvers aim at automatically finding a simulation step size for each block in the model to achieve a level of precision set by the model developer. *Fixed-step* solvers omit this step at the expense of precision while increasing performance. The former class of solvers is commonly used for hybrid or purely time-continuous systems that form differential equations, while the latter is used for time-discrete models forming difference equations. For the development of embedded control software, i.e., when code is generated from the developed models, solvers with a fixed time step size are used [Conrad 2004; The MathWorks 2017e]. In this thesis, our focus is the analysis of such discretely-timed models.

Stateflow

Stateflow [The MathWorks 2017d] is an extension to the MATLAB Simulink framework that enables the modeling of decision logic using a semantics based on Statecharts, originally introduced by Harel [1987]. Stateflow automata are hierarchical state machines composed of states labeled with actions and transitions labeled with guards as well as actions. Stateflow state machines are commonly used to model discrete control logic and modal behavior of a system.

The syntax of a Stateflow automaton *SF* given by Tiwari [2002] is described by a tuple $SF = (D, E, S, T, f)$, where:

$D = D_I \cup D_O \cup D_L$ is a finite set of typed variables partitioned into input variables D_I, output variables D_O and local variables D_L;

$E = E_I \cup E_O \cup E_L$ is a finite set of *events* partitioned into input events E_I, output events E_O and local events E_L;

S is a finite set of states, in which each state contains three set of actions: *entry*, *exit*, and *during*. An action can either be a variable assignment, as in imperative programming languages, or the broadcast of an event;

T is a finite set of transitions, each given as a tuple (src, dst, e, c, ca, ta), in which $src \in S$ is the source state, $dst \in S$ is the destination state, $e \in E \cup \{\epsilon\}$ is an event, $c \in WFF(D)$ is a condition given as a well-formed formula in predicate logic over the variables D, and ca, ta are sets of *transition* and *condition* actions that are triggered when the transition is taken or a condition is evaluated to true, respectively;

$h : S \to \left(\{and, or\} \times 2^S\right)$ is a mapping from the states to the Cartesian product of $\{and, or\}$ with the power set of S, which describes the hierarchy of the Stateflow chart. It satisfies the following properties: (1) there exists a unique root state s^{root}, i.e., $s^{root} \notin \cup_i$ *descendants*(s_i), where $descendants(s^i)$ is the second component of $h(s_i)$, (2) every non-root state s has exactly one ancestor state, that is, if $s \in$ *descendants*(s_1) and $s \in$ *descendants*(s_2), then $s_1 = s_2$, and (3) the function h contains no cycles, i.e., the relation $<$ on S defined by $s_1 < s_2$ iff $s_1 \in$ *descendants*(s_2) is a strict partial order. If $h(s) = (and, \{s_1, s_2\})$, then the state s is an AND-state consisting of two substates s_1 and s_2. If $h(s) = (or, \{s_1, s_2\})$, then s is an OR-state with substates s_1 and s_2.

Simulation of Stateflow Automata. During simulation of a Stateflow automaton, the configuration describing the current state of the automaton $C \in 2^S \times \mathscr{D}$ is a tuple consisting of the set of active states and a valuation of all variables in D, denoted by $\mathscr{D}$. If a non-leaf OR-state is active, then exactly one of its descendant substates is active, and if a non-leaf AND-state is active, then every descendant substate is active. The set of all configurations that satisfy these conditions, denoted by $\mathscr{C}$, is called the set of valid configurations. The Stateflow semantics is given by a function $|SF| = \mathscr{C} \times \mathscr{D}_I \times E_I \to \mathscr{C}$. This function maps a configuration, a valuation of the input variables, and an input event to a new configuration.

This semantics is only provided informally by the Stateflow specification [The MathWorks 2017d]. An input event e triggers the execution of the initial state. A state executes by performing its corresponding *entry* actions and *firing* all of its transitions that can be fired. If none of its transitions can be fired, the state triggers executions of its descendant states: either one or all, depending on the state being an OR-state or an AND-state. A

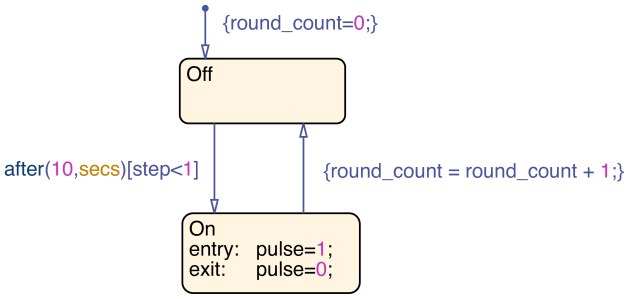


Figure 2.4: Example of a Stateflow automaton

transition $t = (src, dst, e, c, ca, ta)$ can be fired if (1) event e is present, (2) condition c is evaluated to true, and (3) state *src* is currently active. If a transition executes, it preempts execution of state *src*, executes its corresponding condition actions *ca*, enters state *dst*, and executes its transition actions *ta*. Whenever an assignment action $x := expr$ is executed, the variable $expr \in D$ will be assigned the value x. An event broadcast can be considered similar to a function call in imperative programming languages, as it triggers execution of the states and transitions accepting the event.

Consider the example given in Figure 2.4. The automaton shown there utilizes two states *Off* and `On`. The initial state, `Off`, is set using the default transition from the ● mark. When the automaton is initialized, the value of the output variable `round_count` is set to 0. Shown in square brackets on the out-going transition is its *transition guard*, that only allows activation of the next state if its corresponding condition is fulfilled. To progress to state `On` from `Off` the condition `step < 1` has to be fulfilled. Furthermore, this condition is only evaluated 10 s after state `Off` has been activated, i.e., after the *temporal logic condition* is fulfilled. Upon entering state `On`, its `entry` action is executed, setting the value of the internal variable `pulse` to 1. Upon exiting the state, its `exit` action is executed, setting `pulse` to 0. As its out-going transition is not marked by a condition, the state will always be left one simulation step after it has been entered. When taking this transition, its transition action is executed and the output variable `round_count` incremented by 1.

2.2.4 Modelica

Modelica is a modeling language that is used for the specification of mathematical models of complex systems whose behavior evolves as a function of time [Fritzson 2004]. First introduced in Fritzson and Engelson [1998], Modelica implements non-causal modeling

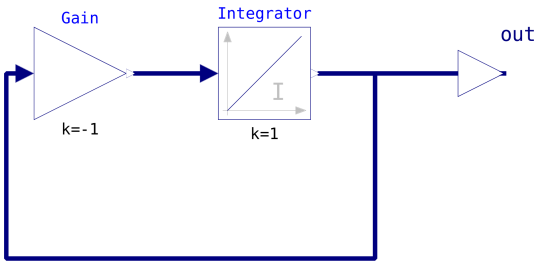


Figure 2.5: Graphical model representation

of signal-flow graphs, similar to that of MATLAB Simulink, i.e., based on equations instead of assignment statements. This enables reusability of components in multiple situations. In addition to this concept, Modelica is an object-oriented language with a generic class concept and implements *generics* (similar to *templates* in C++) and subtyping to further increase reusability of components [Modelica Association 2017]. Finally, it supports the modeling and simulation of physical systems with components from multiple domains. A set of libraries for each domain as well as a connecting layer that, e.g., resolves signal units, allows modeling of systems which share, e.g., mechanical, hydraulic, or electrical components. The syntax of Modelica is, analogous to MATLAB Simulink, defined as presented in Section 2.2.2.

```
model SimpleExample
  Modelica.Blocks.Math.Gain Gain(k = -1)
    annotation(Placement(/* omitted */));
  Modelica.Blocks.Continuous.Integrator Integrator(y_start = 1)
    annotation(Placement(/* omitted */)));
  Modelica.Blocks.Interfaces.RealOutput out
    annotation(Placement(/* omitted */)));
equation
  connect(out, Integrator.y) annotation(
    Line(/* omitted */)));
  connect(Gain.u, Integrator.y) annotation(
    Line(/* omitted */)));
  connect(Gain.y, Integrator.u) annotation(
    Line(/* omitted */)));
  annotation(
    uses(Modelica(version = "3.2.2")));
end SimpleExample;
```

Listing 2.1: Textual representation of the Modelica example model with layouting information omitted

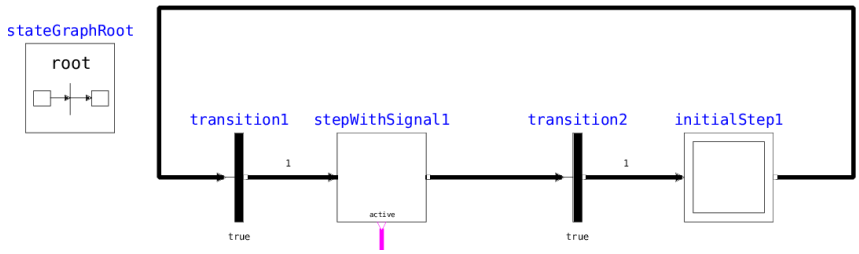


Figure 2.6: StateGraph example

An example of a Modelica model can be seen in Figure 2.5. Shown there is, analogous to functionality available in MATLAB/Simulink, a `Gain` block multiplying its incoming signal by -1, and an `Integrator` block, summing the values of its incoming signals from the start of the simulation. The internal state of the `Integrator` at the start of the simulation, i.e., the initial value necessary to solve the underlying difference equation, is set to 1. Listing 2.1 shows the textual representation of the system that is created automatically when designing the system. Shown there are the blocks comprising the system and the connections between them defining the functionality.

Due to the focus on component reusability and structural modeling, Modelica has found widespread adoption in the domain of high-level architectural design of safety-critical embedded systems [Tiller et al. 2003; Brückmann et al. 2009; Chrisofakis et al. 2011; Sutherland et al. 2016].

State-Machine-Based Modeling in Modelica

Modeling of state machines in Modelica is possible via the StateGraph library [Donath et al. 2008; The Modelica Association 2019b]. The blocks in this library offer the possibility to model finite state machines based on `Steps` and `Transitions`, i.e., states and edges, respectively. An example is shown in Figure 2.6. As can be seen there, steps represent to the possible states of a StateGraph model [Otter et al. 2005]. If a step is active, a Boolean variable corresponding to the step is true. Initially, all steps are deactivated, and the `initialStep` object is activated. This initial step is characterized by a double box. The state of the model is changed using transitions. When a step is activated, its outgoing transition is initially deactivated, and only activates once its transition condition is evaluated to *true*.

In addition to these basic modeling possibilities, controllers in StateGraph support parallel execution of transition, i.e., multiple states can be active at the same time.

2.3 Information Flow Analysis

The protection of confidentiality of information inside a software system as well as the protection from modification by unauthorized sources is a long-standing and increasingly important problem in the area of general computing as well as of embedded, and, in particular cyber-physical systems. Techniques that assert whether a system complies with a set of *security properties* are grouped under the term *information flow control* (IFC) [Hammer 2009].

When analyzing the security of a software system, the following main dimensions commonly referred to as the *CIA triad* [Saltzer and Schroeder 1975; Cherdantseva and Hilton 2013], can be identified:

Confidentiality states that every party in a system is only able to read data according to its specified security level, i.e., that only authorized recipients are able to read data marked as confidential.

Integrity ensures that critical computations cannot be manipulated from the outside or that data has not been altered during transmission from source to target.

Availability guarantees that information or resources are available to authorized users when required, i.e., according to a specified extent and timing.

As first noted in Biba [1977], integrity can be considered a dual to confidentiality and both can be enforced by controlling *information flow* through a program. As explained above, confidentiality prohibits the flow of information to inappropriate sources while integrity requires that information is prevented to flow from inappropriate sources. To identify violations of either property, the flow of information through software systems has to be analyzed.

2.3.1 Types of Information Flow

Within a software system, different kinds of information flow can be observed: *explicit* and *implicit* flows. The former describes flows of information through variables that are explicitly stated in the source code of the software under analysis, i.e., information is explicitly leaked to a publicly observable variable. An example, shown in Listing 2.2, uses two variables, h and l of differing security levels *HIGH* and *LOW*, respectively. As is shown there, the value of the private variable h is assigned to the publicly observable, lower-security variable l, a violation of confidentiality.

```
1 var l, h;
2 l := h;
```

Listing 2.2: An explicit information flow from variable h to variable l

The second type of information flow, *implicit flow*, can manifest in a number of ways:

Control flow With knowledge of the program source code and structure, an observer is able to deduce the values of private variables by observing the control flow of a program. An example for a flow of this kind is shown in Listing 2.3. There, the value of the private variable h is never directly assigned to a publicly observable variable but an observation of the public value of l allows deduction of h [Denning and Denning 1977].

Timing Similarly, by using knowledge of the program structure, this implicit flow allows the deduction of the values of private variables by analyzing the execution time of the software for different sets of input values. As the example in Listing 2.4 shows, the execution time allows an observer to draw conclusions about the value of h, albeit not being directly assigned to a publicly observable variable [Kocher 1996].

Power By measuring the power consumption of the system executing the software under analysis, e.g., a cryptographic algorithm, an observer can deduce private information, such as keys utilized in the computation [Kocher et al. 1999; Singh et al. 2017].

```
1 h := h mod 2;
2 l := 0;
3 if h == 1 then
4   l := 1;
```

Listing 2.3: Implicit information flow through the control structure of a program

Which type of information flow is a concern to the developer depends on what attackers are able to observe. For example, smart cards or *Near Field Communication* (NFC) devices draw power from the potentially untrusted device they are inserted into or come into contact with, which makes it necessary to mask the power-based information flow in such systems.

```
1  var l, h;
2  if h == 1 then
3    // execute time-consuming function f()
4    f();
5  l := 0;
```

Listing 2.4: Implicit information flow through program timing

To be able to analyze information flow through a program, a variety of methods and approaches has been developed since the inception of the field in Lampson [1973]. These IFC approaches aim to make it possible to follow implicit as well as explicit flows of information through software and are additionally used to *control* the flow of information. Early works in the field, such as Fenton [1973] or Bell and LaPadula [1973], developed *mandatory access control*. In this approach, variables in a program are assigned *security levels*, and during the course of the execution of the program, an additional software layer calculates the dissemination of the data. In addition to its computational and storage overhead, this method has proven to be too restrictive for use in general code as no sharing of information between security levels is allowed. A further flaw of these purely run-time enforcement mechanisms lies in its inability to identify *implicit* information flow,.

2.3.2 The Lattice Model of Secure Information Flow

In software systems, data is commonly classified into *security levels* $L = \{l_1, \ldots, l_n\}$. To express security policies, i.e., rules describing between which levels information is permitted to flow and has to be prevented, respectively, the following relations can be defined [Hammer 2009]:

$\rightsquigarrow$: $L \times L$ is the transitive, reflexive and antisymmetric *interference relation*. $x \rightsquigarrow y$ expresses that information from class x is only permitted to flow into class y.

$\not\rightsquigarrow$: $L^2 \setminus \rightsquigarrow$ is the complement *noninterference relation*. $x \not\rightsquigarrow y$ denotes that information flow from class x to class y must be prevented in the system.

In an early approach to define a mathematical framework to formulate requirements for secure information flow between sets of security levels, Denning [1976] identified *lattices* to naturally express relations between security levels. If these levels are arranged in a lattice, the relation $\rightsquigarrow$ is equivalent to $\leq$. If a direct comparison between security

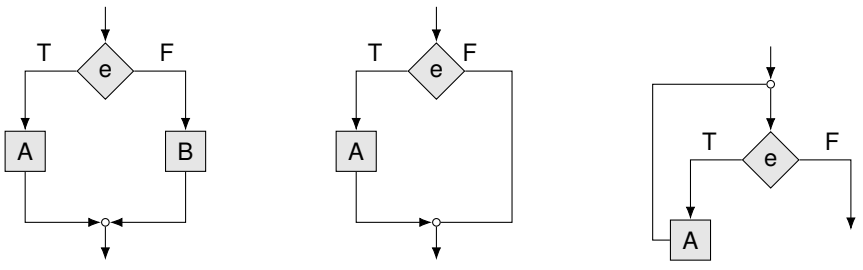
(a) A linear priority lattice **(b)** A lattice with incomparable elements

Figure 2.7: Examples of lattice representations of security models

levels is not possible, a complete lattice $\mathscr{L} = \{L, \leq, \bot, \top, \sqcup, \sqcap\}$ can be defined. Figure 2.7 shows examples of both situations. In Figure 2.7a, an example of a linear priority lattice, the three security levels Confidential (0), Secret (1) and Top Secret (2) are ordered in a linear fashion and a direct comparison between every level is possible, i.e., $0 \leq 1 \leq 2$. Figure 2.7b shows the more complex example of a security classification, in which a direct comparison between levels is not possible in every case, i.e., neither $x \leq y$ nor $y \leq x$. In this case, the supremum operator $\sqcup$ defines the resulting security level when two pieces of information from incomparable levels are combined. As the example in Figure 2.7a shows, when information from level x and y are joined, the resulting level must be z.

2.3.3 Information Flow Control

An early method to track explicit as well as implicit flows on the source code level has been introduced in Bergeretti and Carré [1985]. In this work, the authors describe an IFA technique based on data flow equations, which are heavily utilized in the program optimization steps of compilers [Nielson et al. 1999].



(a) if e then A else B **(b)** if e then A **(c)** while e do A

Figure 2.8: Control-flow graphs of program statements [Bergeretti and Carré 1985]

These information flow equations are constructed with special consideration of the three main program flow elements, shown in Figure 2.8. For each structure, the authors present equations to calculate three properties of information flow. For a variable $v \in V$ and the expression $e \in E$ in the statement S, the first property $v\,\lambda_S\,e$ states that the *v may be used* in the evaluation of e in S. The example in Listing 2.5 contains two expressions, `y > 0` and `y + z`. The value of y *may* be used in the evaluation of both expressions, while the value of z *may be used* be used in the expression `y + z` only. A more complex example for this property is shown in Listing 2.6. There, the value of x *may be used* in the evaluation of the final expression `2 * y` as depending on the value of x, an assignment to y will have been made in the previous statement $y := 1$ and this value will be used in the expression. This implicit flow of information between variables x and z therefore becomes visible.

```
1 if y > 0 then
2   x := y + z;
```

Listing 2.5: A simple example of the λ property

```
1 if x > 0 then
2   y := 1;
3   z := 2 * y;
```

Listing 2.6: A more complex example of the λ property

The second property $e\,\mu_S\,v$ signifies that a value of the expression e in S *may be used* in obtaining the value of v on exit from S. An example is shown in Listing 2.7. Both the expressions `w > 0` and `y + z` may be used in obtaining the value of the variable x on exit from the presented statement. In case of a successful evaluation of the first expression, the value of x on exit from S is the sum of y and z, otherwise it will retain its original value. In either case, both expressions *may be used*.

```
1 if w > 0 then
2   x := y + z;
```

Listing 2.7: A simple μ property example

Finally, the third property $v\ \rho_S\ v'$ describes a relation between two variables on entry and on exit of the statement S. The value of v *may be used* in obtaining the result of an expression e in S, which in turn *may be used* in obtaining the value of v' on exit from S. It can be expressed as $\rho_S = \lambda_S \mu_S$ and therefore forms the necessary connection to put two variables in a single program construct S into an information flow relation to each other. The authors further extend the concept to the connection of multiple program statements to be able to calculate the information flow between arbitrary variables in the source code.

As is shown in these examples, this form of *static* information flow analysis is able to detect explicit as well as implicit flows of information in a given program. In comparison to dynamic IFA approaches, such as Suh et al. [2004], which are only able to track the flow of information over a single execution of a program, static methods identify flows over *all* executions and paths.

2.3.4 Non-Interference

If data inside a program is to be kept confidential, its developer might create a security policy stating that the computation of this secret data is not affected by publicly observable inputs or outputs of the program. This allows secret data to be calculated and modified inside the program as long as visible outputs do not reveal any information about this data. Such a policy is called a *non-interference* policy, first introduced in Goguen and Meseguer [1982]. A usual method to show that a non-interference policy holds is to demonstrate that an observer of the public variables cannot distinguish between two executions of the program that only differ in their confidential inputs [Goguen and Meseguer 1984; Sabelfeld and Myers 2003], i.e., no information flow from the secret inputs to the publicly observable outputs is allowed. In this manner, information flow control techniques can be used to prove non-interference [Hammer and Snelting 2009].

2.3.5 Path Conditions

As explained in Section 2.3.3, a static information flow analysis is able to detect both implicit as well as explicit flows through a program due to its analysis of all paths through said program. Therefore, these analysis techniques are inherently *safe* with regard to their ability to detect information flow. However, to remain computable, such analyses only provide approximate answers [Nielson et al. 1999], i.e., they present an over-approximation of the actual solution. To increase the precision of a static information

```
1  a[i + 3] = high;
2  if (i > 10)
3    low = a[2 * j - 42];
```

Listing 2.8: Example utilization of path conditions in information flow analysis

flow analysis, King [1976] introduced *path conditions* that describe necessary conditions for paths to be executed. In Hammer et al. [2006, 2008], path conditions are used to capture all paths where information might flow from a source to a target.

Consider the example given in Listing 2.8. As is shown there, an element of the array `a` serves as storage for the value of `high`, in this example a variable of a high security level. Inside the `if` scope, the variable of low security, `low`, is assigned a value of the array. Given the policy that no high security information is to be leaked into a lower security domain, a static analysis, as presented in Section 2.3.3, detects a policy violation in the two assignments as information is able to flow through the array. However, when analyzing the condition of the `if` clause, it becomes apparent that information flow through the array only occurs for certain ranges of the variables `i` and `j`. It is therefore possible to express *path conditions* for the flow of information between Line 1 and Line 3 of the given example. Only if `i > 10`, the assignment to the low security variable occurs and a policy violation occurs. The following describes the path condition of information flow between Line 1 and Line 3 of Listing 2.8:

$$\begin{aligned} PC(1 \rightarrow 3) &= \exists\, i, j \big((i > 10) \wedge (i + 3 = 2j - 42) \big) \\ &= \mathit{true} \end{aligned}$$

```
1  a[i + 3] = high;
2  if ((i > 10) && (j < 5))
3    low = a[2 * j - 42];
```

Listing 2.9: More complex example utilization of path conditions in information flow analysis

A slightly more complex example is shown in Listing 2.9. There, the information flow condition is extended to include the variable `j` and the path condition is expressed in the following:

$$\begin{aligned} PC(1 \rightarrow 3) &= \exists\, i, j\big((i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42)\big) \\ &= \mathit{false} \end{aligned}$$

As opposed to Listing 2.8, this extended path condition allows a safe conclusion about the existence of information flow between the high and low security variables. In fact, as there is no assignment of values to `i` and `j` such that the additional conditions are fulfilled. Information flow is therefore impossible.

2.4 Constraint Logic Programming

In the following, we give a brief introduction to the concept of *Constraint Logic Programming* (CLP). In this thesis, we utilize CLP to identify solutions to sets of path conditions extracted from the signal-flow-oriented components of control system models under analysis.

CLP was introduced in Jaffar and Lassez [1987] and extends *logic programming* to include the concept of *constraints*. In general, CLP can be seen as a technique to solve CSPs, which are defined as a triple $\mathscr{P} = (V, D, C)$ consisting of [Frühwirth et al. 1992; Niederliński 2011]:

- a finite set of variables $V = \{v_1, \ldots, v_n\}$;
- a set of domains $D = \{D_1, \ldots, D_n\}$ with $\{v_1 : D_1, \ldots, v_n : D_n\}$;
- a set of constraints $C_j(V_j), j \in \{1, \ldots, m\}$ with each constraint establishing a relation of a subset of variables $V_j = \{v_{j_1}, \ldots, v_{j_k}\} \subseteq V$ to each other and to solutions from a subset of $D_{j_1} \times \cdots \times D_{j_k}$.

A solution to a CSP is given by any assignment of domain values to variables that satisfies all constraints. It may be *non-unique* or *unique*; Additionally, CSPs may contain an *objective function* which, as an additional part of the solution, is to be minimized or maximized. In this case, they are referred to as *Constraint Optimization Problems* (COPs) and their solutions as *optimum* solutions. If the domains D are restricted to *finite* domains, the problem is called *a finite constraint satisfaction problem* (FCSP) [Mackworth

Figure 2.9: Map of the seven states and territories of Australia [Marriott et al. 1998]

and Freuder 1993]. As in this case the set of variables as well as each domains is finite, the respective solution as a subset of $D = \{D_1, \ldots, D_n\}$ is finite as well. In Haralick and Shapiro [1979], the authors have shown the finite constraint satisfaction problem to be NP-complete.

An example of a constraint satisfaction problem is introduced in Figure 2.9. In this problem, each of the seven Australian states and territories are to be assigned one of three colors such that no adjacent areas share a color. The formulation of this task as a CSP can be seen in Listing 2.10, expressed using the `MiniZinc` [Marriott et al. 1998; Marriott and Stuckey 2013] constraint solving language, which has been chosen here due to its readability and concision. Before declaring the variables in the example, the number of colors is defined, as is shown in Line 1. Each variable, representing the individual areas, is declared to lie in the domain $\{x \in \mathbb{Z} \,|\, x \geq 1 \wedge x \leq nc\}$. The subsequent section of the code, beginning in Line 5, shows the constraints imposed on the solution. The value of each pair of adjacent areas is constrained to be non-equal, thereby forcing the solver to find an assignment of values to the variables that does not violate any of the presented constraints. Line 15 instructs the solver to find such a solution satisfying the set of constraints expressed in the code, and finally, Line 19 outputs the solution by printing the values of each variable.

To find a solution to this CSP, in this example, we utilized the `Gecode` [Schulte et al. 2010] constraint solving toolkit. The identified solution is shown in Listing 2.11.

```
int: nc = 3;
var 1..nc: wa; var 1..nc: nt; var 1..nc: sa; var 1..nc: q;
var 1..nc: nsw; var 1..nc: v; var 1..nc: t;

constraint wa  != nt;
constraint wa  != sa;
constraint nt  != sa;
constraint nt  != q;
constraint sa  != q;
constraint sa  != nsw;
constraint sa  != v;
constraint nsw != q;
constraint nsw != v;

solve satisfy;

output ["wa=", show(wa),  "\t nt=",  show(nt),  "\t sa=", show(sa), "\n",
        "q=",  show(q),   "\t nsw=", show(nsw), "\t v=",  show(v),  "\n",
        "t=",  show(t),   "\n"];
```

Listing 2.10: The coloring of Australian states and territories, expressed as a `MiniZinc` model

```
wa=2   nt=3  sa=1
 q=2  nsw=3   v=2
 t=1
```

Listing 2.11: The solution to the Australia CSP problem as output by the `Gecode` solver.

2.5 Model Checking

In the following, we briefly present model checking as a technique to formally verify properties on systems. Specifically, we discuss the UPPAAL timed automata verification framework, which we use a tool to verify properties on the control-flow-oriented components of the control systems we aim to analyze.

To verify the correct functionality of a software system, i.e., that the system adheres to its specifications under all circumstances, it can be verified formally. Formal verification methods, such as *model checking* cover all possible executions of the system under analysis. Model checking is an automated technique for verifying concurrent systems with a *finite* number of states. Independently developed in Clarke and Emerson [1981] and Queille and Sifakis [1982], *temporal logic model checking* allows for the modeling of software systems and circuit designs as state-transition systems and of requirements as

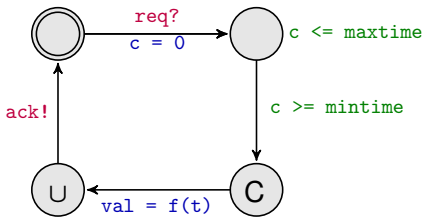


Figure 2.10: UPPAAL example

propositional temporal logic formulae. On these formal specifications, a search technique determines whether the formulae, i.e., the requirements hold. In other words, the transition system is checked to see whether it is a model of the formal specification [Clarke and Grumberg 1999].

2.5.1 UPPAAL Timed Automata

One technique to formally model systems is given by TA, first introduced in Alur and Dill [1994]. Timed automata are a timed extension of finite state automata that include a notion of time by introducing *clock variables*, which are used in clock constraints to model time-dependent behavior. To model concurrent systems, timed automata can be combined into networks, that communicate over and synchronize on *channels*.

To enable the automatic verification of systems of timed automata, UPPAAL [Behrmann et al. 2004; Bengtsson and Yi 2004] implements the timed automaton semantics and enables the graphical as well as text-based modeling, simulation, animation and verification of networks of timed automata. Additionally, UPPAAL extends the semantics of timed automata by bounded integer variables as well as binary channels, enabling a one-to-one synchronization, and broadcast channels, enabling a one-to-many communication and synchronization.

A small example UPPAAL timed automaton is shown in Figure 2.10. The symbol ⓞ denotes the initial location of the automaton. The label `req?` denotes that the transition is enabled as soon as the process receives on channel `req`, analogously, the label `ack!` denotes that whenever the transition is taken, the process emits on channel `ack`. The label `c = 0` denotes that on taking this transition, the clock `c` is reset to 0. Shown in green are two conditions on the clock `c`: the location invariant `c <= maxtime` and the transition condition `c >= mintime`. The semantics of the former states that the location must be left before clock `c` becomes greater than the constant `maxtime`. For the latter, the transition is not enabled before clock `c` is greater than or equal to `mintime`.

Additionally, the symbol Ⓤ depicts an urgent location and the symbol Ⓒ a committed location. Urgent and committed locations are used to model locations where no time may pass, i.e., no clock variables are increased.

The UPPAAL model checker enables fully-automatic verification of (unnested) *Computation Tree Logic* (CTL) formulae on a given network of timed automata as input *queries*, i.e., descriptions specifying conditions and properties on paths and their branches throughout the automaton under analysis.

2.6 Summary

In this chapter, we have presented the preliminaries that form the foundation of this thesis. We have started with a brief overview over MDD to provide an understanding of the realm this thesis is placed in. Subsequently, we have introduced the concept of signal-flow-oriented modeling languages as well as two example languages that have found widespread adoption in the domain of safety-critical systems. Further, we have presented the ideas behind information flow analysis as well as techniques to control information flow in software systems. We have closed with a brief overview of Constraint Logic Programming and model checking, specifically property checking using the UPPAAL verification framework.

In the next chapter, we discuss prior research work related to our approach.

3 Related Work

In the design of discrete embedded control system models, modeling languages based on the signal-flow-oriented programming paradigm are widely used. In this chapter, we first give an overview of information flow analyses for sequential, text-based programming languages in Section 3.1. In Section 3.2, we provide a similar overview for synchronous systems, which, due to their inherent timing and concurrency, possess semantics similar to those discrete embedded control system models.

In Sections 3.3 and 3.4, we discuss a number of verification and information flow analysis methodologies for control system models consisting of signal-flow-oriented and state-machine-based components. Finally, in Section 3.5, we present similar methodologies for the state-machine-based components of heterogeneous control system models. Our discussion of related work closes with a summary in Section 3.6.

3.1 Information Flow Analyses of Sequential Programs

As we have presented in Chapter 2, the protection of confidentiality of information inside a software system is a long-standing problem [Graham 1967; Lampson 1969]. For traditional text-based sequential programs, a wide range of information flow analyses have been developed, which are able to cope with the syntactical possibilities present in those language [Sabelfeld and Myers 2003; Hedin and Sabelfeld 2012]. Especially in the field of highly expressive object-oriented programming languages, represented by, e.g., C++ or Java, information flow is hidden by complex features such as object hierarchies, procedures, exceptions, or threads. In Hammer and Snelting [2009] and Hammer [2009], the authors present an exhaustive method to analyze the information flow through Java programs. Their method is based on the extraction of the *program dependence graph* (PDG) from the bytecode of a given Java program, on which they subsequently calculate the path conditions necessary for execution of a code fragment

under analysis. Based on this extracted PDG, they define a system of flow equations for various language features, e.g., exceptions and procedure calls. Using this system, they are able to precisely calculate the flow of information in Java programs. Similar analyses have been developed for languages such as C [Barany and Signoles 2017], Python [Chen et al. 2014], Assembly languages [De Francesco and Martini 2007], and JavaScript [Hedin et al. 2014]. However, as these methodologies target the inherently different semantics of sequential programming languages, which lack notions of timing and concurrency, they are not applicable to the domain of control system models. This holds especially true when considering the difference in abstraction between the presented languages and control system models. Consequently, the presented techniques to analyze information flow in sequential programming languages are unable to be applied to the problem stated in this thesis.

3.2 Information Flow Analyses of Synchronous Systems

Synchronous systems utilize a notion of non-relative timing and concurrency similar to that of signal-flow-oriented modeling languages. Such systems represent hardware circuits, synchronized by a global clock signal, and are usually developed using hardware description languages such as *Very High Speed Integrated Circuit Hardware Description Language* (VHDL). The *register-transfer level* (RTL), on which such systems are modeled, is an abstract representation of time-dependent circuit elements, i.e., registers, and of combinatorial elements, which perform logical operations on signals inside the system. In Tolstrup et al. [2005], the authors have developed an information flow analysis for a subset of VHDL. In VHDL, the functionality of a system is described by a set of concurrent statements. To emulate the physical aspect of the RTL, i.e., the propagation of an electrical current through the system, the concept of *delta cycles* is used, which describe simulation cycles in which the simulation time does not advance. Tolstrup et al. [ibid.] presents an adaption of the *Reaching Definitions* analysis originally from Nielson et al. [1999], which tracks the flow of variables and present values of signals with special consideration to the semantics of delta cycles. While this method is able to analyze information flow through the concurrent semantics of VHDL, it does neither take into account the timing of information passing through the system nor the precise control behavior responsible for the execution of information paths.

A similar methodology to analyze information flow in synchronous systems is presented in Köpf and Basin [2006]. There, the authors utilize a automata-based representation of synchronous circuits on which they identify timing side channels. They abstractly model the system behavior using Mealy automata synchronized by a global clock signal. The execution time of calculations is expressed in numbers of clock cycles. On an observer-based security model, they utilize model checking to analyze whether all possible executions of the system with respect to a secret input are indistinguishable to the observer. While this approach is able to identify timing side channels on small synchronous systems and limited signal bit-widths, the analysis efforts increases dramatically for larger models and signal bit-widths. Additionally, the observer model utilized does not allow for a precise determination of the timing behavior of individual model signals and cannot be applied automatically to given models.

To the best of our knowledge, no other methods to analyze information flow in synchronous languages, such as VHDL, exist. While there exist similarities in the semantics of synchronous languages and signal-flow-oriented modeling languages, these methodologies are not applicable to the problem stated in this thesis, as (1) they target a strongly different abstraction level than that of embedded control system models, and (2) they lack support for the analysis of the heterogeneity of combined signal-flow-oriented and state-machine-based models.

3.3 Information Flow Analyses of Control System Models

In the following, we present approaches most closely related to this thesis, i.e., information flow analysis methods based directly on the semantics of heterogeneous control system models consisting of signal-flow-oriented and state-machine-based components.

In Whalen et al. [2010], the authors present a method to analyze information flow in control system models implemented using MATLAB Simulink. Similar to the procedures presented above, a given model is translated to Lustre to enable formal verification of properties on the system. Based on a system of information flow equations, their method tracks the flow of *principal variables* through the model, and defines an information flow theorem on these flows. Subsequently, information flow properties, such as noninterference, are formulated using this theorem, and the Prover model checker [Prover Technologies AB 2019] is used to verify these properties. While the goals of their methodology are closely aligned to those of this thesis, specifically the development of an automated information flow analysis method for discrete signal-flow-oriented control

system models, their method does not consider important characteristics of the semantics of such modeling languages, namely the precise timing semantics and the heterogeneity of models combining signal-flow-oriented and state-machine-based components. Although time-dependent components are part of the case study presented in Whalen et al. [2010], the information flow theorem does not consider the timing semantics inherent to these components. What is more, the method is not able to analyze the influence of the integrated Stateflow controller on its surrounding signal-flow-oriented components. Finally, while the authors do not provide a discussion on the analysis performance of their approach, its translation of the *whole* model to a formally verifiable representation raises doubts about its applicability to control system models of industrially relevant size. We present a more detailed discussion on this in Section 3.4.1.

Finally, we discuss two methodologies, which have transferred the concept of *program slicing* [Weiser 1981] to the domain of control system models. While slicing cannot be considered a complete information flow analysis, it can be regarded as a tool to identify non-interference between components in a software system [Weiser 1982]. If a forward slice from an arbitrary program statement A does not include statement B, information flow from A to B is impossible and non-interference is guaranteed. In Reicherdt and Glesner [2012], the authors present a technique to transfer the concept of slicing to MATLAB Simulink. Their methodology bases the calculation of a slice on the *control flow graph* (CFG) of a given Simulink model, which they calculate based on data and control dependence relations they established as part of their work. To identify a slice for a given Simulink model element, they calculate post-dominance relations on the constructed CFG. Their case studies from the automotive domain indicate that, on average, the complexity of a model can be reduced by approximately 37 %. They subsequently utilize this circumstance in their model-checking-based verification methodology for MATLAB Simulink models [Reicherdt 2015]. A similar approach can be found in Gerlitz and Kowalewski [2016]. Their slicing methodology for MATLAB Simulink models increases the slicing precision by integrating an analysis of the virtual blocks used to implemented signal buses found in the Simulink syntax. Using their method, they are able to decrease the average model complexity by approximately 52 %. Both slicing-based approaches can be used to identify non-interference between arbitrary model elements. However, they present a coarse over-approximation of the precise behavior of the model as they neither include the timing or the control flow behavior of models, nor consider the influence of state-machine-based controllers on the signal-flow-oriented model components.

3.4 Formal Analyses of Signal-Flow-Oriented Modeling Languages

While the goals of the techniques presented in the previous section closely align with those of this thesis, they do not meet the criteria we have set in Chapter 1. Thus, the following presents a discussion of a wide range of formal analysis methodologies for signal-flow-oriented modeling languages.

3.4.1 Approaches Based on Model Checking Intermediate Languages

Extensive work has been done in the area of translating subsets of control system models combining signal-flow-oriented and state-machine-based components into formal languages with well-defined semantics.

A large research community has established itself around the translation of signal-flow-oriented models into Lustre and the graphical modeling suites SCADE [Abdulla et al. 2004] and SIGNAL [LeGuernic et al. 1991]. More specifically, due to its industrial relevance, the translation of subsets of MATLAB Simulink has been the focus of numerous approaches. Lustre [Halbwachs et al. 1991], a synchronous programming language based on the data-flow programming paradigm, is widely used in the design of safety-critical reactive systems, such as control systems in the aerospace domain. Similar to VHDL, a system is designed using concurrent statements, such as assignments or arithmetical and conditional operations, and structured using *nodes*. Each variable defines a possibly infinite sequence of values, or *flow*, over a global clock-like signal. Assignments to a variable can be performed at the same, previous, or subsequent position in the flow. Building on these similarities between Lustre and signal-flow-oriented modeling languages, the authors of Caspi et al. [2003] and Tripakis et al. [2005] present a set of block-wise translation rules from the Simulink `Discrete` library to Lustre, mapping Simulink blocks to nodes and signals to flows. Building on this work, in Dajani-Brown et al. [2004], the authors present a methodology to formally verify control system models developed using MATLAB Simulink using the translation to Lustre and its graphical model design counterpart SCADE. In SCADE, control systems are designed using a graphical interface not unlike that of Simulink, which acts as a front-end for Lustre. On the example of a small case study from the aerospace domain [Osder 1999], the authors utilize the SCADE Design Verifier to perform model checking of the translated model and are able to confirm the correct functionality of the Simulink model according to its requirements. In Joshi

and Heimdahl [2005], the authors utilize the case study of a wheel brake system from the automotive domain, implemented using Simulink. They manually extend the original model with a fault model, such that each component can enter a failure mode in order for the behavior of the surrounding components to be analyzed. Based on a manual translation of this extended model to SCADE, the authors are able to utilize the SCADE Design Verifier to formally verify properties embedded into the systems. A method to translate Simulink models to SIGNAL, a synchronous programming language very closely related to Lustre, has been presented in Messaoud [2014]. It is heavily influenced by Caspi et al. [2003] and utilizes the same translation procedure.

A different approach has been taken by works such as Miller et al. [2005], Herber et al. [2013], and Reicherdt [2015]. In Herber et al. [2013], the authors translate a discrete subset of Simulink models into the input language for the *satisfiability modulo theory* (SMT) solver UCLID [Bryant 2004]. In contrast to the model checker built into SCADE, SMT solvers, such as UCLID, are able to leverage an, e.g., arithmetical, theory on the model state to drastically decrease the size of the possibly infinite state space of the problem under analysis. In Herber et al. [2013], the authors present (1) transformation rules to translate a subset of Simulink to UCLID, and (2) a method to verify properties on the resulting model using *bounded* model checking and *k-inductive invariant checking* [De Moura et al. 2003]. Their translation process automatically generates a number of verification conditions, such as to detect variable overflow and division by zero, as well as the concrete and symbolic models required for bounded model checking and k-inductive invariant checking, respectively. Using their technique, they were able to utilize the bit arithmetic theory built into UCLID to automatically detect an overflow due to a faulty implementation of a case study from the automotive domain. In a similar approach [Reicherdt 2015], the author presents a translation of discrete time MATLAB Simulink models to *Boogie* [Barnett et al. 2004], the input language for the SMT solver *Z3* [De Moura and Bjørner 2008]. In a number of case studies from the automotive industry, the system developed by the author is able to automatically identify common run-time errors in the model design using bounded model checking and k-inductive invariant checking.

In addition to works presented above, there are techniques to translate control system models developed using MATLAB Simulink to the model checker NuSMV [Cimatti et al. 1999, 2002], such as Miller et al. [2005] and Meenakshi et al. [2006]; to perform verification based on contracts, such as presented in Boström [2011], Wiik and Boström

[2014], and Liebrenz et al. [2018]; and to HybridCSP [Liu et al. 2013], an extension to the process calculus of *communicating sequential processes* [Hoare 1985] to model the semantics of hybrid automata, in Zou et al. [2015].

Research in the area of formal analysis and verification techniques for our second example language, Modelica, is scarce. In Lundvall et al. [2004], the authors present ideas on a translation of a subset of Modelica models to the input language for the hybrid model checker *HyTech* [Henzinger et al. 1997]. However, a planned implementation seems to have been abandoned and, thus, no evaluation was presented. To the best of our knowledge, no further approaches to translate Modelica models into a formally verifiable representation exist. Works such as Klenk et al. [2014] and Otter et al. [2015] present methodologies to embed formal requirements into Modelica models. However, similar to Lundvall et al. [2004], neither an implementation nor an evaluation was included in their publications.

3.4.2 Approaches Built into Modeling Environments

A number of model analysis and verification techniques are integrated directly into MATLAB Simulink by way of the *Simulink Design Verifier* (SLDV) [The MathWorks 2017c]. It is based on the model checker presented in Andersson et al. [2002] and offers the possibility to automatically generate test cases and detect design errors, such as integer overflows or division by zero. As it is integrated into Simulink and verification properties are added as specialized blocks directly into the model, it can be utilized as part of a model-based workflow without the need for intermediate representations or external tools. Its scalability, however, is limited as it utilizes a model checking-based technique to analyze the model as a whole. Works such as Leitner [2008], Reicherdt [2015], and Nellen et al. [2018] have discussed the capabilities of the SLDV and discovered semantic inconsistencies and worse performance compared to similar model-checking based methods.

3.4.3 Summary

The common factor in all the methodologies mentioned above is their translation of the signal-flow-oriented control system model to a formally verifiable representation *as a whole*. This has a number of significant consequences: (1) The translation abstracts from the original signal-flow-oriented semantics. Specifics of the original system are therefore lost, such as the physical timing of models, and the timing of models can therefore not

be precisely analyzed. (2) The translations pose strong restrictions on the design of the source model and are therefore often not directly applicable to industrial case studies, such as discovered by Walde and Luckner [2015]. And finally, (3) the verification effort required increases exponentially with the model size. While the presented methods are able to verify properties, e.g., the absence of overflows, on the model, their analyses reach run-times of up to 12 h for models consisting of 264 blocks, such as in the case of Reicherdt [2015]. Similarly, the authors of Joshi and Heimdahl [2005] conclude their work with a remark on the highly doubtful scalability of their approach to translate MATLAB Simulink models to Lustre.

Most importantly, however, it is not possible to directly apply the presented methods to analyze time-dependent information flow in signal-flow-oriented control system models. For methodologies such as Herber et al. [2013], Reicherdt [2015], and Liebrenz et al. [2018], this is due to the absence of a timing model in the generated formal representation. For other approaches, such as Joshi and Heimdahl [2005], Miller et al. [2005], and Zou et al. [2015], the reason lies in the limited expressiveness of the verification properties. While, e.g., CTL allows for the expression of properties in relation to each other along paths in the model checking process, the operators, such as `next`, `globally`, and `until`, do not relate to the physical simulation time of the original model.

3.5 Formal Analyses of State-Machine-Based Models

Although the *development* of a formal analysis technique for state-machine-based controllers implemented using MATLAB Stateflow is not part of the main focus of this thesis, we briefly introduce research works related to this task. Only few authors have addressed the problem of formalizing the complete behavior of Stateflow automata. In Hamon and Rushby [2004] and Hamon [2005], the authors have presented operational and denotational semantics for a subset of Stateflow. While they succeed in representing a wide range of the Stateflow functionality, they neither consider the timing nor the connection of the automaton with surrounding Simulink models.

A similar approach has been taken by the authors of Chen et al. [2012]. In their work, they present a translation of a subset of Stateflow modeling features to *CSP#* [Sun et al. 2009], an input language to the model checker *Process Analysis Toolkit* (PAT) [Liu et al. 2008]. Using their automatic translation and subsequent model checking, they have

been able to identify modeling errors in case studies provided as part of Stateflow, which include complex modeling features such as *history junctions* and transitions between levels in the state hierarchy.

In Scaife et al. [2004], the authors present a methodology to convert a subset of Stateflow constructs into Lustre. However, they pose strong restrictions on the use of modeling features such as events, inter-level transitions and, more fundamentally, temporal logic conditions.

Due to the similarities between Stateflow and Statecharts, we also briefly discuss previous formalization efforts for such models, most notably Harel [1987]. However, these similarities are merely superficial, as the underlying solver for Stateflow automata works in a purely sequential fashion, and their semantic differences make an elevation of the methodology presented in this work infeasible. In contrast, the authors of Jiang et al. [2016] and Yang et al. [2016] present a method for an automatic translation of Stateflow automata to UPPAAL timed automata, which captures both the functionality and the precise timing of Stateflow and enables automatic verification via model checking. This translation does not pose any restrictions on a given Stateflow model and accurately captures its semantics. We utilize this in our method and provide a discussion in Chapter 6.

3.6 Summary

As we have discussed in this chapter, there exists a broad spectrum of research related to the analysis of information flow through embedded control system models. Due to their importance and widespread use, two languages from the domain of embedded control system models, MATLAB Simulink/Stateflow and Modelica, have been the subject of numerous research endeavors with the goal of developing analysis and verification techniques on a formal basis. First, we have presented methods to analyze information flow in traditional, text-based sequential programming languages in Section 3.1. While these techniques have matured to be applicable for a wide range of domains and languages, they are unable to cope with the semantics of signal-flow-oriented modeling languages, i.e., their timing and inherent concurrency. They are therefore unable to be applied to these models.

In Section 3.2, we have discussed methods to analyze information flow in synchronous programming languages, which bear a semantic resemblance to signal-flow-oriented control system models. The methods presented, however, are not applicable to the

domain of control system models as they do not consider the timing behavior of models. In addition to that, they are too coarse-grained for a precise information flow analysis due to their inability to analyze the control behavior of models.

With the techniques discussed in Section 3.3, we have presented the research works whose goals are most closely aligned to this thesis. Most notable is the work by Whalen et al. [2010], in which the authors present an information flow analysis for signal-flow-oriented control system models. Using their method, they are able to track information flow through signal-flow-oriented control system models. However, as their solution is based on a translation of MATLAB Simulink models to the formally verifiable synchronous language Lustre, the complexity of their model-checking-based methodology makes its utilization impractical for industrially-sized models. In addition to that, their information flow theorem neither captures the precise timing behavior of signals through a model nor the behavior of embedded state-machine-based controllers. Consequently, the presented approach is inapplicable to the problem statement of this thesis.

In Section 3.4, we have discussed a number of approaches based on the idea of translating complete control system models to a formally verifiable representation with the goal of subsequently performing model checking to verify properties on the translated model. A unifying characteristic of these methodologies is their reliance on a translation of the model *as a whole* to a formally verifiable representation. Due to the state explosion problem, this casts doubts on their applicability to models of industrially relevant size. Additionally, during the translation process, semantic characteristics of the source language are lost, such as the timing behavior of the model. As we have shown in Chapter 1, a precise information flow analysis has to take the precise timing of the source language into account, which makes the methodologies presented in Section 3.4 inapplicable to the problem stated by this thesis.

While it is not part of the main focus to develop a formalization and verification technique for state-machine-based model components, we have briefly discussed a number of such methodologies to in Section 3.5. Specifically, we have discussed the methodology presented by Jiang et al. [2016] and Yang et al. [2016], which we use in this thesis.

In the following chapter, we present an overview of our approach to analyze information flow in time-discrete embedded control system models.

4 Information Flow Analysis Approach

As embedded systems are often employed in safety-critical domains, they have to uphold strict requirements in terms of both *safety* and *security*. To manage the strongly increasing complexity of these safety-critical control systems in every application domain [Zurawski 2009; Bello et al. 2019], manufacturers have shifted towards model-driven development techniques [Chalé et al. 2011], which focus on implementation of functionality and various levels of abstraction from implementation details. While these systems are successful in managing the complexity in the *development* of large, interconnected systems [Hutchinson et al. 2014; Rodrigues da Silva 2015], *analysis* and *verification* techniques for model-driven development systems and languages still have to reach a similar level of maturity as those for text-based imperative programming languages traditionally used in the development of safety-critical embedded systems.

One such technique, the analysis of information flow through these *embedded control system models*, has the potential to identify possible violations of both safety requirements and security policies by analyzing where and under which conditions information travels through a program. When developing an information flow analysis technique for model-driven development procedures, their specific semantics, which inherently differs from those of imperative languages, must be considered front and center. This holds especially true for the two most widely-used languages for model-driven development of safety-critical embedded control systems: MATLAB Simulink/Stateflow and Modelica. Both languages, based on the semantics of signal-flow graphs, heavily rely on concurrency and a complex notion of timing. Additionally, they make use of inherently differing modeling styles by incorporating signal-flow-based as well as state-machine-based components. While the former are utilized to model the modification of incoming signal data, the latter control the flow of data through the program according to rules based on the same or separate incoming signals. Due to the possibility to include multiple program-

ming paradigms in the same modeling language, these *combined* models have found widespread adoption in the design of automotive and aerospace control systems [Wang and Yuko 2010; Nellen et al. 2018].

This thesis presents a novel procedure enabling a sound information flow analysis of embedded discrete control system models, which integrates the inherent timing behavior of such models as well as the combined analysis of systems comprised of components of fundamentally different modeling styles, i.e., signal-flow-oriented and state-machine-based components.

Our information flow analysis methodology is based on the main idea to extract only that information from an existing model which is required to analyze information flow through the model in respect to both its timing and functionality. Only this information, i.e., the conditions under which paths are executed as well as when and how these conditions are triggered, is retained during the analysis. All aspects of a model that do not influence the information flow between elements of interest are discarded to increase analysis performance. Further, to enable the analysis of models combining signal-flow-oriented as well as state-machine-based programming paradigms, our methodology is based on the idea of first translating a state-machine-based controller into a formally verifiable representation and, second, using the well-established technique of model checking to observe the controller to identify precisely the behavior that leads to the execution of information flow paths under analysis.

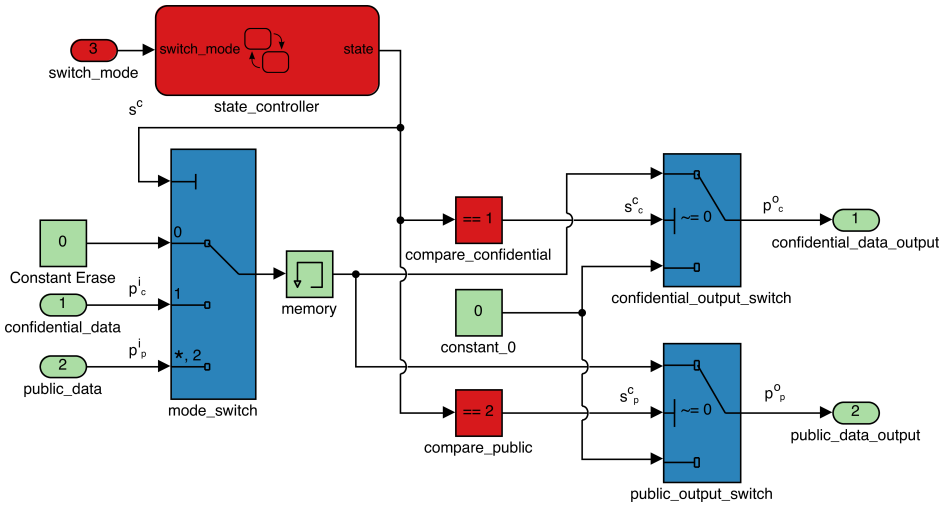
With this process, we are able to safely rule out the existence of information flow on specific paths through a model, which enables us to reason about non-interference between model parts and the compliance with security policies.

In this chapter we present an overview of our procedure to analyze information flow in time-discrete embedded control system models. First, we present a model consisting of both signal-flow-oriented and state-machine-based components as an introductory example. Using this example, we demonstrate the model structure we base our process on and terms we use in this thesis. Subsequently, we discuss assumptions that we require models to fulfill in order to be analyzable by our methodology and present every step of our system as well as the automated analysis framework we have developed.

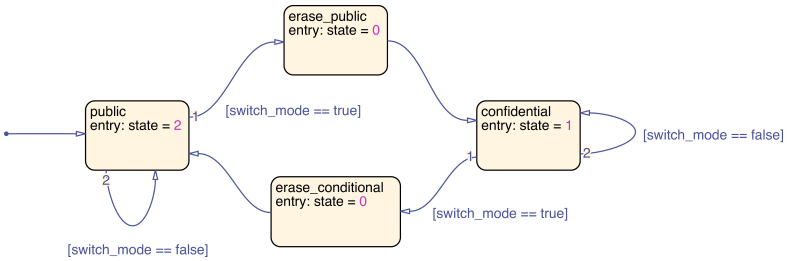
4.1 Motivating Example

To illustrate our approach and the structure of embedded control system models typically developed using signal-flow-oriented model-driven development languages, we use a shared communication infrastructure as a introductory example. Such systems, commonly found in the design of automotive software [ISO 1993, 2015], combine complex timing as well as control behavior. Figure 4.1 shows the corresponding model as a MATLAB Simulink/Stateflow implementation. It uses a time-dependent `Memory` block as an internal buffer and `Switch` blocks to route the incoming and outgoing data according to their source and target, respectively. In this example, information of two different security levels is fed into the shared buffer via the public input p^{i_p} and confidential input p^{i_c}, respectively. According to the value of the signal s^c, confidential or public information is saved in the buffer and passed to the corresponding output. Although confidential and public data entering the model share the same memory block as buffer, the routing conditions implemented using the `Compare`-type blocks are intended to ensure that confidential input data can never flow to the public output. To this end, the operation mode set via the signal s^c defines which input should be routed to the output and to which output the content of the buffer element is routed to.

An obvious security requirement for this model states that information which entered through the confidential input must never reach the public output. This has to be ensured by the design of the controller. Consider the example situation shown in Figure 4.2a. There, the value of the signal s^c is shown in green and the values for the public input p^{i_p} and the confidential input p^{i_p} are shown in red and blue, respectively. Whenever the signal s^c has the value 1, the buffer content is routed through the confidential output, and to the public output, whenever s^c has the value 2. For three simulation cycles, the value 5 is fed into the buffer through the public input, after which the value 3 is fed into the buffer through the confidential input. Figure 4.2b plots the operation mode set by the signal s^c and the behavior of the confidential output p^{o_c} in green and the public output p^{o_p} in red. There, a violation of the security requirement becomes apparent. When switching the operation mode from confidential to public, the current buffer contents, i.e., confidential data, is leaked through the public output.



(a) Combined Model

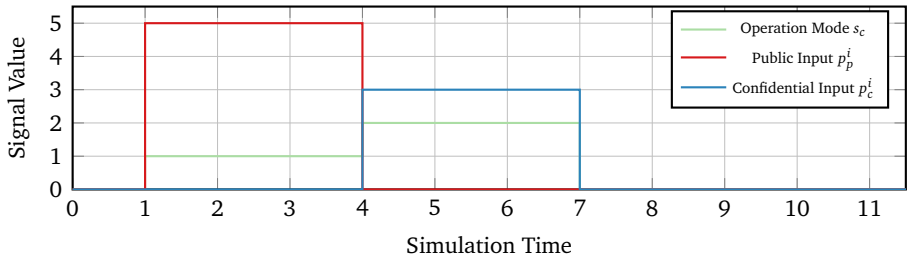


(b) The state-machine-based controller `state_controller`

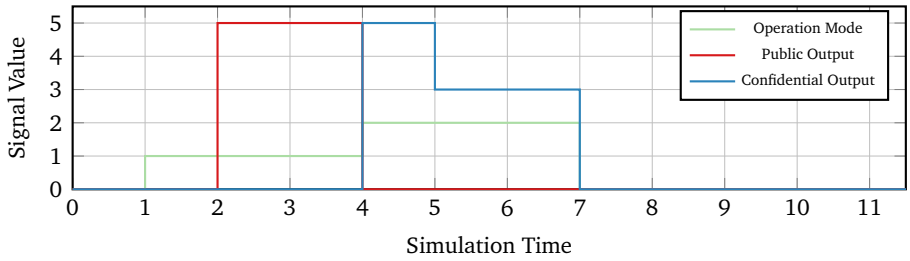
Figure 4.1: Motivating example implemented as combined model

To identify such security violations caused by the complex control and timing behavior of the model, an information flow analysis for signal-flow-oriented model components is required, which takes into account both the precise data and timing behavior of data flowing through the model. We present our system to identify information flow through such signal-flow-oriented models in Section 4.4.1 and, in more detail, in Chapter 5.

Additionally, our running example shown in Figure 4.1 utilizes a state-machine-based controller `state_controller`, implemented in Stateflow and shown in Figure 4.1b, to control the operation mode, i.e., execution of the switches, by setting the signal s^c. The controller switches the operation mode depending on the value of the input fed through the `Boolean` input port p^{i_3}. If this input signal is set to *true* for a single simulation cycle, the operation mode is switched. As the Stateflow controller is responsible for the modification of the operation mode through the signal s^c, an analysis of its state-machine-based semantics in conjunction with the signal-flow-based model components is



(a) Input signal timing



(b) Operation mode and output signal timing

Figure 4.2: Signal output and timing in our motivating example

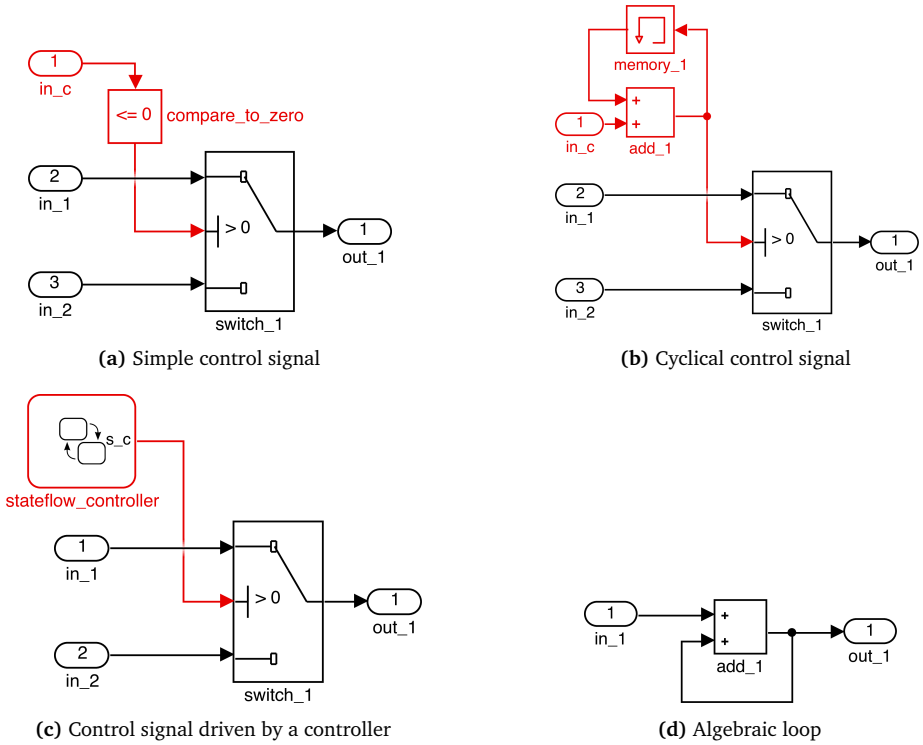
necessary to evaluate whether such a *combined* model suffers from the discussed security policy violation. In Chapter 6, we revisit the example presented in Figure 4.1 and provide a detailed discussion of our methodology to analyze information flow through combined signal-flow-oriented and state-machine-based models.

Note that, as presented in Chapter 2, the discrete embedded control system models our method analyzes, are not *executed* directly, but interpreted and simulated by a simulation engine specific to the modeling language. For our motivating example, we have haven chosen a *fixed-step* solver with a simulation step size of 1.

4.2 Model Structure

The control system model[1] displayed in Figure 4.1 represents the general structure of discrete embedded control system models that we base our method on. When analyzing the information flow through such models, we consider the flow between arbitrary model elements, such as, e.g., input and output blocks. These flows through the dynamic signal-flow-oriented components, shown in green, are controlled by sets of routing blocks.

[1]Note that we use the term *control system model* to denote the complete model under analysis, consisting of both signal-flow-oriented and state-machine-based components.



(a) Simple control signal

(b) Cyclical control signal

(c) Control signal driven by a controller

(d) Algebraic loop

Figure 4.3: Control signal styles (control signal paths shown in blue)

Additionally, time-dependent model elements, such as `Memory` or `UnitDelay` blocks, hold information along these *data paths* and release them according to their specifications and properties set by the model developer.

The information flow through such models is controlled by *control signals* responsible for the execution of the switches, shown in blue. These signals and their *control signal paths* are shown in red. In discrete embedded control system models, these control signal paths can be structured in three possible ways: (1) An embedded state-machine-based controller[2] emits the control signals, as shown in Figures 4.1 and 4.3c. (2) The control-flow elements are set without the use of any controller and are acyclical, as shown in Figure 4.3a. (3) The control-flow elements are set without the use of any controller but contain cyclical components, as shown in Figure 4.3b. (4) The output of a state–machine-based controller is routed through a cyclical control signal. (5) Control-flow structures are nested, i.e., control-flow elements contain routing blocks themselves.

[2]We use the term *controller* to denote the state-machine-based components of the control system models we analyze.

Additionally, the controller can receive input signals from the surrounding signal-flow-based components.

4.3 Assumptions

To apply our system to analyze the flow of information, a given discrete embedded control system model model has to fulfill the following assumptions:

Control Paths Our information flow analysis method supports unnested non-cyclical and cyclical control paths, as well as control paths driven by state-machine-based controllers. For non-cyclical control paths, we support only data modifications using blocks implementing unary functions. For cyclical control paths, we support paths with cyclic dependencies that can be solved using the Mathematica *computer algebra system* (CAS) [Wolfram Research 2018].

Solver Semantics In the model, every block is set to use the same sample time and a *time-discrete, fixed-step* solver is used. Additionally, the model does not contain algebraic loops or loop subsystems.

Language Support As our procedure targets signal-flow-oriented modeling languages, we demonstrate its applicability to two widely-used representatives from this language family, MATLAB Simulink/Stateflow and Modelica. When targeting Modelica models as the source language, we support the analysis of blocks from the discrete controller library as well as the standard library [The Modelica Association 2019a]. When targeting MATLAB Simulink/Stateflow models, we support blocks from the *Discrete*, *Logic and Bit Operations*, *Math Operations*, *Ports & Subsystems*, *Signal Routing*, *Sources* and *Sinks* block sets [The MathWorks 2018a].

As we target safety-critical embedded software systems stemming from a model-centric development style, which typically do not contain any continuous components, our restriction to purely discretely-timed models is acceptable. The restriction to a uniform sample time is chosen for simplicity of presentation and can be relaxed as future work. We discuss this in Section 8.2. Finally, our requirement to prohibit algebraic loops is acceptable as our focus on model-centric development methodologies prohibits the use of such algebraic loops, as shown in Figure 4.3d, in the control system models. This is due to the fact that such untimed loops, i.e., loops that do not contain any time-dependent model elements and execute multiple iterations in a single simulation time step, cannot

be used for code generation [The MathWorks 2017a]. To support the information flow analysis through algebraic loops, a fixed-point analysis would be required to identify a stable loop state for a given simulation time step.

As our scheme analyzes the control flow of discrete embedded control system models, we do not impose any restrictions on the data paths, i.e., complex modeling elements such as integrators and transfer functions, together with arbitrary feedback loops, may be used on the data path.

On the control paths, i.e., when analyzing the precise conditions under which information flow paths are executed, our system makes a number of fundamental assumptions. From the control path variants presented in Figure 4.3, we fully support control paths containing state-machine-based controllers and do not make any assumptions on the design of such controllers. The case studies from the automotive domain, which we present in Chapter 7, do not rely on complex control logic implemented using signal-flow-oriented modeling but utilize only simple signal modifications on unnested control paths and make heavy use of state-machine-based controllers to control the execution of paths through the model. Our restriction to unary functions on unnested non-cyclical control paths is therefore, while fundamental, acceptable. We discuss possible solutions to relax this requirement in Section 8.2. Finally, in case of cyclical control paths, we only support the resolution of those cyclic dependencies that the employed CAS, Mathematica, is able to solve, such as linear difference equations [Wolfram Research 2008]. We consider a precise examination of the limitations using more complex case studies as future work and provide a more in-depth discussion in Section 8.2. As we, in case of cyclical control signals, assume arbitrary inputs to the control path, a state-machine-based controller driving a cyclical control signal is supported by our procedure.

All discrete embedded control-system models based on signal-flow-based semantics that satisfy these assumptions can be safely analyzed using our methodology.

4.4 Information Flow Analysis of Discrete Embedded Control System Models

An overview of our proposed method is shown in Figure 4.4. Shown on the left, we also depict a more abstract view of the structure of embedded discrete control system models which we take as a basis for our analysis.

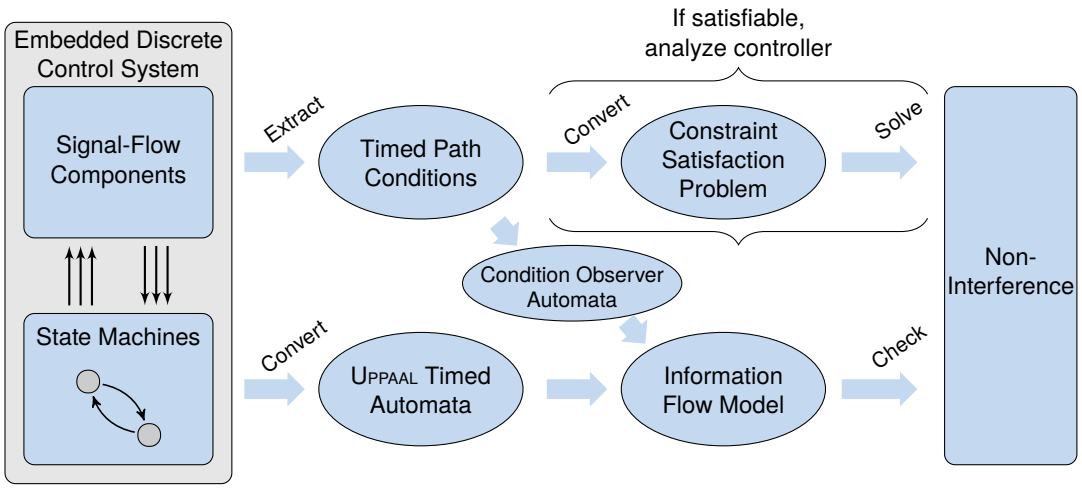


Figure 4.4: Proposed Solution

4.4.1 Information Flow Analysis of Signal-Flow-Oriented Models

To enable a fine-grained analysis of the flow of information through embedded control system models, our system adapts the concept of path conditions to the semantics of signal-flow-based models. These *timed path conditions* capture the conditions necessary for paths to be executed as well as the precise timing of the control behavior along the paths. The main idea behind this first major step of our procedure is that by only capturing the information necessary to verify the absence of information flow, i.e., control flow conditions and timing information, the timed path conditions form an ideal compromise between analysis precision and efficiency. At the same time, they do not sacrifice soundness.

Our scheme first constructs timed path conditions from the dynamic signal-flow-oriented components of a given control system model, i.e., between the components shown in blue and green in our motivating example. The main challenge we face is that all dependencies between a given pair of model elements in a given design must be considered. Thereby, dependencies might be indirectly introduced via control flow, or delayed (such as demonstrated by the `Memory` block in our motivating example), which introduces dependencies between signals from different points in time. Our process starts with a static over-approximation of all potential dependencies on a path between a timed input signal and a timed output signal and collects all control flow conditions. Then, for each path, we compute a set of constraints on all input signals by performing a backward propagation of control flow conditions, which also takes timing

dependencies into account. The result of this step of our analysis is a precise description of the timed dependencies between input and output signals of the signal-flow-oriented components of the control system, represented by timed path conditions that solely depend on model-wide input variables.

The output of the first step of our method can be utilized to reason about information flow relations between arbitrary model elements in the signal-flow-oriented components of the system. If, for example, the timed path conditions between two blocks of interest do not overlap, information flow between these blocks can safely be ruled out, i.e., we are able to prove non-interference between them. In order to determine the relationship between such model elements of interest, we formulate CSPs for each pair of elements. We achieve this by gathering the timed path conditions guarding the execution of the paths between the pair of elements and translating them into constraint sets. Finally, we employ a *constraint solver* to identify whether a combination of model-wide input variables exists such that the set of constraints for the particular path under analysis is satisfiable. If not, then non-interference between the model elements under analysis is proven in the face of complex timing as well as control behavior, i.e., information flow on the path under analysis is impossible.

If the set of constraints for a path under analysis is satisfiable, then, given a purely signal-flow-oriented model, information flow is possible. As we show in Chapter 5, the leak of confidential data present in the design of the signal-flow-oriented components of our motivating example is shown by the satisfiability of the CSP constructed from the timed path conditions of the path between p^{i_c} and p^{o_p}.

4.4.2 Information Flow Analysis of Heterogeneous Models

The first step of our analysis of a given embedded control system model captures the complete behavior of the signal-flow-based components of the model. For models, in which the execution of paths is controlled by state-machine-based components, such as commonly found in industrial applications, however, the result of the first step, i.e., the solution of the CSP constructed using the timed path conditions, forms an over-approximation of the possible behavior of the model. This is demonstrated in our motivating example, shown in Figure 4.1. There, the extracted timed path conditions form an over-approximation, as the controller limits the possible sequences of the control signal s^c. Thus, the sequence of operation mode switches resulting in the security policy

violation described in Section 4.1 may be spurious. Extending the analysis to include the state-machine-based components controlling the execution of each information flow path therefore has the possibility to yield more precise results.

We accomplish this by using a *combined* analysis of the state-machine-based controllers and the extracted sets of timed path conditions. A central characteristics of our analysis is a formal view on the execution behavior of the controllers as well as on the interaction between the controllers and execution of the paths through the signal-flow-oriented model components in terms of timing as well as data. To achieve this, we first translate the controller into a timed automata representation in order to get access to the verification capabilities of a timed model checker. Then, we generate *condition observer automata* from the sets of timed path conditions, i.e., timed automata representations of the sets of timed path conditions that we combine with the translated controller. The design of the condition observer automata ensures that a defined state is reached if the translated controller produces control signals that match the timed path conditions in data as well as in timing.

In the final step, we automatically generate model checking properties for these combined automata. In order to prove non-interference on a path under analysis for a *complete* model, we utilize a timed model checker to verify whether it is possible to reach the defined final state of the timed path condition observer automaton. If this state is reachable, we have proven that the controller produces a sequence of control signals that satisfies the timed path conditions. If, however, the corresponding property is unsatisfiable, we have proven non-interference on the path under analysis. Our system to combine model checking and timed path conditions therefore reduces the identification of information flow through combined embedded control system models to a *reachability* check on the controller automaton.

4.4.3 Automation of our Information Flow Analysis Approach

To reach a high degree of automation, we have implemented our procedure as a fully automated analysis solution for time-discrete control system models. Starting from a system model and source and sink of information flow to be analyzed, our implementation automatically performs all steps described above and presents its results both machine-readable as well as graphically directly in the model. This enables the integration of our procedure in a model-based workflow.

The current instantiation of our *framework* supports information flow analysis of embedded control system models implemented using MATLAB Simulink/Stateflow or Modelica. As a timed model checker to verify properties on the state-machine-based components, we utilize the UPPAAL tool environment [Bengtsson et al. 1996]. As our framework utilizes a language-independent intermediate language, the integration of additional source languages is possible via corresponding translation front-ends. Further, our method and the developed framework allow for the utilization of timed path conditions as a starting point for additional analyses, such as compositional verification or generation of efficient test cases.

4.5 Summary

In summary, we present an analysis technique that is able to prove information flow properties for embedded discrete control system models consisting of signal-flow-oriented components and state-machine-based controllers. Our approach is *highly efficient* due to the separate analysis of the different underlying semantics, and *highly automated* as our framework enables the analysis of system models without user interaction.

In the following chapters, we present our process in greater detail. First, we introduce our information flow analysis of the signal-flow-oriented components of embedded control system models in Chapter 5. In Chapter 6, we present our process to identify information flow in embedded control system models consisting of both signal-flow-oriented and state-machine-based components.

5 Information Flow Analysis of Signal-Flow-Oriented Models

The goal of this thesis is to provide a methodology to analyze information flow through discrete embedded control system models. To achieve this, our technique takes into consideration the specific semantics of signal-flow-oriented model components as well as their connection to state-machine-based controllers. We achieve this by (1) calculating an over-approximation of the time-dependent control flow conditions through the signal-flow-oriented components, and (2) performing a satisfiability check of these control flow conditions on a formally verifiable representation of the state-machine-based components. In this chapter, we present the first step of our method, the analysis of information flow through the signal-flow-oriented components of embedded control system models.

5.1 Approach

In our approach to analyze the flow of information through signal-flow-oriented models, we perform multiple analysis steps starting directly on the source models.

Intermediate Model Representation & Path Identification. The first step of our analysis is a translation of the source model into an language-independent intermediate representation called *Java Intermediate Representation* (JIR). This translation and representation, originally developed in Reicherdt [2015] for Simulink models and extended by us to support additional source languages, encapsulates all syntactical features of the source models and allows for a detailed analysis of the structure and the behavior of the source model. On this representation, we then identify all paths between a selected pair of model elements.

Identifying Timing Dependencies. Using the extracted set of paths as an input, the next step of our methodology analyzes each path to identify whether information flow along the path is conditional or unconditional. If, on any path, unconditional flow is identified, information flow is possible and non-interference cannot be proven. In this case, our analysis returns this information and terminates. If no path allowing unconditional flow is detected, our analysis continues and returns a set of routing blocks for each path between the selected model elements of interest.

Extracting Local Timed Path Conditions. Subsequently, our scheme identifies the timing relation between the control flow elements in each set by analyzing whether stateful model elements are present on the path. If so, it collects the identified control flow elements into *time slices*, which concisely describe the timing relation between elements on a single path in a discretely-timed model. Subsequently, we extract the conditions for information flow for each control flow element on the path and collect them into their corresponding *time slice*. The information extracted in this step relates all *local* control flow conditions for each path, which, however, does not allow us to draw a conclusion about the behavior of the path as local signals in the signal-flow-oriented models we aim to analyze do not directly correspond to each other. In the next step, we therefore aim to relate local control flow conditions to each other to be able to draw conclusions about the global behavior of the path, i.e., whether non-interference can be proven or information flow is possible.

Evaluation of Control Signals. In this step, we first perform a backwards path calculation starting from the control signal input of each control flow element on the paths under analysis and ending whenever a global input block is reached. Here, two situations can occur: (1) The extracted subgraph is acyclic, i.e., does not contain any feedback loops. In this situation, we perform a backward propagation of the control signal modification from the routing block to the global input block in order to relate the local control signal with a global input. (2) The subgraph describing the path contains a cycle. In order to support this case, our methodology extracts the mathematical formula describing the cycle and aims to find an analytical solution to the extracted difference equation using a CAS. Using the determined solution, our approach can then relate the global input signals on the path to the local control flow conditions on the path.

Translating and Solving Path Conditions. Finally, to draw a conclusion whether information between the pair of model elements is possible, we formulate a CSP from the global control flow conditions of each path and solve it using a constraint solver. If, for a path under analysis, the resulting CSP has a valid solution, information flow is possible. Similarly, if a solution does not exist, non-interference is proven.

In the remainder of this chapter, we present each step in detail.

5.2 Intermediate Model Representation

The first step of our scheme translates the signal-flow-oriented components of the input model into a *language-independent intermediate representation* on which we perform our analyses. This intermediate format, which we adapt from Reicherdt [2015], is a representation of the execution behavior of the source model that enables easier parsing of the model structure.

To demonstrate the source language independence of our system, our contribution in this first step is the development of an additional front-end to translate our second example modeling language from the domain of signal-flow-oriented modeling, Modelica, into the JIR. The result of this first preparatory step of our analysis is a model representation that is semantically equivalent to the source model but allows for easier extraction of structural and semantical features of the model. Additionally, this step enables a unified analysis mechanism for multiple source languages.

5.3 Finding Paths of Interest

The next step of our analysis, of which an overview is shown in Figure 5.1, is the identification of possible paths of information flow through the signal-flow-oriented components of the model. A path, defined in Definition 5.1, describes a sequence of blocks over which information *may* flow.

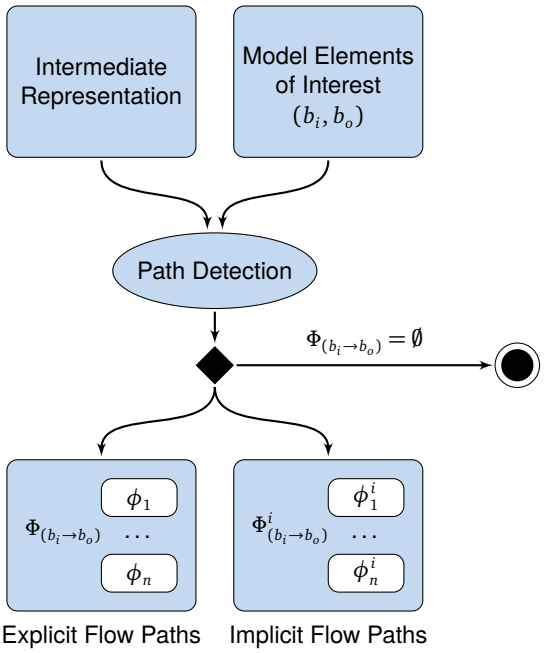


Figure 5.1: Finding paths of interest between b_i and b_o

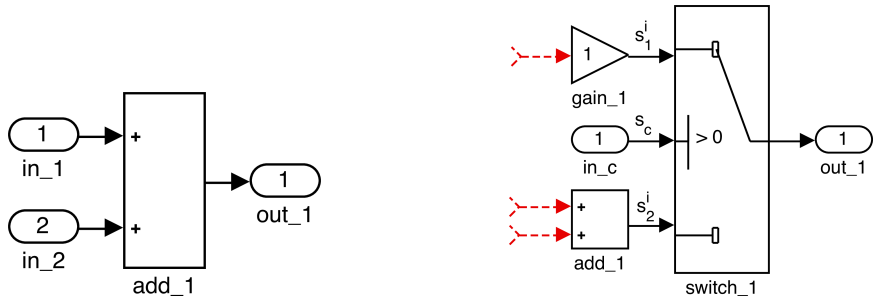
Definition 5.1: Path

A **path** $\phi = \langle b_0, \ldots, b_n \rangle$ through a signal-flow-oriented model is defined as a sequence of blocks between two model elements of interest b_0 and b_n, such that a signal connection between b_i and b_{i+1}, with $0 \leq i < n$, exists.

A **set of paths** between the model elements b_0 and b_n is denoted by:

$$\Phi_{b_o \to b_n} = \{\phi_0, \ldots, \phi_l\}.$$

The input to this first step is a pair of model elements $(b_i, b_o) \,|\, b_i, b_o \in B$ between which information flow is to be analyzed. To analyze the information flow between both elements, this first step identifies *all* paths between (b_i, b_o) present in the model by traversing the model. Our path detection starts at the given model element b_o and implements a *depth-first* recursive search while marking already visited blocks and follows explicit signal as well as control flow connections between model elements. Thus, it detects *explicit* as well as *implicit* information flow types.



(a) Explicit flow from $b_{\text{in\_1}} \rightarrow b_{\text{out\_1}}$ **(b)** Implicit flow from $b_{\text{in\_c}} \rightarrow b_{\text{out\_1}}$

Figure 5.2: Information flow relations detected using our path detection algorithm

Examples for both situations can be seen in Figure 5.2. For an explicit flow, direct signal connections between blocks are considered, such as an arbitrary block driving an input of an Add block, shown in Figure 5.2a. Additionally, implicit flows through control structures, such as seen in Figure 5.2b, are considered. There, our path detection algorithm performs its depth-first search through the explicit data connections on the input signals s_1^i and s_2^i as well as through the control signal s^c.

We return the set we denote $\Phi_{(b_i \rightarrow b_o)}$, which contains all paths between b_i to b_o, and our methodology continues with its next step using $\Phi_{(b_i \rightarrow b_o)}$ as an input.

Application to Motivating Example. Revisiting our motivating example presented in Section 4.1, we apply this step of our technique to identify all paths between the confidential input p^{i_c} and the public output p^{o_p}. The analysis yields a single path ϕ_1 shown below:

$$\Phi_{(p^{i_c} \rightarrow p^{o_p})} = \{\phi_1\}$$
$$\phi_1 = \langle p^{i_c}, b_{\text{mode\_switch}}, b_{\text{memory}}, b_{\text{public\_out\_switch}}, p^{o_p} \rangle$$

5.4 Identifying Timing Dependencies

In this step of our methodology, illustrated in Figure 5.3, we introduce our analysis of the precise timing behavior of each path of possible information flow between the model elements of interest. This makes it possible to analyze the flow of information through a model in the face of complex timing and control behavior.

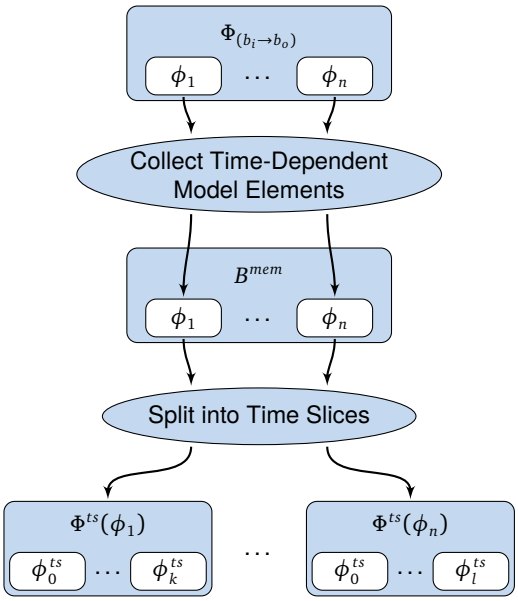
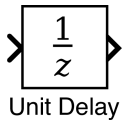
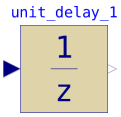


Figure 5.3: Identifying timing dependencies on the paths in $\Phi_{b_i \to b_o}$

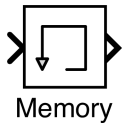
Time-Dependent Model Behavior. Unlike traditional text-based imperative programming languages, the semantics of signal-flow-oriented models contains time-dependent elements. During simulation of a model, these elements contain an internal state which depends on information from the both the current and previous simulation steps. In discretely-timed models, which our method focuses on, these *time-dependent* model elements hold information arriving at their inputs over the course of a fixed amount of simulation steps before releasing it at their respective outputs. Figure 5.4 shows the four block types used in our example languages MATLAB Simulink and Modelica to hold information across simulation steps. Figures 5.4a and 5.4b show `UnitDelay` blocks utilized in both languages, which hold information for a single simulation step in time-continuous as well as time-discrete models. As we focus on time-discrete models utilized in the domain of safety-critical software systems, the semantics of both `UnitDelay` blocks are equal to those of the `Memory` block seen in Figure 5.4c. This block type can only be used in time-discrete models and holds its input value for a single simulation step. Finally, the `Delay` block shown in Figure 5.4d, utilized in MATLAB Simulink, holds its input value for a configurable number of simulation steps. In the example, this value is set to 2. For all presented block types, a parameter can be set which configures the initial value at the output at the first simulation step.



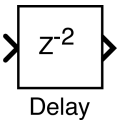
(a) Simulink `UnitDelay` block



(b) Modelica `UnitDelay` block



(c) Simulink `Memory` block



(d) Simulink `Delay` block

Figure 5.4: Time-dependent model elements B^{mem} in MATLAB Simulink and Modelica

To achieve an analysis that takes the time-dependent behavior of signal-flow-oriented models into account, we utilize the set $\Phi_{(b_i \to b_o)}$ built in the previous step and identify timing dependencies along each detected path in the set. Note that we do not express the precise *data* dependencies between b_i to b_o, i.e., we do not perform an analysis on the data behavior between both elements. Rather, through the identification of a path between b_i and b_o, we have ensured a syntactical connection between both elements and, in this step, assume a data dependency. We are, however, interested in the *timing* relation between information leaving b_i and arriving at b_o. In our motivating example, the timing behavior of information held in a time-dependent model element is responsible for the leak of confidential information. The knowledge of the *age* of information at every element through which it passes therefore has to be considered an integral part of an information flow analysis for signal-flow-oriented models. To identify this timing relation, we iterate over each block b on each path $\phi \in \Phi_{(b_i \to b_o)}$ and analyze it regarding time-dependent model elements and their parameters. While iterating over the path and collecting the time-dependent model elements we encounter, we either (1) establish an *untimed* dependency relation between b_i and b_o, if the path ϕ_n does not contain any time-dependent model elements, or (2) establish a *fixed-delay* dependency relation, if the path contains time-dependent model elements.

Untimed Relation. If no time-dependent model element is found on the path, there is no delay between information leaving b_i and arriving at b_o, as no model element holds the information along the path. We express this *untimed* dependency relation as shown in Definition 5.2

Definition 5.2: Untimed Dependency

An **untimed dependency** between two arbitrary model elements b_i and b_o is denoted by

$$b_o \, \mathrm{dep}_0^{\phi} \, b_i.$$

This relation states that information *leaving* block b_i *potentially enters* block b_o in the same simulation step and, additionally, that b_i and b_o are connected via the path ϕ.

Fixed-Delay Relation. If we, while iterating over the path, detect any of the time-dependent model elements, we analyze type and parameters of each block to express the precise timing behavior of data passing through them. Using this information, we can establish a *fixed-delay relation* between the first block of the path and an arbitrary block along the path. We denote this relation as defined in Definition 5.3. There, t describes an arbitrary simulation step and the value of k is determined by reading the type and parameters of the time-dependent model elements between b_i and b_o.

Definition 5.3: Fixed-Delay Dependency

An **fixed-delay dependency** between two arbitrary model elements b_i and b_o is denoted by

$$b_o \, \mathrm{dep}_k^{\phi} \, b_i.$$

This relation states that information *leaving* block b_i in an arbitrary simulation step t *may enter* block b_o in simulation step $t + k$ via the path ϕ.

Time Slices. Using this extracted information about the timing behavior of data along the path, we divide the path in subpaths we denote as *time slices*. A time slice describes the subpath on which the age of information on the path is the same with respect to the simulation step, as defined in Definition 5.4.

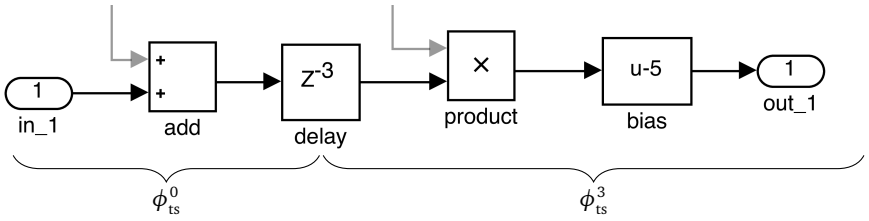


Figure 5.5: Creation of *time slices* from an example path in MATLAB Simulink

Definition 5.4: Time Slice

A **time slice** ϕ_k^{ts} is defined as a subpath of $\phi = \langle b_{\text{in}}, \ldots, b_{\text{out}} \rangle$, such that:

$$\phi_k^{\text{ts}} = \langle b_0, \ldots, b_n \rangle \quad \text{with } b_i \, \text{dep}_k^{\phi} \, b_{\text{in}} \mid \forall b_i \in \phi_k^{\text{ts}}$$

This defines a sequence of those blocks on the path ϕ, on which the fixed-delay relation dep_k^{ϕ} with the first block on ϕ holds. Consequently:

$$\begin{aligned} b_0 &\in B^{\text{I}} \cup B^{\text{mem}} \\ \{b_1, \ldots, b_{n-1}\} &\in B^{\text{b}} \\ b_n &\in B^{\text{b}} \cup B^{\text{O}} \end{aligned}$$

Each time slice has a fixed time delay to the first block of the path under analysis, i.e., for a given path we precisely know the age of information passing through blocks along the path.

Figure 5.5 shows the example of a path with a single `Delay` block and a delay parameter set to 3, from which we create the two time slices ϕ_0^{ts} and ϕ_3^{ts}.

After extracting the set of time slices of each path in $\Phi_{(b_i \to b_o)}$, the result of this step of our scheme is a concise view on the timing dependencies between every block on every path between with b_i and every other arbitrary block on the path. Note that each block can be part of multiple time slices as it can be part of multiple paths between b_i and b_o. As an additional notation, we denote the factor describing the maximum time slice depth over every path in a given model as d_{max}.

Finally, to express the values of signal variables at arbitrary simulation steps, we define *timed variables* as shown in Definition 5.5.

Definition 5.5: Timed Variables
For every signal variable s in the model, we define a sequence of **timed signals** $\langle s_0, s_1, \ldots, s_{d_{\max}} \rangle$. For an arbitrary delay k, the variable s_k describes the value of the signal at simulation step k.

Application to Motivating Example. When iterating over the previously identified path ϕ_1, our algorithm detects a single time-dependent model element and therefore establishes the following time slices and dependency:

$$
\begin{aligned}
& p^{o_p} \quad \mathrm{dep}_1^{\phi_1} \; p^{i_c} \\
& \phi_0^{\mathrm{ts}} = \langle p^{i_c}, b_{\mathrm{mode\_switch}} \rangle \\
& \phi_1^{\mathrm{ts}} = \langle b_{\mathrm{memory}}, b_{\mathrm{public\_output\_switch}}, p^{o_p} \rangle
\end{aligned}
$$

The extracted *fixed delay* dependency illustrates that the information leaving through the output p^{o_p} has entered the system via the input p^{i_c} precisely one simulation step prior.

5.5 Extracting Local Timed Path Conditions

After obtaining the precise timing information of every block on every path in $\Phi_{(b_i \to b_o)}$, in this step of our approach, we extract the *timed* conditions under which the respective path under analysis is executed. Figure 5.6 illustrates this step.

To gather the local control flow conditions necessary for execution of a path, i.e., under which conditions information flows along a specific path, we iterate over every block on every path $\phi \in \Phi_{(b_i \to b_o)}$ and analyze its type. Upon encountering any of the routing blocks presented in Section 2.2.3, we extract its parameters for further analysis. Each routing block evaluates a propositional logic formula set by the model designer, comparing the value of a single control signal s_c with a single value or range of values.

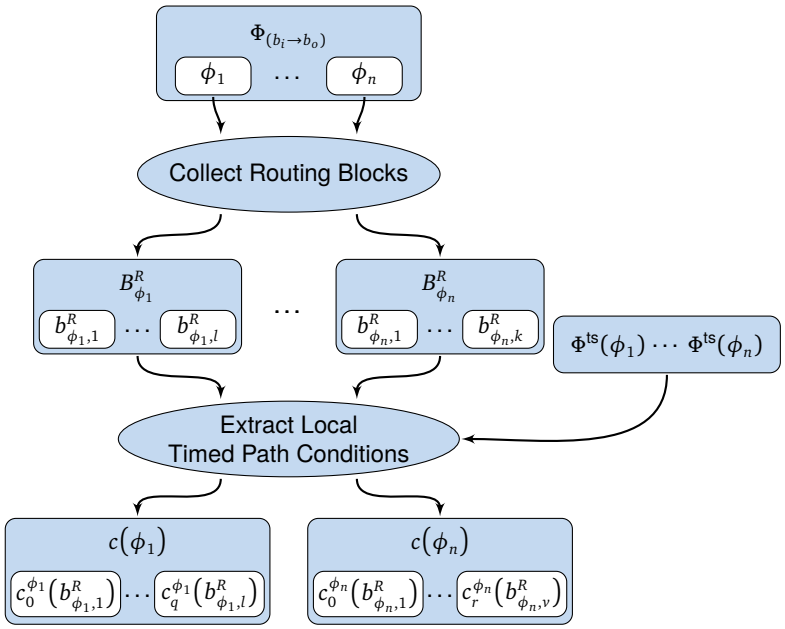


Figure 5.6: Extracting local timed path conditions on the paths in $\Phi_{b_i \to b_o}$

For the routing blocks utilized in MATLAB Simulink models, the possible formulae are:

`Switch` A `Switch` block routes one of two input signals s^{i_1} and s^{i_2} to its output signal s^o depending on the evaluation of a propositional logic formula $f_{\text{Switch}}(s^c)$ that either takes the form $s^c \neq 0$ or $s^c \bowtie v$, where $\bowtie \in \{\geq, >\}, v \in \mathbb{Z}$. The routing follows the definition $f_{\text{Switch}}(s^c) \to (s^o = s^{i_1}) \wedge \neg f_{\text{Switch}}(s^c) \to (s^o = s^{i_2})$.

`MultiPortSwitch` Similar to a multiplexer in hardware design, a `MultiPortSwitch` routes one of a configurable number of input signals $s^{i_1}, \ldots, s^{i_n}$ to a single output signal s^o depending on the value of the control signal s^c. The routing follows the definition $s^c == n \to s^o = s^{i_n}$, i.e., whenever the control signal s^c has the value n, the n^{th} input signal is routed to the output signal s^o.

Equally, for the routing blocks utilized in Modelica:

`Switch` In the Modelica model library, a `Switch` block has a semantics similar to that of the block with the same name in MATLAB Simulink. The control signal s^c, however, is defined as a logical signal of `Boolean`, such that the routing follows the definition $s^c == \text{true} \to (s^o = s^{i_1}) \wedge s^c == \text{false} \to (s^o = s^{i_2})$. Note that the Modelica

language library contains a block `LogicalSwitch`, which utilizes precisely the same semantics as the `Switch` block, but can only be utilized to switch input signals of type `Boolean`.

For every routing block that we encounter on the path under analysis, we extract its respective routing definition and specific condition necessary for execution of the current path. These conditions for information to flow over path ϕ through routing block b take the form shown in Definition 5.6.

Definition 5.6: Path Condition

A **path condition** $c^{\phi}(b)$ describes a necessary condition for information to flow through routing block b along path ϕ.

In our notation, a path condition $c^{\phi}(b)$ is an expression describing a logical comparison operation of a signal with a given constant.[a]

[a] Note that due to the semantics of the routing blocks, this comparison with a constant is sufficient.

Additionally, from the location of each routing block on the path, i.e., from the time slice it is sorted into, we know the delay of the input signal reaching the routing block compared to the first block of the path. Using the information extracted in the previous step, we therefore precisely know *when* the current routing block has to switch in order for information of a certain age to travel along the path we are analyzing. We capture this information in what we have called *timed path conditions*. We build timed path conditions by annotating the condition above with the time slice ϕ_{ts}^{k} the routing block b is sorted into, which takes the form shown Definition 5.7.

Definition 5.7: Timed Path Condition

A **timed path condition** $c_k^{\phi}(b)$ describes a necessary condition for information leaving the first block on the path ϕ at an arbitrary simulation step t to flow through routing block b along path ϕ at simulation step $t+k$.

Finally, the set of all local timed path conditions controlling the execution of the paths in Φ over all detected time slices takes the form shown in Definition 5.8.

Definition 5.8: Timed Path Condition Set

A **timed path condition set** $C(\Phi)$ is defined by:

$$C(\Phi) = \bigvee_{\phi \in \Phi} \bigwedge_{b \in B_\phi^R} c_d^\phi(b), \quad \text{where } B_\phi^R \text{ describes the set of routing blocks on path } \phi$$

This describes the conjunction of all timed path conditions on a single path, disjunctively combined over all paths in the set Φ.

Note that the timed path conditions extracted in this step are *local*, as they refer to the behavior of the local control signal of each routing block. To be able to relate multiple timed path conditions to each other, i.e., to be able to to draw conclusions about the control behavior of a path as a whole, the next step in our technique analyzes each local control signal. We achieve this by propagating local control signals backwards until a global model input is reached. We discuss this next analysis step in the following section.

Application to Motivating Example. On iterating over the detected path ϕ_1, our algorithm identifies the blocks $b_{\text{mode\_switch}}$ and $b_{\text{public\_output\_switch}}$ as routing blocks and proceeds to extract their parameters and timing information corresponding to the time slices they are placed into:

$$c_0^{\phi_1}\left(b_{\text{mode\_switch}}\right) = \left(s_0^c == 1\right)$$
$$c_1^{\phi_1}\left(b_{\text{public\_output\_switch}}\right) = \left(s_1^{c_p} > 0\right)$$

5.6 Evaluation of Control Signals

To be able to draw conclusions about the control behavior of a path, we resolve the *local* timed path conditions extracted in the previous step to *global* ones, i.e., timed path conditions that do not depend on local control signals but on global input signals. In this step, we evaluate the reachability of extracted sets of local timed path conditions by extracting the precise modifications of control signals, i.e., of those signals controlling the execution of the data paths extracted in the previous steps. This means that we need to analyze how they are driven by the model input signals and how these input signals are modified on their way to the routing blocks.

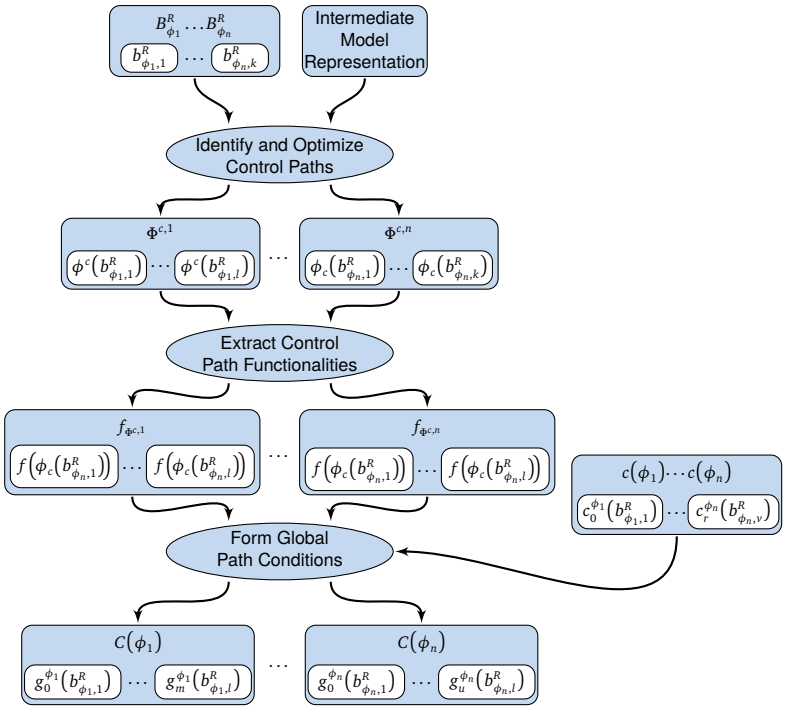


Figure 5.7: Evaluation of control signals for the routing blocks on the paths in $\Phi_{b_i \to b_o}$

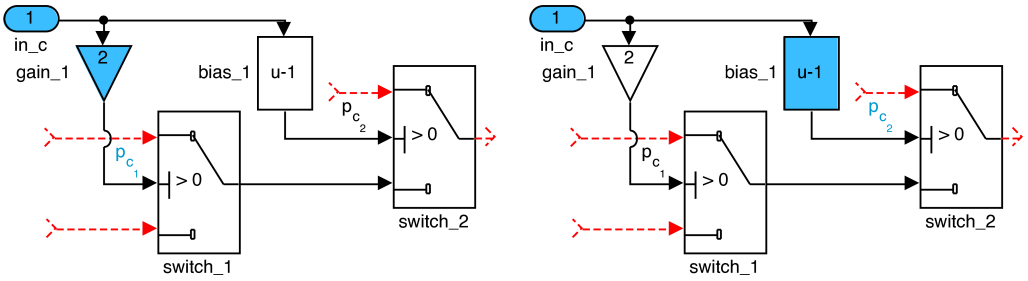
Specifically, for each path ϕ in $\Phi_{(b_i \to b_o)}$, we perform the following steps, illustrated in Figure 5.7:

1. For each routing block b on ϕ, we use our path identification algorithm to identify all paths between the global model inputs P^i and the *control signal input* p^c of b, which we call *control paths*. Each control path can either be non-cyclical[1] or cyclical.

2. For each identified control path we analyze the precise behavior of the data flowing from the global model input to the routing block input.

   (a) For non-cyclical paths, we achieve this by propagating the local path condition backwards along the path until a global model input is reached while collecting the functionality of each block along the path. The result is a *global* path condition only dependent on global model inputs.

[1] On the identified non-cyclical control paths, we perform an additional optimization step to avoid the redundant analysis of subpaths shared by multiple control paths.



(a) Control path driving block $b_{\text{switch\_1}}$ **(b)** Control path driving block $b_{\text{switch\_2}}$

Figure 5.8: Identified control path examples

(b) For each cyclical control path, we collect the functionality of each block along the control path in a similar fashion, yielding a difference equation. Subsequently, we utilize a CAS to identify whether the local timed path condition of the current routing block is satisfiable by the difference equation generated from the control path.

3. In the final step, for each routing block b on ϕ, we collect the translated sets of control paths to obtain a concise definition of the subset of the control system model which controls the execution behavior of the current path ϕ.

In the following, we illustrate each described step.

5.6.1 Identifying Control Paths

As we have introduced above, when evaluating the satisfiability of sets of control signals on paths under analysis, we elevate *local* to *global* path conditions. To analyze the dependency of these local conditions on global model inputs, we identify and extract the paths through the model connecting the control signal input of each routing block $b \in \Phi_{(b_i \to b_o)}$ to the global model inputs P^i.

For each control signal input p_c, we utilize our path identification algorithm presented in Section 5.3 to identify all paths between p_c and every global model input $p^I \in P^I$. Figure 5.8 shows an example of the identified control paths (shown in blue) driving the control signal inputs p_{c_1} and p_{c_2} of two `Switch` blocks.

5.6.2 Backwards Propagation of Non-Cyclical Control Signals

To elevate the local path conditions to model-wide conditions for information flow, we extract the data manipulations from the control paths to the routing blocks. To accomplish this, we analyze each control signal path separately and iterate over each non-cyclical path from the routing block control signal p_c to its drivers while collecting the functionality of each block.

For a single block b, we define its local functionality as shown in Definition 5.9.

Definition 5.9: Block Functionality

We denote the **functionality** of a block b as follows:

$$p_{o,k} = f_b(p_{i,k-l})$$

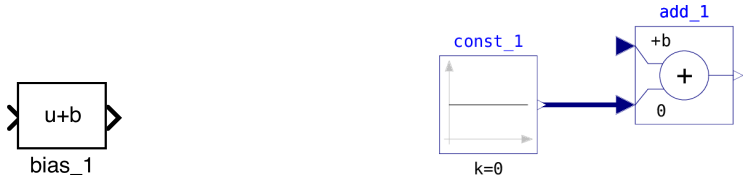
There, p_i and p_o are the input and output signal ports of block b, respectively. l describes the delay in simulation steps the block imposes on information entering through p_i, i.e., information entering b at an arbitrary simulation step t is output through p_o at simulation step $t + l$.

f_b is an expression which describes an arithmetical or logical operation the block applies to the input signal to calculate the value of the output signal, and is derived from the block type and its parameters.

When considering a complete path $\phi_{b_1 \to b_n}$, the resulting function f_ϕ is formed by the composition of each output function along the path according to its structural connections, which yields:

$$f_\phi := f_{b_n} \circ f_{b_{n-1}} \circ \ldots \circ f_{b_1}$$

For each block type, a specific set of parameters is extracted and its resulting functionality is recorded. We currently support block types implementing unary relations, i.e., blocks with a single input signal, such as blocks of type `Bias`, `Gain`, `Abs`, and `Compare`. In the following, we present the translation of the functionality of each block type presently supported by our information flow analysis algorithm.



(a) `Bias` block in Simulink **(b)** `Bias` block representation in Modelica

Figure 5.9: `Bias` block type

`Bias`. The output of a `Bias` block, shown in Figure 5.9, is calculated by performing an addition with a constant $c \in \mathbb{R}$, which is configurable by the developer. The formula we extract is therefore set as:

$$p_{o,k} = f_{\texttt{Bias}}(p_{i,k})$$
$$f_{\texttt{Bias}}(p_{i,k}) = p_{i,k} + c$$

(a) `Gain` block in Simulink **(b)** `Gain` block in Modelica

Figure 5.10: `Gain` block type

`Gain`. Similarly, the output of a `Gain` block, shown in Figure 5.10, is set by multiplication of the input signal with a constant $g \in \mathbb{R}$. The formula we extract is set as:

$$p_{o,k} = f_{\texttt{Gain}}(p_{i,k})$$
$$f_{\texttt{Gain}}(p_{i,k}) = p_{i,k} \cdot g$$

(a) `Abs` block in Simulink **(b)** `Abs` block in Modelica

Figure 5.11: `Abs` block type

`Abs`. The output of a block of type `Abs`, shown in Figure 5.11, is set by calculating the absolute value of the input signal, i.e.:

$$p_{o,k} = f_{\texttt{Abs}}(p_{i,k})$$
$$f_{\texttt{Abs}}(p_{i,k}) = |p_{i,k}|$$

(a) `Const` block in Simulink **(b)** `Const` block in Modelica

Figure 5.12: `Const` block type

`Const`. The output of a block of type `Const`, shown in Figure 5.12, is a constant value r.

$$p_{o,k} = r$$

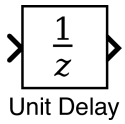
(a) `Compare` block in Simulink **(b)** `Compare` block in Modelica

Figure 5.13: `Compare` block type

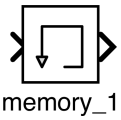
`Compare`. The functionality of a `Compare`-type block[2], shown in Figure 5.13, is set by defining a threshold $c \in \mathbb{R}$ and a comparison operator. Using both parameters, the output is set by comparing the value of the input signal with the threshold using the operation set by the developer. We extracted its functionality as:

$$p_{o,k} = f_{\texttt{Compare}}(p_{i,k})$$
$$f_{\texttt{Compare}}(p_{i,k}) = p_{i,k} \bowtie c \qquad \text{with} \quad \bowtie \in \{==, \leq, \geq, <, >\}$$

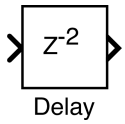
[2]The functionality of a `Compare` type block is implemented in multiple specific block types in our example languages. For example, the block types `Compare To Constant` and `Compare To Zero` implement the same behavior in MATLAB Simulink, while Modelica offers the the blocks `GreaterThreshold`, `LessThreshold`, `GreaterThanThreshold` and `LessEqualThreshold`.



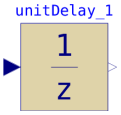
(a) `UnitDelay` block in Simulink



(b) `Memory` block in Simulink



(c) `Delay` block in Simulink



(d) `UnitDelay` block in Modelica

Figure 5.14: Time-dependent block types

Time-Dependent Model Elements. When encountering time-dependent modeling elements, as shown in Figure 5.14, on the control paths, we extract the parameters of the respective block, i.e., the delay length l, as presented in Section 5.4 and update the timing information of the function composition, such that:

$$p_{o,k} = f_b(p_{i,k-l}), \quad \text{with } b \in B^{\text{mem}}$$
$$f_b(p_{i,k}) = p_{i,k-l}$$

5.6.3 Analysis of Cyclical Control Paths

In time-discrete embedded control system models, cyclical paths describe a recurrence relation [Bonchi et al. 2017b], i.e., a relation whose result at an arbitrary time depends on current as well as prior inputs. For control paths implementing such a relation, as demonstrated in Figure 4.3b, we perform an analysis based on the translation of the control path functionality to a CAS. After extracting the functionality of the control path, i.e., the precise equation the control path implements, we instruct the CAS to identify a non-recursive solution to the relation, which forms a concise definition of the behavior of the control path at arbitrary simulation steps. Note that, while a CAS employs a range of techniques to find non-recursive solutions for arbitrary recurrence relations [Wolfram 1999; Wolfram Research 2019b], it is not guaranteed that a solution, even if existing, can be found.

To extract the recurrence relation of a given control path, our method iterates backwards over the path while recording the functionality of each block. Until an input block or a previously recorded model element is encountered, the translation recursively collects the functionality of each block. For this translation process, we utilize the block functionalities presented in the previous section and add a basic definition for 2-ary arithmetical blocks:

$$p_{o,k} = f(p_{i_1,k}, p_{i_2,k})$$
$$f(p_{i_1,k}, p_{i_2,k}) = p_{i_1,k} \bowtie p_{i_2,k} \qquad \text{with} \quad \bowtie \in \{+, -, \cdot\}$$

After constructing this concise relation describing the behavior of the control signal input of the routing block, we utilize Mathematica to identify a non-recursive solution to the recurrence relation. To obtain this solution, we use `RSolve` [Wolfram 1999, p. 96]. If the CAS is able to identify a solution, the result of this operation is a pure, non-recursive function which enables us to calculate the value of the control signal at arbitrary simulation steps. Note that, while the resulting function is non-recursive, it generally is dependent on the sequence of model inputs driving the cyclical path.

Finally, to identify the possibility of information flow in the presence of cyclical control paths, we utilize the `Reduce` functionality built into Mathematica [ibid., p. 86]. Given a function, a domain limiting the independent variable, and a set of conditions, it returns the range of function inputs in the given domain which fulfill the condition. Appendix C presents details on this translation and solution process.

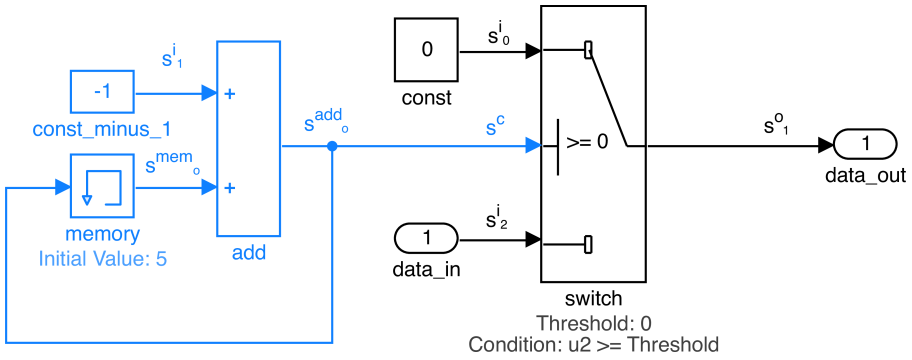
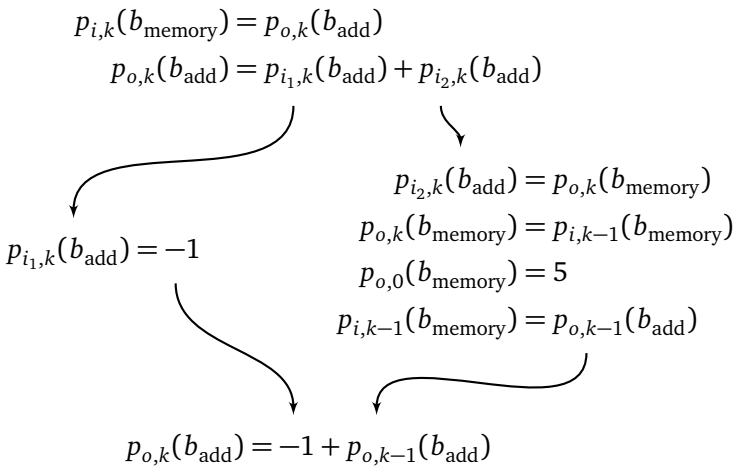


Figure 5.15: Example of a cyclical control path

To illustrate this, consider the example shown in Figure 5.15. There, a routing block is driven by a cyclical control signal s^c, shown in blue. This control signal contains the stateful modeling element b^{memory} with an initial output value of 5, the `Add` block b^{add} and

the constant model input $b^{\text{const\_minus\_1}}$. The routing block `switch` routes the signal s^{i_0} to the output signal s^{o_1} whenever the control signal s_c has a value greater than or equal to 0. When, e.g., analyzing the possibility of information flow between p^{i_2} and p^{o_1}, we are interested in the precise condition for information flow, under data as well as timing aspects.

For the path shown in Figure 5.15, we can extract the following functionalities:



The non-recursive solution to this recurrence relation, obtained using `RSolve`, is:

$$p_{o,k}(b_{\text{add}}) = 4 - k$$

This states that the value of the control signal s^c at an arbitrary simulation step k is $4-k$, i.e., a linear function depending only on the current simulation time step.

Subsequently, we call the `Reduce` function with the domain $k > 0$, as the simulation time step is non-negative, the function $p_{o,k}(b_{\text{add}}) = 4 - k$, and the timed path condition $p_{o,k}(b_{\text{add}}) \leq 0$, and receive the result:

$$p_{o,k}(b_{\text{add}}) > 0 == true \iff 0 \leq k < 4$$

This result states that the condition holds at the first four consecutive simulation time steps. Therefore, information flow between p^{i_2} and p^{o_1} is only possible after four simulation time steps have passed.

5.6.4 Composition of Global Timed Path Conditions

Using the functionalities of the non-cylical control paths extracted in Section 5.6.2, we are able to raise the scope of the path conditions from routing-block-local to model-wide by combining the local path conditions with the composed functions. Considering a single local path condition on the control signal $c_k^{\phi}(b) = p(s^c)$ and the control path functionality f_{ϕ}, which describes the precise data and timing behavior of the signal s_c depending on only global input signals, we can form a *global timed path condition* as defined in Definition 5.10.

Definition 5.10: Global Timed Path Condition

A **global timed path condition** $g_k^{\phi}(b)$ describes a necessary condition for information leaving the first block on the path ϕ at an arbitrary simulation step t to flow through routing block b along path ϕ at simulation step $t+k$. Additionally, the operation expressed by the global path condition is a comparison of a *global model input* signal with an expression.

To form a global timed path condition g_k^{ϕ}, we iteratively replace all occurrences of non-global signals in $c_k^{\phi}(b) = p(s^c)$ with the control path functionality expressed in f_{ϕ}, such that:

$$
\begin{aligned}
& & s^c &= f_{\phi}(s^i) \\
& & &= f_{b_n} \circ f_{b_{n-1}} \circ \ldots \circ f_{b_1}(s^i) \\
c_k^{\phi}(b) = p(s^c) & & s^c &= \underbrace{f_{b_n}\big(\ldots f_{b_1}(s^i)\ldots\big)} \\
g_k^{\phi}(b) &= p\Big(\overbrace{f_{b_n}\big(\ldots f_{b_1}(s^i)\ldots\big)}\Big)
\end{aligned}
$$

Analogous to Definition 5.8, we define a global timed path conditions set as shown in Definition 5.11. This set describes the global timed path conditions controlling the execution of all paths in Φ. If the conjunction of the global timed path conditions on any path described in $G\big(\Phi\big)$ is evaluated to *true*, information flow between the model elements under analysis is possible. Correspondingly, non-interference is proven if our approach is able to show that none of conjunctions of the global timed path conditions is *true*.

Definition 5.11: Timed Path Condition Set

A **global timed path conditions set** $G(\Phi)$ is defined by:

$$G(\Phi) = \bigvee_{\phi \in \Phi} \bigwedge_{b \in B_\phi^R} g_d^\phi(b), \quad \text{where } B_\phi^R \text{ describes the set of routing blocks on path } \phi$$

This describes the conjunction of all global timed path conditions on a single path, disjunctively combined over all paths in the set Φ.

Application to Motivating Example. Using the identified control signals s^c and s^{c_p} of the `Switch` blocks $b_{\text{mode\_switch}}$ and $b_{\text{public\_output\_switch}}$, respectively, as starting points, our algorithm detects the control path sets $\Phi^{c_{\text{mode\_switch}}}$ and $\Phi^{c_{\text{public\_output\_switch}}}$ with a single identified control path each:

$$\phi^{c_{\text{mode\_switch}}} = \langle p^{i_3}, s^{c_1} \rangle$$
$$\phi^{c_{\text{public\_output\_switch}}} = \langle p^{i_3}, b_{\text{compare\_public}}, s^{c_p} \rangle$$

To raise the scope of the local path conditions, our algorithm extracts the following functionalities for each block along the paths:

$$p_{o,k}(b_{\text{compare\_public}}) = p_{i,k}(b_{\text{compare\_public}}) == 2$$

Subsequently, our algorithm uses the extracted functionalities to construct a global path condition from each local path condition:

$$g_0^{\phi_1}(b_{\text{mode\_switch}}) = \left(s_0^c == 1\right)$$
$$g_1^{\phi_1}(b_{\text{public\_output\_switch}}) = \left(s_1^c == 2\right)$$

Finally, the global path condition set for the execution of ϕ_1 is constructed:

$$G(\phi_1) = \left(s_0^c == 1\right) \wedge \left(s_1^c == 2\right)$$

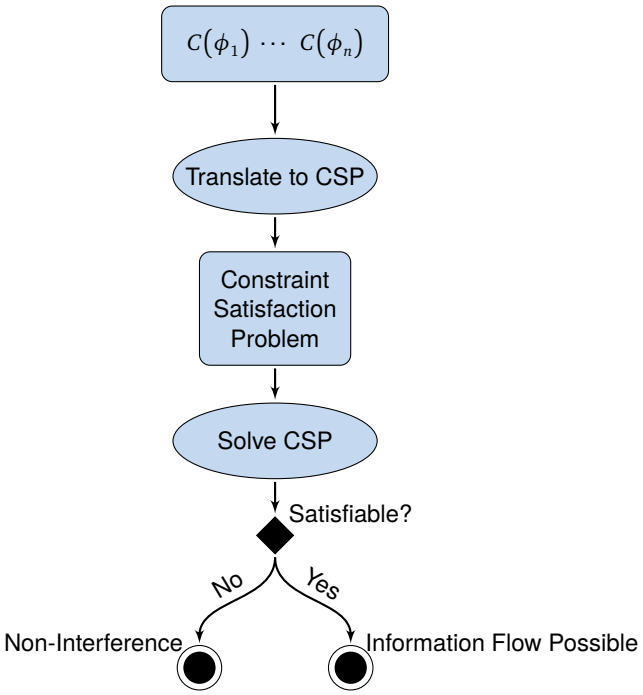


Figure 5.16: Translating and solving global path conditions

The global path condition set $G(\phi_1)$ states, that information flows on path ϕ_1 if the control signal s_c takes the value 1 at an arbitrary simulation step and the value 2 at the subsequent step.

5.7 Translating and Solving Path Conditions

As a final step of our algorithm to analyze information flow in signal-flow-oriented models, illustrated in Figure 5.16, we translate the global path conditions generated from the non-cyclical control paths in the previous step to a constraint satisfaction problem and utilize a constraint solver to achieve an automatic analysis of the satisfiability of the conditions.

As part of our framework, we have implemented the translation of global path conditions into the `MiniZinc` language [Marriott and Stuckey 2013; NICTA 2014], which is supported by a wide range of constraint solving backends, such as `JaCoP` [Kuchcinski and Szymanek 2013] and `Gecode` [Schulte et al. 2009, 2010]. In the following, we present our translation of the global path conditions, i.e., of the local path conditions as well as the extracted control paths, to the `MiniZinc` language.

For every input signal in the definitions of the global path conditions extracted in the previous step, we initially define a decision variable. To include the timing information of every signal of interest that we include in the decision process and to differentiate between the same signal in different time slices, we expand the respective decision variable definition by a suffix describing the time slice it is placed into. In MiniZinc, this takes the form shown in Listing 5.1. There, two decision variables for the same control signal s_c are defined. One for the signal sorted into time slice t, the other for the signal sorted into time slice $t + 3 \cdot t_s$.

```
1 var int: s_c_t;
2
3 var int: s_c_t_plus_3_ts;
```

Listing 5.1: Defining decision variables for control signals

Following the definition of the decision variables, we create a representation of the global path conditions, specifically the precise signal modifications along the extracted control paths by by unfolding the function composition of each global path condition. For each function application $p_{o,k} = f_b(p_i)$, this unfolding process performs the following steps:

1. As we consider the timing behavior of control paths, we define a decision variable for the current function input p_i annotated with its timing information.
2. We extract the functionality f_b and add the matching constraint connecting p_o and p_i with the functionality of f_b.

This translation of the functionality f_b of each block to MiniZinc constraints is presented in the following.

Equality. To define an equality relation between two decision variables, we use the == operator to define an equality constraint, such that:

```
p_o == p_i;
```

Bias. If an addition operation is encountered, we translate it into a equality constraint and add the parameter b, such that:

```
p_o == p_i + b;
```

```
function var int: compare(var int: in_1, var int: in_2) =
        if in_1 ⋈ in_2 then 1 else 0 endif;
```

Listing 5.2: Comparison function template

Gain. Similarly, when encountering a multiplication operation, we translate it into an equality constraint and multiply the constant g, such that:

```
p_o == p_i * g;
```

Absolute Value. For the calculation of an absolute value, we make use of the `abs` function built into `MiniZinc`, such that we create the following constraint:

```
p_o == abs(p_i);
```

Comparison. When encountering a comparison operation, we make use of the possibility to define functions in the `MiniZinc` language. For comparison operations, we developed the function template shown in Listing 5.2, which we instantiate with the corresponding assignment for the comparison operation $\bowtie \in \{==, \leq, \geq, <, >\}$. Subsequently, we add a constraint and call the function using the threshold c such that:

```
p_o == compare(p_i, c);
```

After translating the outermost function, we proceed to translate the next global path condition in the same manner and add them to the constraint solving problem we are constructing. After translating each global path condition, we instruct the constraint solver to identify a valid assignment to all decision variables given the specified constraints using the `solve satisfy` statement in the `MiniZinc` declaration. Finally, we call the constraint solver and two situations can occur: (1) The solver identifies a valid assignment to the decision variables. This means that the path conditions extracted from the routing blocks along the paths between the model elements of interest are satisfiable and, consequently, information flow *can* occur. We therefore cannot rule out the possibility of information flow and must conclude that the property of non-interference between the model elements is violated. Additionally, our algorithm offers the possibility to perform a path-wise analysis of the information flow between model elements to identify which specific path permits information flow. (2) The solver is not able to identify a solution.

```
% variables
var int: signal_out_input_3_t;
var int: signal_out_input_3_t_plus_1;

var int: signal_in_control_mode_switch_t;
var int: signal_in_control_public_output_switch_t_plus_1;

var int: signal_in_compare_public_t_plus_1;
var int: signal_out_compare_public_t_plus_1;

% functions
function var int: compare(var int: x, var int: y) =
        if x == y then 1 else 0 endif;

% constraints
constraint signal_in_control_mode_switch_t == 1;
constraint signal_in_control_mode_switch_t == signal_out_input_3_t;
constraint not (signal_in_control_public_output_switch_t_plus_1 == 0);
constraint signal_in_control_public_output_switch_t_plus_1 ==
↪  signal_out_compare_public_t_plus_1;
constraint signal_out_compare_public_t_plus_1 ==
↪  compare(signal_in_compare_public_t_plus_1, 2);
constraint signal_out_input_3_t_plus_1 == signal_in_compare_public_t_plus_1;

solve satisfy;
```

Listing 5.3: Translated constraint satisfaction problem for path ϕ_1 between blocks (p^{i_c}, p^{o_p}) through our motivating example

In this case, the path conditions specifying the execution of the paths under analysis cannot be satisfied and, consequently, the paths cannot be executed and non-interference between the model elements of interest can be shown.

In Appendix A, we present a number of examples to show the operation of our information flow analysis algorithm.

Application to Motivating Example. In the final step, our algorithm translates the extracted set of global path conditions into a constraint satisfaction problem and solves it using a constraint solver. When instructed to solve the constructed CSP shown in Listing 5.3, the constraint solver determines that a viable solution to the problem exists, i.e., the problem is satisfiable. Our approach therefore concludes that non-interference between the confidential data input and the public data output cannot be proven and that information flow is possible.

5.8 Summary

In this chapter, we have presented our methodology to analyze the information flow in signal-flow-oriented software models in the presence of complex timing behavior.

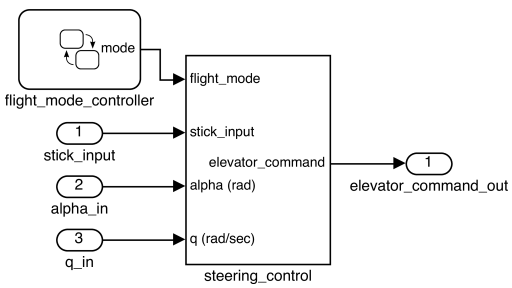
In the following chapter, we present our technique to incorporate the analysis of signal-flow-oriented models in which state-machine-based system controllers are present in the control paths of the model. Due to the strongly heterogeneous semantics of the signal-flow-oriented and state-machine-based components, a static analysis based on timed path conditions is not able to capture the complete behavior of system controllers. To enable an analysis of combined systems, we present a method based on a reachability check of the timed path conditions on the formal representation of the system controller.

6 Information Flow Analysis of Heterogeneous Models

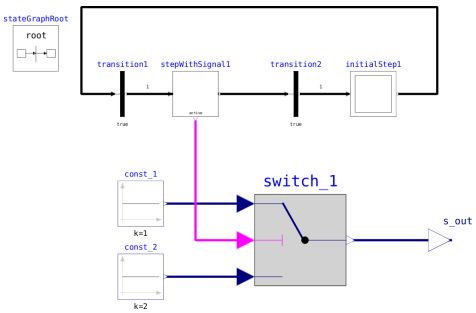
In this chapter, we present our technique to identify information flow in control system models consisting of both signal-flow-oriented and state-machine-based components. Specifically, we explain how we relate the strongly heterogeneous semantics of both modeling styles in order to enable an information flow analysis of combined models.

6.1 Approach

The analysis of embedded control system models, in which the flow of information is directed by a state-machine-based controller, poses a difficult challenge. Examples of the utilization of such controllers in our example languages Simulink/Stateflow and Modelica are shown in Figures 6.1a and 6.1b, respectively. The examples show typical modeling situations in the design of safety-critical embedded control system models. Especially Figure 6.1a, which displays a Stateflow controller embedded in a flight control model from the avionics domain, stands representative for the widespread utilization of state-machine-based controllers when modeling complex control systems in an industrial context. There, the execution of the control path from the input blocks on the left



(a) Stateflow controller in an avionics case study [Pussig et al. 2014]



(b) Modelica StateGraph controller

Figure 6.1: State-machine-based controllers in our example languages

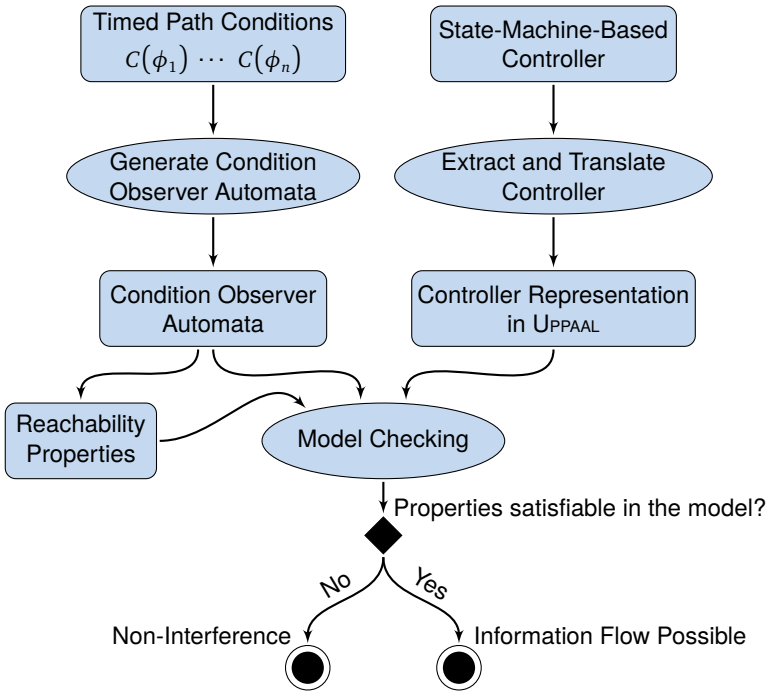


Figure 6.2: Information flow analysis of combined models using model checking

to the output blocks on the right is controlled by a state-machine-based controller embedded directly into the control system model. Even though our procedure based on the translation of sets of timed path conditions and the corresponding control paths to constraint satisfaction problems, which we have presented in Chapter 5, is able to detect the *possibility* of information flow, it is unable to prove non-interference in our motivating example shown in Section 4.1. While our approach based on timed path conditions made it possible to extract only those conditions of a model which must hold to enable the execution of specific paths under analysis, the presence of a state-machine-based controller requires a more exhaustive analysis. To identify non-interference in heterogeneous models, the timed path conditions together with the behavior of the controller under all possible input situations have to be considered.

To enable the information flow analysis of models combining signal-flow-oriented and state-machine-based components, we suggest the following approach, illustrated in Figure 6.2 [Mikulcak et al. 2018, 2019]:

Extract Timed Path Conditions. We use our method presented in Sections 5.3 to 5.5 to extract timed path conditions for all paths between a set of model elements of interest from the signal-flow-oriented components of the model. Along these paths, we gather the conditions for information flow as well as their timing and express them as sets of timed path conditions $C(\phi)$.

Translate Controller to Uppaal. To formalize the semantics of a state-machine-based controller, we extend the method presented in Jiang et al. [2016] and Yang et al. [2016] to translate embedded controller components to a system of UPPAAL timed automata. Note that in our current implementation of our framework, we only support the translation and analysis of Stateflow state machines embedded into Simulink models. We consider the adaptation of additional languages, such as the state-machine-based controller implementation language StateGraph built into Modelica, as future work and provide a more detailed discussion in Chapter 8.

Generate Condition Observer Automata. To make the representation of the control flow conditions extracted from the signal-flow-oriented components of the model compatible with the semantics of the state-machine-based controllers embedded into the models, we generate *condition observer automata* from the extracted timed path conditions. This step of our methodology is based on the idea that timed path conditions form an ordered sequence of conditions which have to be satisfied in their correct order for information flow to be enabled. To this end, we generate one *condition observer automaton* from the timed path conditions of each path represented in $C(\phi_1) \cdots C(\phi_n)$. As we extract timed path conditions for *all* paths between model elements of interest and, on these paths, extract all control flow conditions, we achieve a sound over-approximation of the possible information flow through the signal-flow-oriented components of the model, as shown in Chapter 5. At the same time, these conditions form an over-approximation of the controller behavior in such heterogeneous models.

Verify Reachability Properties. We combine the generated condition observer automata with the translated controller and generate reachability properties for the UPPAAL model checker. This enables us to effectively utilize model checking to analyze whether the timed path conditions derived from the signal-flow-oriented components can be satisfied by the embedded controller, i.e., if one or multiple of the timed path conditions expressed in $C(\phi)$ can be satisfied by the translated controller model. If one or multi-

ple timed path conditions can be satisfied, information flow over the corresponding path, and therefore between the selected source and sink of information flow, is possible. If they cannot be satisfied, on the other hand, then we have safely shown that information flow over the paths under analysis is impossible as the path is never executed in the combined model and that the property of non-interference holds.

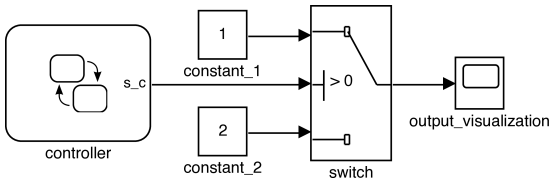
In the following, we present each step in detail.

6.2 Relating the Timing Behavior of Signal-Flow-Oriented and State-Machine-Based Components

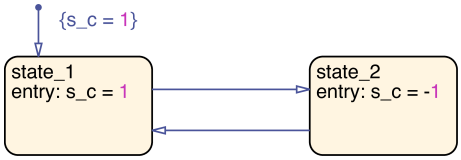
To enable the analysis of the behavior of combined signal-flow-oriented and state-machine-based model components, we explain how a state-machine-based controller is executed and how its output signals are evaluated when embedded in a signal-flow-oriented model. Figure 6.3 shows an example of a small Simulink model into which a Stateflow controller is embedded. The controller, shown in Figure 6.3a and in more detail in Figure 6.3b, is comprised of two states and sets the signal s_c, alternating between the values -1 and 1. This signal is connected to the control input port p_c of the `Switch` block b_{switch}. Whenever $s_c > 0$, the signal connected to the upper input port p_1, i.e., the constant 1 is routed through the block and to the `Scope` block, which visualizes the signal. Similarly, whenever $s_c \leq 0$, the lower input port is routed through the block, i.e., the constant 2.

Figures 6.3c to 6.3d demonstrate the activation behavior of Stateflow automata using two examples. There, output of b_{switch} is visualized over the simulation time using two different solver configurations. In Figure 6.3c, the fixed-step solver is set to a step size t_s of 1 s over 10 s, while in Figure 6.3d, the step size is set to 0.2 s. As shown there, the output signal alternates between 1 and 2 with higher frequency, i.e., depending on the solver configuration, the Stateflow controller is evaluated at different intervals even though its design did not change between simulation runs. As explained in The MathWorks [2018b], a Stateflow controller is activated once at every simulation step.

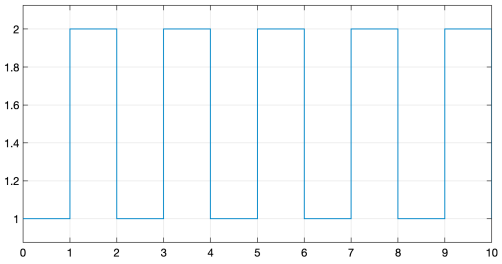
Applying this knowledge to our method, we can therefore define a known time interval between possible changes in the output of a state-machine-based automaton. Under the assumption of a uniform sample time throughout the model, this time interval is equal



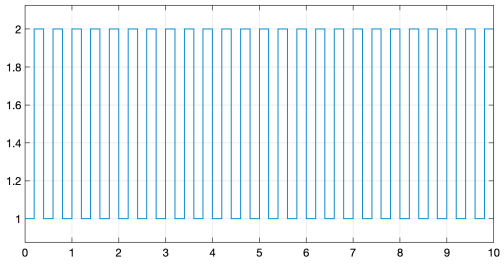
(a) Simulink model



(b) Embedded Stateflow controller



(c) Output signal behavior at $t_s = 1$



(d) Output signal behavior at $t_s = 0.2$

Figure 6.3: Timing behavior of shared Simulink/Stateflow models

to the simulation step size t_s. This makes it possible to relate the timing behavior of the signal-flow-oriented model and the evaluation of an embedded controller automaton, and enables our method to be applicable to models using arbitrary simulation step sizes.

6.3 Translating State-Machine-Based Controllers to Timed Automata

To enable the verification of properties of an embedded controller, we translate state-machine-based controllers embedded into discrete control system models into the formally well-defined UPPAAL timed automata representation. This makes it possible to use the UPPAAL model checker to verify timing as well as data properties on the controller behavior. Together with a timed automaton representation of the timed path conditions extracted by our analysis approach, this allows us to *verify* the absence of information flow through the signal-flow-oriented components of combined models. Note that in Mikulcak et al. [2016], we have published an approach to integrate the analysis of our example language MATLAB Stateflow into our information flow analysis. In this version, we have based the analysis of non-interference in heterogeneous models on a comparison of extracted sets out *timed output traces* of the state-machine-based controller with sets of timed path conditions. This version, however, offered only limited support for

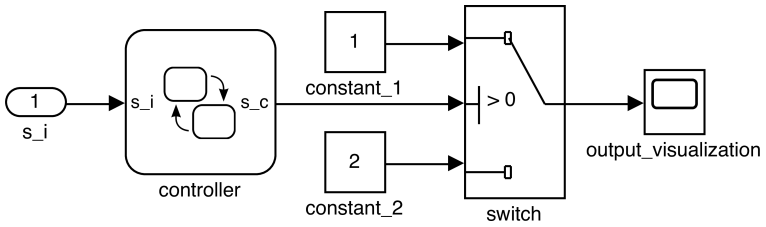
complex Stateflow features such as hierarchical states, time-dependent transition guards and the modeling of parallel states. To mitigate this, our information flow analysis builds upon a translation described in Jiang et al. [2016] and Yang et al. [2016], which poses no restrictions on the Stateflow modeling features used. Using this translation, a Stateflow automaton can be represented by a network of UPPAAL timed automata, which emulates the behavior of the original automaton both under timing as well as data aspects [Jiang et al. 2016].

In Appendix B, we provide details on this translation process of state-machine-based controllers developed in Stateflow. In the following, we present our extensions to it, and how we integrate it into our technique.

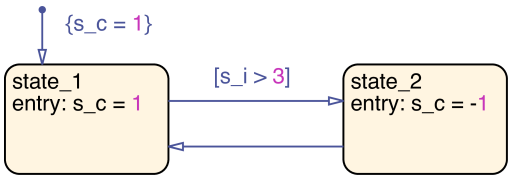
6.3.1 Generalization to Arbitrary Inputs

While the work presented in Jiang et al. [2016] and Yang et al. [2016] is able to emulate the precise data and timing behavior of Stateflow automata using the UPPAAL syntax, the authors do not consider the possibly non-deterministic environment the controller might be embedded into. This means that the communication with the surrounding signal-flow-based components, such as when reading input signals in state actions or transition guards, is not part of the verification of the controller. This makes it impossible to use the translation in its original form, as the controller model structure we use as basis for this thesis assumes the presence of one or multiple input signals to the controller. An example is shown in Figure 6.4. There, the global input signal s_i is used as an input for the system controller and used in a transition guard. Whenever the value of s_i is greater than 3, the controller activates state `state_2` and, after a single simulation step, returns to `state_1`. In the original translation, the verification never enters `state_2` as the value of s_i is held as *uninitialized*.

To overcome this limitation, we extend the original translation to enable simulation of the complete environment of the controller. To this end, we embed *generic tester automata* [Robinson-Mallett et al. 2006]. We automatically generate these tester automata for a given controller by evaluating its input signals and their respective data types. Two examples of generic tester automata are shown in Figure 6.5. Both consist of a single state and two edges. On each edge, a signal is set non-deterministically by using the `select` syntax built into UPPAAL [Behrmann et al. 2004]. In Figure 6.5a,

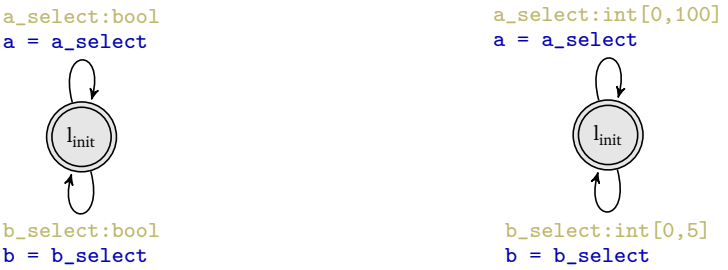


(a) A system containing an input-dependent system controller



(b) Controller implementation using a transition guarded by an input signal

Figure 6.4: A system controller utilizing input signals



(a) Setting `Boolean` signals

(b) Setting `Integer` signals

Figure 6.5: Generic tester automata

two `Boolean` signals `a` and `b` are set to either `true` or `false`, while in Figure 6.5b, the values of the two `Integer` signals are set to values from the range $[0, 100]$ and $[0, 5]$, respectively.

This makes it possible to simulate every combination of both signals in every step of the model checking process as the tester automaton runs concurrently to all other automata. We analyze all input signals to the Stateflow controller as well as their data types and ranges. Using this information, a tester automaton for the controller under analysis is constructed to simulate arbitrary inputs to the controller, thereby acting as the non-deterministic environment of the controller.

In our approach, we use the concept of generic tester automata by introducing a single automaton for each input signal to the controller. For the example shown in Figure 6.4, our algorithm creates one automaton with a single state and a single self-loop. The edge is annotated using the `select` syntax to non-deterministically choose a value from the range of the variable, which is read from our JIR.

We assume that the translation provided by Jiang et al. [2016] and Yang et al. [2016] is sound, as it provides a direct mapping of each Stateflow state and edge into a semantically equivalent timed automata representation We extend the resulting system with a *generic tester automata* that provide arbitrary input signals. This enables a sound and comprehensive analysis of the behavior of the Stateflow controller, as we simulate the complete arbitrary environments, i.e., all possible combinations of input signals to the controller.

Our extended translation of Stateflow controllers to UPPAAL timed automata presented in this section makes it possible to use the UPPAAL timed model checker to verify properties on the controller behavior. Our main contribution in this section is the generalization of the execution behavior of the translated network of timed automata to allow model checking of the controller for arbitrary inputs.

In the following, we present our technique to combine the translated system controller with the timed path conditions extracted in Chapter 5, which enables the analysis of the information flow through combined models based on model checking.

6.4 Translating Timed Path Conditions to Timed Automata

In the following, we present our method to create compatibility between the translated state-machine-based controller and the sets of global timed path conditions between signal-flow-oriented model elements of interest, as explained in Chapter 5. To create a fully automatic method to verify the absence of information flow in combined models, in this step, we generate *condition observer automata*, based on a concept first presented in Mokadem et al. [2010], for the information flow paths under analysis. Using this concept, we are able to *observe* the behavior of the translated controller under both timing and data aspects, which allows us to draw sound conclusions about the reachability of sets of global timed path conditions over the simulation time of the model for arbitrary simulation step sizes.

Consider the global timed path conditions over a single path $\phi \in \Phi$:

$$G(\phi) = \bigwedge_{b \in B_\phi^R} g_k^\phi(b)$$

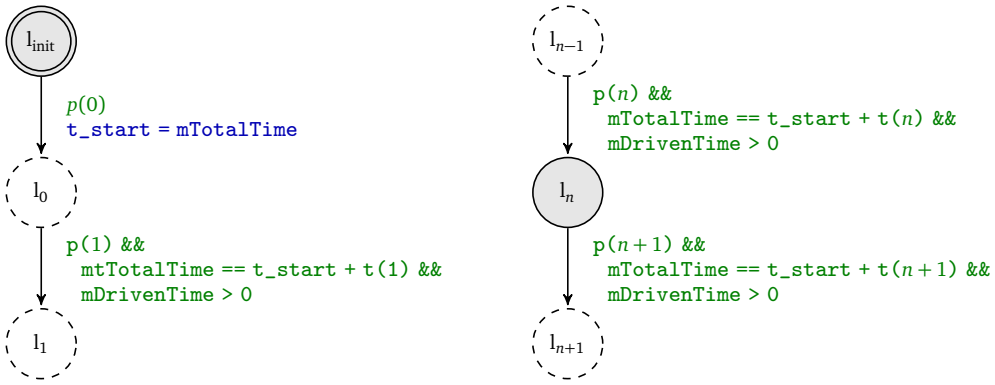
Each global timed path condition $g_k^\phi(b)$ describes the condition that has to hold on the routing block b in order for information flow to be possible along the path ϕ. From this conjunction of conditions, our approach generates a single condition observer automaton. This automaton observes the timing as well as the precise values of a given set of signals on which the global timed path conditions are defined. For each path detected between the source and sink of information flow under analysis, we generate one observer automaton from the corresponding set of global timed path conditions. A complete execution of the condition automaton, i.e., if its final location is reachable during the model checking process, signifies that each condition could be satisfied and, consequently, information flow on the corresponding path is possible.

Our generation algorithm uses this description of global timed path condition sets and first creates an initial location as well as a single location for each timed path condition for the current path. The final location, which our approach uses during the reachability check, corresponds to the final timed path condition and is denoted by l_f. Subsequently, we generate all forward transitions between the created locations as well as their guards corresponding to the global timed path conditions. As defined in Definition 5.10, a global timed path condition encodes one or multiple comparison operations of a timed signal with a constant, such as:

$$\begin{aligned} g_k^\phi(b) &= p(s_k^c) \\ &= s_k^c \bowtie g \quad \text{with } g \in \mathrm{Z}, \bowtie \in \{==, \leq, \geq, <, >\} \end{aligned}$$

Each forward transition is guarded by the condition of the timed path condition its destination location corresponds to. In other words, with every timed path condition that the controller satisfies during simulation time, the observer automaton enters its next state.

The timing condition, i.e., the precise simulation time step in which the condition is to be evaluated, is guarded depending on the simulation time representation of the controller translation as described in Appendix B.3. When entering the location corresponding the to first path condition, we record the current simulation time by reading the



(a) Initial location UPPAAL template

(b) Timed Path Conditions

Figure 6.6: Observer automata templates

variable `mTotalTime` into `t_start` and use it in subsequent transition guards to observe the timing behavior of the output signal. For a given timed path condition $g_k^{\phi}(b)$, a guard of the form `mTotalTime == t_start +` k observes the correct timing condition of output signal. There, k denotes the Stateflow simulation steps since activation of the condition observer automaton, i.e., since the first timed path condition for the current path was satisfied. The design of our timed path conditions as an ordered sequence of conditions over the simulation time, together with this corresponding translation to condition observer automata, ensures that evaluation of a set of timed path conditions can be initiated at an arbitrary simulation step. Additionally, to ensure the minimum time interval t_s between operations of the output signal, we add the guard `mDrivenTime > 0`. This encodes that at least one simulation step has passed at the current location. Note that guards on UPPAAL edges do not *enforce* but merely *enable* progress [Bengtsson and Yi 2004]. These forward transitions can therefore be taken whenever the timing and control signal requirement to proceed to the next location, i.e., timed path condition, is met. This ensures that output signal sequences with arbitrary prefixes and at arbitrary initial simulation steps can be detected. For the same reason, UPPAAL does not require the definition of backward edges or self-loops.

By checking initially whether the timed path condition that refers to the earliest time slice may be fulfilled, and then subsequently checking whether all timed path conditions referring to the subsequent time steps may be fulfilled, we ensure that we detect all sequences of outputs of the Stateflow which might satisfy the set of global timed path conditions.

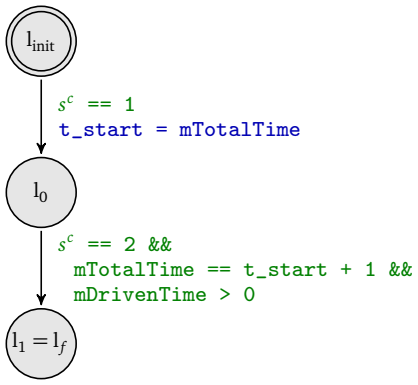


Figure 6.7: Observer automaton for path ϕ_1 of our motivating example

Figure 6.6 illustrates the result of our generation algorithm using UPPAAL timed automata templates. There, dashed states denote placeholders for additional states that our algorithm may add during the translation process, and the functions $p(n)$ and $t(n)$ denote the condition p and delay t of the global path condition of the n^{th} routing block along the current path. The template for the initial location of an observer automaton is shown in Figure 6.6a. As can be seen there, the transition leaving the initial location l_{init} is guarded by condition of the first global timed path condition and, when taken, initializes the variable `t_start` with the current simulation time step. Shown in Figure 6.6b is the template for subsequent global timed path conditions, demonstrating the timing and value guard on the forward transitions.

Application to Motivating Example. Figure 6.7 shows the automaton created from the global timed path condition set $G(\phi_1)$ of our running example, which we have presented in Section 4.1. There, we initially check whether the control signal s^c equals 1. If this condition is satisfied, progress is enabled, we record the current simulation step and may check whether s^c equals 2 in the next time slice, i.e., in the subsequent simulation step. If this happens, we can reach the final location l_f of our observer automaton, which means that the timed path condition can be satisfied. The location l_1, which corresponds to the condition of the second `Switch` block on the path, is the final location of the observer automaton. Our approach records it as the final location l_f. If this final location can be reached, the underlying Stateflow controller model can generate control signals that enable information flow along ϕ_1.

6.5 Reachability Analysis

To analyze information flow in heterogeneous control system models, the previous steps of our system have (1) extracted sets of timed path conditions from the signal-flow-oriented model components and derived *condition observer automata* from this representation; (2) translated the integrated state-machine-based controller to a UPPAAL timed automata representation; and finally (3) generalized this timed automata representation for arbitrary inputs by extending it with generic tester automata.

In the final step of our methodology, we verify the reachability of the final locations of the condition observer automata using model checking. As each observer automaton encodes the conjunction of global timed path condition set executing a single path between the model interest of interest, information flow is possible if any of the final states is reachable. Similarly, if none of the final states is reachable, then information flow between the model elements of interest is impossible and we have proven non-interference in the combined heterogeneous control system model.

To enable verification of the reachability of the final locations of the observer automata, we (1) add the generated automata to the system of UPPAAL automata that is generated from the integrated Stateflow controller; (2) generate one verification query from each recorded final location; and (3) start the verification process by calling the UPPAAL timed model checker.

After combining the observer automata with the translated controller system, we generate a query that is satisfied if the model checker identifies at least one path through the computation tree that reaches the final location. As the UPPAAL model checker uses CTL formulae, we achieve the generation of an appropriate query by using the *exists* quantifier and the *eventually* operator [Clarke and Grumberg 1999]. In combination, they assert that a given property must hold on some computation path in the future. For a single final location l_f, the query therefore takes the form:

$$EF(l_f)$$

To model check non-interference between the model elements of interest, we generate a disjunction of individual queries, such that:

$$EF(l_{f_1}) \vee \cdots \vee EF(l_{f_n}), \quad \begin{array}{l}\text{where } l_{f_1}, \ldots, l_{f_n} \text{ are the final locations of the observer} \\ \text{automata for paths } \phi_1 \ldots \phi_n\end{array}$$

If this generated query is satisfiable, i.e., if any of the final locations can be reached on any computation path, information flow between the model elements of interest is possible.

Application to Motivating Example. To analyze information flow in our motivating example, we construct a query using the information about the final location found in Figure 6.7. As we analyze the flow over a single path ϕ_1, we generate a single observer automaton with a single final location, such that the resulting query states:

```
E<> observer_phi_1_process.l_f
```

We perform the verification of this query on the translated controller system, condition observer automaton, and tester automaton for the signal `switch_mode`, using the UPPAAL timed model checker. The result shows that the query is *unsatisfiable*, i.e., that l_f cannot be reached. Using this information, we have proven that, due to the design of the controller, shown in Figure 4.1b, information flow along the path ϕ_1, i.e., between the confidential data input and the public data output is impossible and that non-interference is guaranteed. While the global timed path conditions extracted in Chapter 5 alone are satisfiable by a constraint solver, they form an over-approximation of the possible behavior of the path. Our information flow scheme based on model checking thus shows that the sequence of control signals causing confidential data to be leaked is spurious.

When examining the automaton in Figure 4.1b, it becomes apparent that upon receiving the signal to switch the operation mode via the `switch_mode` signal, the controller leaves the current operation mode and enters the state `erase_public` or `erase_confidential`, depending on the current operation mode. In these states, the content of the internal buffer are overwritten with 0 and both outputs emit the value 0. Thus, the controller ensures that every mode switch is preceded by a deletion of the internal buffer, thereby eliminating the possibility of a leak of confidential information. Using our technique, we verify the correct functionality of this deletion behavior by proving that the control signal sequence leading to a leak of information can never be emitted by the controller.

6.6 Summary

In this chapter we have presented our methodology to identify information flow in combined models consisting of both signal-flow-oriented and state-machine-based components. Our technique makes use of the translation of a given state-machine-based controller into a formally verifiable timed automata representation, which we extend by tester automata simulating all possible controller environments. In order to prove non-interference between model elements of interest, our technique verifies that the timed path conditions extracted in the previous chapter, which form the conditions the controller outputs have to fulfill, cannot be generated by the controller. To achieve this, we break down the verification of non-interference to a reachability check on condition observer automata generated from sets of global timed path conditions.

In the next chapter, we present details on the implementation of our technique as a fully automatic and extensible framework, present case studies from the automotive domain, and discuss experimental results.

7 Evaluation

To show the applicability of our approach, we have implemented it as a platform-independent modular framework and evaluated it using case studies supplied by our industrial partners from the automotive domain. In this chapter, we describe the main components of the implementation of our technique as presented in Chapters 4 to 6, a number of optimizations we developed to increase analysis performance, and introduce our case studies and experimental results. We demonstrate the applicability of our approach in industrial contexts with two case studies from the automotive domain. Our approach is applicable fully automatically and with analysis times of approximately 10 s for our case studies, can be used in industrial applications as part of an automated analysis workflow. Finally, we provide a discussion of the computational complexity of the individual steps of our methodology and its implementation.

7.1 Implementation

In this section, we present details on the architecture of our technique as a modular framework as well as its instantiation using MATLAB Simulink/Stateflow and Modelica as example languages. Additionally, we present two optimizations we have developed to lower the analysis effort for highly complex components of our technique, namely the solution of the constraint satisfaction problems we construct and the reachability analysis of final locations of our condition observer automata using timed model checking.

Our implementation is based on the analysis framework first presented in Reicherdt [2015]. This framework, developed in `Java`, uses an object-oriented representation of MATLAB Simulink/Stateflow models, the JIR, which allows us to access the structure of Simulink models and properties of each model element directly from `Java`. The JIR implements the syntax shown in Section 2.2.2. In this thesis, we have raised its scope and extended it as a language-independent intermediate representation. To this end, we have

developed a parser front-end for Modelica models, which similarly extracts the model structure and properties of every model element and stores it into our JIR. During the translation process, we perform a number of syntactical optimizations, such as unfolding of signal branches into separate signal lines, flattening of bus and subsystem structure, as well as inclusion of library and model references. This representation enables our source-language-independent system to be applicable to signal-flow-oriented control system models implemented using a wide range of modeling languages.

7.1.1 Components

The implementation of our procedure spans the components presented in the following:

Source Model Parser Front-Ends. We support the translation of two example languages from the embedded control system domain into our language-independent intermediate representation. To translate models implemented in MATLAB Simulink/Stateflow, we make use of a translation front-end originally developed in Reicherdt [2015], which we have extended to support the translation of the compressed `.slx` model file format. To enable the analysis of models implemented using Modelica, we have developed a parser front-end for models in the `.mo` model file format. Both parsers translate their respective source languages into the language-independent JIR.

Graphical Model Representation and Result Visualization. To access our implementation and start the analysis process, we use a graphical model representation developed in the Bachelor thesis of Danziger [2016]. This representation, built from the JIR, displays the model and allows the selection of arbitrary model elements as source and sink for an analysis of possible information flow. An example can be seen in Figure 7.1. There, our motivating example is displayed, with the model elements p^{i_1} and p^{o_2} selected as source and sink for our information flow analysis, respectively. Additionally, we annotate this graphical representation, as well as the source models directly, with the results of our analysis.

Path Identification Engine. We have developed a path identification component implementing a depth-first search algorithm, which we use to (1) detect syntactical paths of possible information flow between the selected model elements of interest, and (2) identify paths between routing blocks and model inputs, which form the control paths used in our methodology.

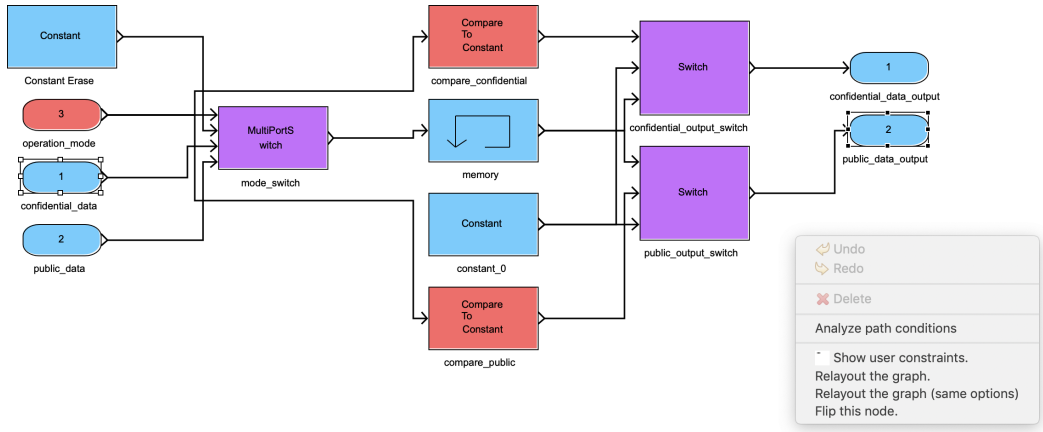


Figure 7.1: Accessing the information flow analysis via the graphical user interface

Constraint System Translation & Solver Back-Ends. To identify the satisfiability of the global timed path conditions extracted in Section 5.6, we have implemented a translation engine which emits a constraint satisfaction problem from the given global timed path conditions. Our implementation supports the translation of the conditions to the input languages of two constraint solvers: `JaCoP` and `Gecode`. While the former is accessible as a `Java` library and integrated into our implementation, the latter is called as an external binary, for which our implementation emits input files in the `MiniZinc` constraint modeling language. Additionally, our automation queries the solvers to evaluate the satisfiability of the constructed problems and presents this information as part of our graphical user interface.

Controller Translation Back-End. We use the UPPAAL timed model checker as a formally verifiable representation of the state-machine-based controllers embedded into the control system models we analyze. The implemented controller translation back-end makes use of the existing translation presented in Jiang et al. [2016] and Yang et al. [2016], which we have integrated into our `Java`-based implementation. In addition to this translation, we (1) operate directly on the generated timed automata and add generic tester automata as required by the controller, and (2) add the condition observer automata we generate from the sets of global timed path conditions. After constructing this network of timed automata and generating the appropriate set of queries, we call the UPPAAL timed model checker using its *application programming interface* (API) and present the results in our graphical user interface.

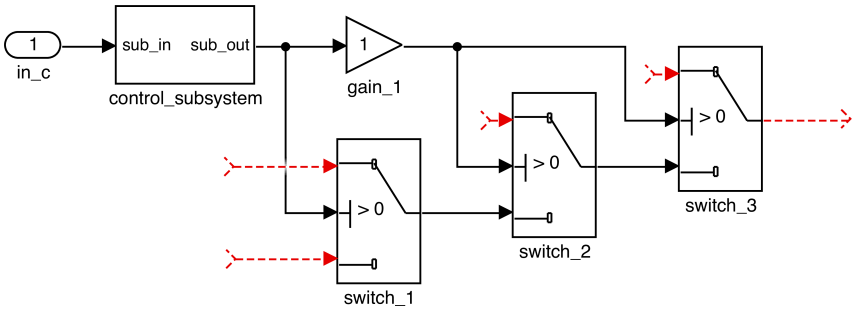


Figure 7.2: Optimizing control paths by merging identical subpaths

Currently, our framework consists of nine plug-ins for the `Eclipse` platform, which together comprise about 15,000 lines of code.

7.1.2 Optimizations

In the following, we present two optimizations we have developed as part of our implementation. Both optimizations improve the run-time of our information flow analysis.

Identifying Shared Driving Blocks

As presented in Section 5.6, our method identifies *control paths* in a model, i.e., those paths that control the execution of the routing blocks in the signal-flow-oriented components of the models. Using our path identification engine, we collect these control paths between global model inputs and the routing blocks in order to translate them into a CSP.

To minimize the size of the CSP constructed in this step of our method, we identify overlapping control paths, i.e., blocks shared by subsets of the control paths. Effectively, we identify the shortest subpath between common drivers and routing blocks. After finding these control paths, we identify the shortest common subpath shared between *all* control paths in order to identify common driving blocks of the routing block control signals.

Consider the example shown in Figure 7.2. There, the control paths starting from the control inputs of $b_{\text{switch\_2}}$ and $b_{\text{switch\_3}}$ lead to the subsystem $b_{\text{control\_subsystem}}$ and further to the global input p^{i_1}. As the subpath from the global input to $b_{\text{gain\_1}}$ is shared by both control paths, both can be shortened to include only the `Gain` block. It is therefore only necessary to include the subsystem into the CSP once, thereby reducing the size of the CSP significantly. Our implementation automatically identifies such overlapping control paths.

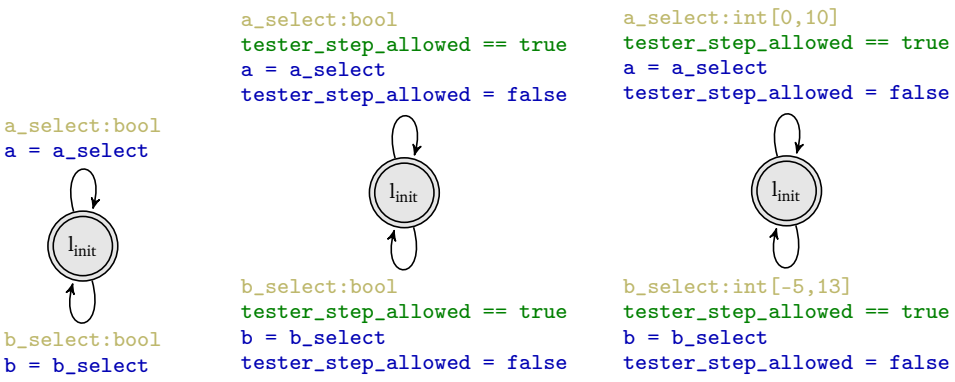
Reducing the Verification Effort

As part of our controller translation back-end, we use an existing procedure to translate MATLAB Stateflow controllers into formally verifiable UPPAAL timed automata. To support the verification of the controller environment, we extend this translation by generating and adding one *generic tester automaton* for each input to the controller. However, as during the verification process, with every non-deterministic choice by the tester automaton, a number of new computation paths have to be established,[1] the effort required for verification increases drastically. In the unoptimized controller translation, the tester automaton generates these choices at *every* step during the model checking process. However, due to the complex design of the translation from Stateflow to UPPAAL, only a small number of transitions directly relate to transitions taken in the Stateflow controller. We therefore have added a global `Boolean` variable `tester_step_allowed`, seen in Figure 7.3b, which is introduced into the automata by our controller translation back-end. In our optimized implementation of the translation process, this flag is raised only when input values to the controller are allowed to change, i.e., with every (emulated) simulation step. It is used by the guards of the edges of the generic tester automata and consumed as soon as the edge is taken. Additionally, during this optimization stage, we read information about the ranges of controller inputs from our JIR and encode them into the tester automata, such as shown in Figure 7.3c.

As we show in Section 7.2, using these optimizations, we are able to greatly reduce the effort required to verify properties of the behavior of the translated Stateflow controller [Selmke 2018].

In the following, we introduce our case studies and experimental results. First, we present an industrial case study supplied by our industrial partners and how we used our system to identify a safety and security policy violation in its implementation. Second, we present a modified version of our case study, which we use to demonstrate the ability of our system to identify information flow in the presence of cyclical control signals.

[1] Equal to the number of possible choices by the tester automaton.



(a) Unoptimized

(b) Optimized

(c) Optimized tester using variable ranges

Figure 7.3: Optimization of our generic tester automata

7.2 Case Study 1: Shared Automotive Communication Infrastructure

To show the practical applicability of our procedure, we use an industrial case study from the automotive domain. Its core is a communication infrastructure over which two distance warners, supplied by our industrial partner Assystem GmbH [Assystem 2019], and a non-critical odometer, supplied by Model Engineering Solutions GmbH [MES 2016], send and receive data. Similar bus structures connecting safety-critical and non-critical components over a shared infrastructure are the most common form of communication between *electronic control units* (ECUs) in modern motor vehicles [Van Rensburg and Ferreira 2003; Goswami et al. 2012; Antoniali et al. 2013]. However, the application of information flow analyses, which are commonly used to analyze safety as well as security properties on such systems, is a difficult challenge due to the complex control and timing behavior of the bus structure and the individual components [Koscher et al. 2010a]. Our case study from the automotive domain, shown as a MATLAB Simulink/Stateflow implementation in Figure 7.4, stands representative for the complexity of this problem.

The distance warners, situated at the front and at the back of the car, send their analysis results, i.e., proximity alerts, to the receiving component, an automated braking system. The odometer receives data from sensors on the axes of the car. The distance warners together with the automated braking system perform inherently safety-critical functions. This holds especially true when considering timing aspects, as dropped or delayed warning signals either to the driver or an automated braking system while

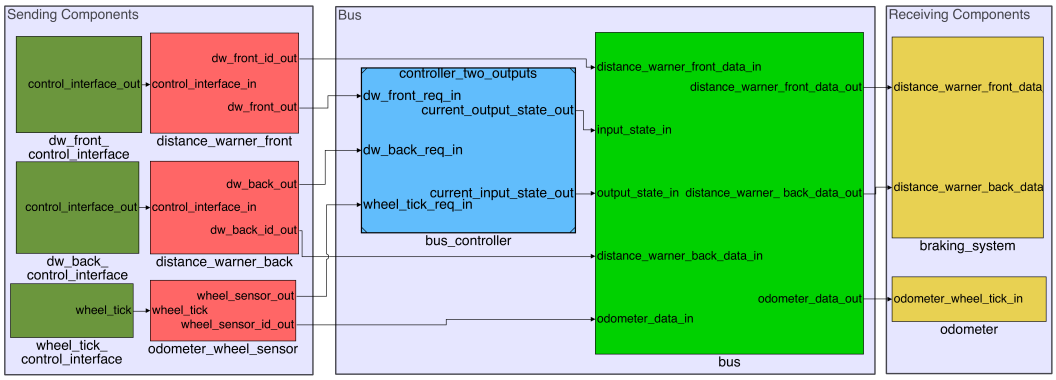


Figure 7.4: Shared automotive communication infrastructure

traveling at high speeds could cause serious accidents. The most important property of the design from a safety and security perspective is that the design has to guarantee that the braking system only receives messages from the distance warners, i.e., that information flow from the non-critical odometer wheel sensor to the critical braking system is prohibited and, consequently, *integrity* is ensured. An additional interesting property is that no information flows from the distance warners to the odometer, as this may indicate that proximity warnings are not properly received by the braking system. The overall model consists of 905 blocks and multiple layers of subsystems, making its size and complexity comparable to models with similar functionality used by our industrial partners in the automotive domain. The main challenge for the analysis of this case study is that the correct routing inherently depends on the timing of the control flow.

The three sending components seen on the left in Figure 7.4 use the bus to send their unique `id` to the receiving components on the right. Inside the channel, a system of switches reacts to the input and output states currently set by the controller and routes the data to and from the communication channel accordingly.

In the following, we present the analysis results for our first case study. These include the timed path condition extracted from the signal-flow-oriented Simulink components of the bus, the controller translation to UPPAAL, as well as analysis results and computation times.

Paths Under Analysis

As explained above, we aim at analyzing potentially critical information flow from the odometer wheel tick sensor to the braking system as well as from both distance warners to the odometer display. We denote the paths as follows:

$$\phi_1 = \phi_{\text{wheel\_sensor\_out}\rightarrow\text{distance\_warner\_front\_data\_in}}$$
$$\phi_2 = \phi_{\text{wheel\_sensor\_out}\rightarrow\text{distance\_warner\_back\_data\_in}}$$
$$\phi_3 = \phi_{\text{dw\_front\_out}\rightarrow\text{odometer\_wheel\_tick\_in}}$$
$$\phi_4 = \phi_{\text{dw\_back\_out}\rightarrow\text{odometer\_wheel\_tick\_in}}$$

In the next step of our algorithm, the paths are analyzed and sets of timed path conditions are extracted.

Extracted Timed Path Conditions

The sets of global timed path conditions extracted for each path are shown in the following:

$$g_0^{\phi_1}(b_{\text{input\_switch}}) = \left(s_{\text{input\_state}} == 1\right)$$
$$g_5^{\phi_1}(b_{\text{dw\_front\_data\_out\_switch}}) = \left(s_{\text{output\_state}} == 2\right)$$
$$g_0^{\phi_2}(b_{\text{input\_switch}}) = \left(s_{\text{input\_state}} == 1\right)$$
$$g_5^{\phi_2}(b_{\text{dw\_front\_data\_out\_switch}}) = \left(s_{\text{output\_state}} == 3\right)$$
$$g_0^{\phi_3}(b_{\text{input\_switch}}) = \left(s_{\text{input\_state}} == 2\right)$$
$$g_5^{\phi_3}(b_{\text{odometer\_out\_switch}}) = \left(s_{\text{output\_state}} == 1\right)$$
$$g_0^{\phi_4}(b_{\text{input\_switch}}) = \left(s_{\text{input\_state}} == 3\right)$$
$$g_5^{\phi_4}(b_{\text{odometer\_out\_switch}}) = \left(s_{\text{output\_state}} == 1\right)$$

In these sets, the timing depths on the communication channels in *time slices* is denoted as a subscript of the condition g. For our case study, it is calculated as 5. At this point in the analysis, due to their timing behavior, we cannot rule out the existence of

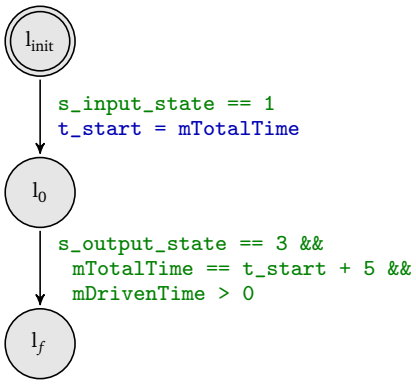


Figure 7.5: Condition observer automaton for information flow over path ϕ_1

information flow on these paths using a constraint solver, as $s_{\text{input\_state}}$ and $s_{\text{output\_state}}$ are distinct signals, both driven by the Stateflow controller responsible for operation of the bus system. It is therefore necessary to continue the analysis, i.e., to generate timed automata from each set of timed path conditions and verify whether these sets of conditions are satisfiable on the condition observer automata combined with the translated Stateflow controller.

Condition Observer Automata

For each path, our procedure collects the global timed path conditions and generates a single UPPAAL condition observer automaton for each path. As explained in Section 6.4, each automaton consists of an initial location as well as one location per entry in the condition set, i.e., three locations. To illustrate this, Figure 7.5 shows the condition observer automaton generated from the global timed path conditions $G(\phi_1)$ extracted from ϕ_1.

Network of Uppaal Automata

In the next step, we translate the controllers of our case study into a network of UPPAAL timed automata and perform the optimization steps described in Section 7.1.2. Further, our methodology combines the translated Stateflow controller with the generated observer automata by adding them to the UPPAAL system declaration. The Stateflow controller of our case study consists of six states and nine transitions which implement the *first in, first out* (FIFO)-like behavior of the shared bus. After translation, the corresponding network of UPPAAL automata consists of 17 automata, ranging in size between

Path	Time			$EF(l_f)$	Non-Interference
	Extract $G(\phi)$	Construct Controller Model	Verification		
ϕ_1	379 ms	830 ms	10.247 s	✓	×
ϕ_2	327 ms		10.559 s	✓	×
ϕ_3	354 ms		5.971 s	×	✓
ϕ_4	302 ms		5.788 s	×	✓

Table 7.1: Evaluation results for our first case study

one and four locations with a large number of self-loops: (1) Ten automata emulate the functionality and semantics of the Stateflow controller, (2) four generated observer automata, corresponding to each path under analysis, observe the data and timing behavior of the two controller output signals, (3) three optimized generic tester automata act as the non-deterministic environment. Due to the design of the case study, each tester automaton non-deterministically generates a `Boolean` value to be input into the controller signals `dw_front_req_in`, `dw_back_req_in`, and `wheel_tick_req_in`, respectively.

Verification Results

In the final step, our technique generates a single verification goal for each set of global timed path conditions $G(\phi)$, which encodes the reachability of the final location of the condition observer automaton, as presented in Section 6.5.

The results of the verification process as well as its run times are shown in Table 7.1. As can be seen there, the first steps of our approach, the extraction of timed path conditions from the combined models and evaluation of the corresponding control signals as well as the subsequent generation of UPPAAL observer automata takes between approximately 300 and 400 ms.[2] The translation of the Stateflow controller, which only has to be performed once per model as we store the translation result for each model revision, takes 830 ms.

Finally, for the cases in which the observer automaton does not reach its final location l_f, namely on ϕ_1 and ϕ_2, the verification of the combined controllers takes approximately 10 s while in all other cases, the corresponding property is verified after 5 s. The respective similarities in verification times are due to the complex structure

[2]Tested on a 2.2 GHz Intel Core i7 with 16 GB main memory, averaged over ten runs.

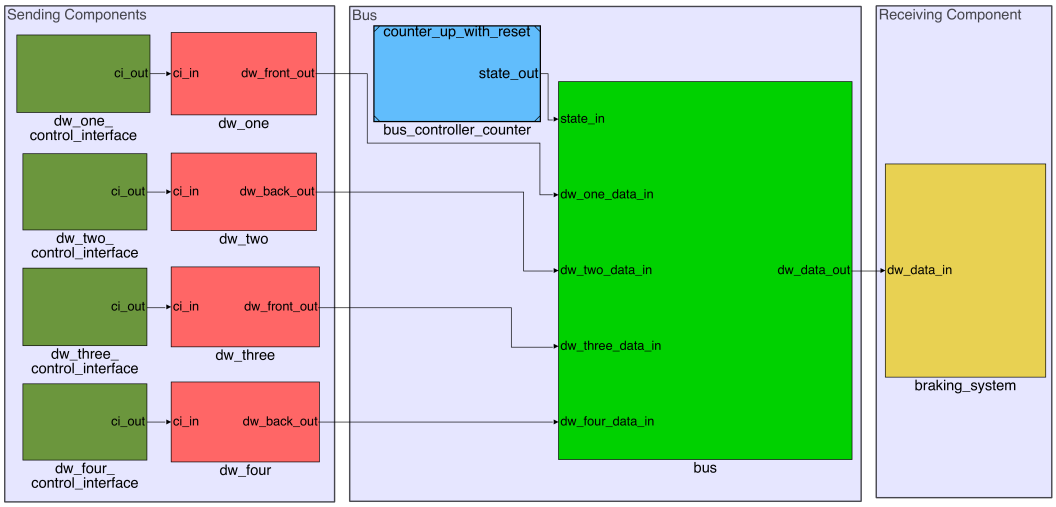


Figure 7.6: Shared automotive bus system implementing sensor error compensation

of the translated Stateflow controller behavior emulation in comparison to the observer automata. Note that using our optimization of the generic tester automata described in Section 7.1.2, we were able to decrease the necessary verification times from multiple hours to the significantly lower values seen in Table 7.1.

As our analysis shows, our technique successfully verified the *absence* of information flow over the critical paths ϕ_3 and ϕ_4. For our case study, our technique showed that there is information flow possible on the first two paths under analysis, i.e., data from the non-critical odometer may enter the braking system, as shown in the two rightmost column of Table 7.1. This is a severe violation of the property of *integrity* which potentially leads to disastrous consequences. To overcome this, we have corrected and successfully verified the controller implementation as presented in the following.

Correcting the Controller Implementation

As Table 7.1 shows, we were able to identify information flow on ϕ_1 and ϕ_2. A closer analysis of the implementation of the controller identified a faulty timed transition guard, which did not correspond to the time slice depth of the shared channel. After correcting these guards, the analysis correctly shows the absence of information flow on both paths with verification times of 5.381 s and 5.293 s and, with that, the validation of all requirements under analysis.

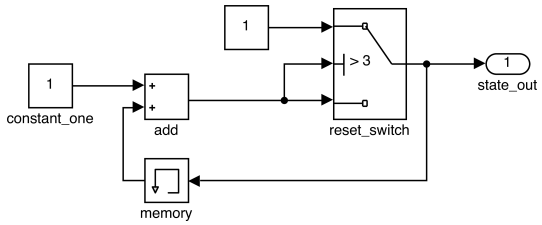


Figure 7.7: Counting controller component of our second case study

7.3 Case Study 2: Shared Communication Infrastructure using Error Compensation

While the case study presented in the previous section demonstrated how our approach is able to identify information flow in combined signal-flow-oriented and state-machine-based models, the case study we present in the following illustrates the ability of our methodology to rule out information flow in complex models in the presence of cyclical control signals. To this end, we have developed an adapted version of the case study presented in Section 7.2, shown in Figure 7.6.

This control system model implements a shared communication infrastructure, over which four safety-critical distance warners communicate with a similarly safety-critical braking system. To minimize the risk of the braking system reacting to faulty data received from a single sensor, a mechanism inside the braking system calculates a moving average of the data received by the distance warners, denoted `dw_one` to `dw_four`. To this end, the bus controller switches the distance warner which reaches the braking system with every new simulation step. The control logic acts as a counting system, which is set to select the distance warners in a round-robin fashion. We have implemented this behavior as a purely signal-flow-oriented model shown in Figure 7.7. There, the counter is implemented as a cyclical control signal connected to the `bus_router` component. This second case study consists of 895 blocks and a hierarchical complexity comparable to our first case study.

Similar to our first case study, we use our methodology to analyze safety properties of the bus infrastructure. However, unlike in our first case study, in which we used our method to identify the *absence* of information flow, we analyze whether the data of each distance warner is able to reach the braking system. The identification of non-interference would therefore indicate a violation of the system requirements.

In the following, we present the analysis results for our second case study. These include the timed path conditions extracted from the signal-flow-oriented Simulink components of the bus, the set of difference equations extracted from the cyclical control signal, as well as analysis results and computation times.

When analyzing the information flow through our second case study, the first steps are similar to the results shown in Section 7.2. We analyze four paths, connecting each distance warner to the braking system. We denote these paths ϕ_1 - ϕ_4. The local timed path conditions extracted from the `bus_router` components are shown below:

$$c_0^{\phi_1}(b_{\text{bus\_switch}}) = (s_{\text{state}} == 1)$$
$$c_0^{\phi_1}(b_{\text{bus\_switch}}) = (s_{\text{state}} == 2)$$
$$c_0^{\phi_3}(b_{\text{bus\_switch}}) = (s_{\text{state}} == 3)$$
$$c_0^{\phi_4}(b_{\text{bus\_switch}}) = (s_{\text{state}} == 4)$$

Note that due to the absence of time-dependent model elements on the bus, the time slice depth of the communication channel is 0. Additionally, due to the presence of cyclical model elements in the control signal path of our case study, the path conditions extracted here are *local*.

After extracting these local timed path conditions, our procedure follows the control signal s_{state}, encounters the cyclical model structure embedded into the subsystem `bus_controller_counter`, and extracts the difference equation that is modeled inside it. The translated system and the extracted difference equations take the following form:

$$p_{o,k}(b_{\text{add}}) = 1 + p_{o,k}(b_{\text{memory}})$$
$$p_{o,k}(b_{\text{memory}}) = 1 + p_{i,k-1}(b_{\text{memory}})$$
$$p_{o,k-1}(b_{\text{memory}}) = p_{o,k}(b_{\text{reset\_switch}})$$
$$p_{o,k}(b_{\text{reset\_switch}}) = \begin{cases} p_{o,k}(b_{\text{add}}), & \text{if } p_{o,k}(b_{\text{add}}) \leq 3 \\ 1, & \text{otherwise} \end{cases}$$

In the next step, our system uses the CAS Mathematica to find a non-recursive solution to the difference equations and to subsequently identify whether each of the extracted path conditions holds on the function output. The results and analysis times are shown

Path	Time			Non-Interference
	Extract $C(\phi)$	Extract Equation	Mathematica Evaluation	
ϕ_1	271 ms	692 ms	169 ms	×
ϕ_2	248 ms		104 ms	×
ϕ_3	285 ms		152 ms	×
ϕ_4	252 ms		159 ms	✓

Table 7.2: Evaluation results for our second case study

in Table 7.2. As can be seen there, information flow is possible over the paths ϕ_1, ϕ_2, and ϕ_3. For path ϕ_4, our system identified a non-interference relation between the fourth distance warner components and the braking system. This constitutes a violation of the correct functionality of the system as described above, as data from the fourth distance warner never reaches the braking system. When analyzing the control design, it becomes apparent that the error lies in the implementation of the state controller subsystem, which never routes data from the fourth distance warner through the switch as it resets the control state output signal whenever it reaches 3.

7.4 Analysis Complexity

In this section, we assess the complexity of our algorithm and each of its individual steps.

Model Translation. In our technique, the model translation step takes place once for each model, as the translation result is stored in a database. As every block and every signal line in the model is visited once during the translation process, we obtain a linear complexity. We obtain the overall complexity shown below. There, we denote $|V|$ as the number of blocks and $|E|$ as the number of signal lines in the model.

$$\mathcal{O}\big(|E| + |V|\big)$$

Path Identification. Our technique implements a recursive depth-first search algorithm to identify syntactical paths in a model, as shown in Section 5.3. During the search process for a single path, every block and signal line is visited at most once, incurring a complexity of:

$$\mathcal{O}\big(|V| + |E|\big)$$

Extraction of Local Timed Path Conditions. The complexity of this step of our process comprises two components. Our process (1) extracts *all* possible paths between source and sink of information flow under analysis, which incurs a complexity of $\mathcal{O}\big((|V|+|E|)^2\big)$; and (2) iterates over each path to extract the local timed path conditions. To achieve this, our method, in the worst case, has to visit each block and signal in the model once, incuring a complexity of $\mathcal{O}\big(|V|+|E|\big)$. Overall, this step of our methodology therefore possesses a worst-case complexity of:

$$\mathcal{O}\big((|V|+|E|)^3\big)$$

Extraction of Global Timed Path Conditions. After extraction of the set of local timed path conditions, our approach elevates this set to global timed path conditions, which only depend on global model inputs. In this step, our technique: (1) identifies *all* possible paths between each control block and the global model inputs, which incurs a complexity of: $\mathcal{O}\big((|V|+|E|)^3\big)$; and (2) composes the set of global timed path condition from the control paths. This incurs one iteration over each extracted control path, which possesses a worst-case complexity of $\mathcal{O}\big(|V|\big)$. Overall, this step of our methodology therefore possesses a worst-case complexity of:

$$\mathcal{O}\big((|V|+|E|)^3 \cdot |V|\big)$$

Constraint Solving. In general, constraint solving is an NP-complete problem [Dechter 2003]. However, as our methodology makes use of a specific sub-problem, i.e., the exploration of the satisfiability of a system of linear inequalities over the domain of rational numbers, a solution can be found in polynomial time [Lenstra Jr. 1983; Schrijver 1998; Matoušek 2007], such that:

$$\mathcal{O}\left(|\nu|^c\right), \quad \begin{array}{l}\text{where } |\nu| \text{ denotes the number of decision variables in the system, i.e., the} \\ \text{number of signal lines on the control paths, and } c \in \mathbb{N} \text{ and } c > 1\end{array}$$

Analysis of Cyclical Control Signals. In this step of our analysis, we use the computer algebra system Mathematica to identify non-recursive solutions to difference equations constructed from cyclical control paths. A basic algorithm used by the CAS to identify non-recursive solutions to difference equations has a worst-case complexity double exponential to the number of variables in the system [Basu 2006; Wolfram Research 2019b]. We therefore have to conservatively assume an overall complexity of:

$$\mathcal{O}(2^{2^{p(c)}}), \quad \text{with } p(c) \text{ a polynomial function of } c \in \mathbb{N} \text{ and } c > 1$$

Generation of Formal Controller Models & Observer Automata. Both operations, the translation of a state-machine-based controller into a formally verifiable representation and the generation of observer automata from sets of global timed path conditions, possess a complexity linear to the number of states in the controller and conditions in the set, respectively. However, as both are bounded by the overall number of blocks in the system, the complexity of both operations is:

$$\mathcal{O}\big(2 \cdot |V|\big) = \mathcal{O}\big(|V|\big)$$

Controller Verification. In this step of our method, we use the UPPAAL model checker to perform a reachability check for a given set of states in our observer automata. Unfortunately, the verification of timed automata does not generally scale well and the effort grows exponentially to the number of clocks used in the system [Bérard et al. 2001]. However, as the translation of Stateflow controllers to UPPAAL automata presented in Jiang et al. [2016] and Yang et al. [2016] does not utilize clocks, the complexity is reduced to the problem of checking CTL properties on the system. As stated in Clarke et al. [1986], Schnoebelen [2002], and Lomuscio and Raimondi [2006], the complexity of this problem is P-complete for a given transition system. The size of this transition system, however, grows exponentially in the number of locations and variables used in the system.

Discussion. As we have shown in this section, while most components of our methodology have linear or polynomial computational complexity over the size of the model under analysis, individual components of our methodology, such as the formal verifica-

tion of state-machine-based controllers or the solution of cyclical control signals, exceed this. However, as these high-complexity components generally receive only very small input sets, as presented in Appendix A as well as this chapter, our technique scales comparatively well for the practical examples we have seen at our partners from the automotive industry. There, the, e.g., control signal paths are comparatively short and the state-machine-based controllers are typically of sizes similar to the size used in our case studies. Consequently, the analysis times shown in Tables 7.1 and 7.2 demonstrate the applicability of our methodology for practical examples.

7.5 Summary

In this chapter, we have described the architecture of our implementation and details on its individual components, as well as optimizations we have developed to reduce the analysis effort. Further, we have demonstrated the practical applicability of our method using two case studies from the automotive domain. Our implemented platform-independent analysis framework can be applied fully automatically to control system models implemented in our two example languages, MATLAB Simulink/Stateflow and Modelica. In addition to that, it offers a convenient graphical user interface to visualize its analysis results.

Our first case study implements a shared communication infrastructure, over which two safety-critical distance warners and a non-critical wheel tick sensor communicate with an automated braking system and an odometer. The three sending components share a communication medium structured as a bus, to which they request access. A controller, implemented as a state-machine-based Stateflow automaton, grants access and configures the bus to receive and send data from the appropriate sources to their corresponding targets. We have used our technique to analyze whether (1) the *integrity* of the safety-critical functionality of the braking system can be ensured by identifying non-interference between the non-critical wheel tick sensor and the braking system, and (2) ensured the correct routing of safety-critical distance warner messages to the braking system by identifying non-interference between both distance warners and the odometer. Our second case study implements an error-compensating implementation of a set of four distance warners which use a bus infrastructure to send their distance data to a braking system. The bus ensures that the distance data is provided to the braking system in a round-robin fashion in order for the braking system to calculate a moving average of the distance results of the distance warners. For this case study, we have used our

approach to verify whether information from each individual distance warner is able to reach the braking system, i.e., that the property of non-interference does *not* hold on the paths connecting each distance warner to the braking system.

The average run-times for our case studies lie at around 1 s for purely signal-flow-oriented control system models and at between 5-10 s for heterogeneous models containing both signal-flow-oriented and state-machine-based components. Although the worst-case time complexity of our procedure is double exponential, these analysis times necessary to analyze information flow through our industrial-sized case studies from the automotive industry demonstrate that our fully automatic implementation can be used in industrial applications.

8 Conclusion

In this chapter, we summarize the results of this thesis. We review the criteria we have presented in the introduction and discuss whether our proposed process meets them. Then, we give an outlook on future work.

8.1 Results

In this thesis, we have presented a novel technique that enables a sound information flow analysis of discrete embedded control system models. Our technique considers the timing behavior of such models as well as the concurrency inherent to their semantics. In addition to that, it offers the analysis of control system models comprised of both signal-flow-oriented and state-machine-based components, which combine fundamentally different semantics and modeling styles.

The first step of our approach is the analysis of information flow through signal-flow-oriented components of discrete embedded control system models. The main idea of this first step is the extraction of only that information from a model that is required to analyze information flow in respect to both timing and functionality. To this end, our methodology captures the precise control, data and timing conditions under which information flow is enabled as well as when and how these conditions are triggered. Model aspects that do not influence the information flow between elements of interest are discarded to increase analysis performance. We provide an algorithm to propagate these conditions backwards through the model and thus enable the fully-automated computation of *timed path conditions*, i.e., conditions under which information flow is possible that solely depend on the model inputs.

The second step of our system enables the analysis of heterogeneous control system models combining the strongly differing semantics of signal-flow-oriented and state-machine-based models. To this end, we first translate a state-machine-based con-

troller into a formally verifiable representation and, second, combine this representation with *condition observer automata* which we generate from the timed path conditions extracted in the first step of our scheme. This enables us to use the well-established technique of model checking to identify precisely that behavior that leads to the execution of information flow paths under analysis.

Using our method, we are able to safely rule out the existence of information flow between arbitrary components of a model. This enables us to reason about non-interference between model elements of interest and the compliance with security as well as safety properties.

Our methodology supports signal-flow-oriented modeling languages, the combined analysis of signal-flow-oriented and state-machine-based modeling styles and various modeling languages. It is applicable fully automatically and we have demonstrated its practical applicability with results from two industrial case studies.

Signal-Flow Semantics. The integration of the specific semantic properties of signal-flow-oriented modeling languages, i.e., the complex notion of timing and the inherent concurrency, enables our methodology to soundly analyze information flow through control system model components based on the signal-flow-oriented modeling paradigm. To capture timed control flow dependencies, we have developed the concept of timed path conditions, which express necessary conditions under which paths of possible information flow are executed, as well as *when*, with respect to the simulation time, these conditions must hold for information to flow. Subsequently, we translate the sets of timed path conditions to a constraint satisfaction problem and use a constraint solver to identify overlap between the conditions. If no overlap is detected, we have successfully identified non-interference between the model elements under analysis.

Combined Analysis. To enable the analysis of models combining signal-flow-oriented and state-machine-based semantics, we have developed a formal view on the execution behavior of state-machine-based controllers integrated into signal-flow-oriented control system models as well as on the interaction between components of both paradigms. Using our translation of state-machine-based controllers into formally well-defined timed automata representations and the generation of sets of *observer automata* from our timed path conditions, we are able to effectively utilize model checking to analyze non-interference in heterogeneous control system models.

Language Support. We have provided an instantiation of our information flow analysis method for discrete embedded control system models implemented using two example languages widely used in the domain of embedded control systems: MATLAB Simulink/ Stateflow and Modelica.

Automation. We have provided an implementation of our methodology as a modular framework, which enables us to fully automatically analyze information flow through a comparatively large subset of discrete MATLAB Simulink/Stateflow and Modelica models. Furthermore, our platform-independent implementation offers a convenient graphical user interface, which provides access to a graphical view of the control system model, and to define sink and source of information flow to be analyzed. To formally verify the absence of information flow in combined models, we use the UPPAAL timed model checker. To enable the automatic analysis of constraint satisfaction problems, our framework implements solver back-ends using the `Gecode` and `JaCoP` constraint solvers.

Applicability. Using our implementation, we have automatically evaluated safety and security policies in industrial case studies from the automotive domain. With our approach, we were able to successfully identify violations of the requirement of non-interference between safety-critical and non-critical components in our first case study. Similarly, our methodology was able to detect an error in the implementation of our second case study by detecting non-interference between a pair of communicating components. The analyses were performed in approximately 10 s and 1 s for our first and second case study, respectively. Note that the current instantiation of our method limits the control path complexity to unary arithmetical blocks and to those cyclical dependencies supported by our computer algebra system. Additionally, the algorithms employed do not scale well in the worst case. However, our results show that our procedure and its fully automatic implementation are applicable to industrial-sized models from the domain of safety-critical embedded control systems.

As these results show, we have successfully met our criteria defined in Chapter 1.

8.2 Outlook

We have presented a method to analyze the information flow in discrete embedded control system models and have implemented it for two widely-used example language from the domain of safety-critical embedded systems, MATLAB Simulink/Stateflow and Modelica.

Our approach and its implementation support the analysis of heterogeneous models implemented using signal-flow-oriented as well as state-machine-based components and can be applied fully automatically. We were able to show its practical applicability with our experimental results. In particular, we have demonstrated the performance and the capabilities of our technique to identify violations in safety and security policies. However, there are still open questions that are worth investigating in further research.

Relaxing Assumptions. In Section 4.3, we have presented the assumptions our method imposes on the models it is able to analyze, such as the limitation to a uniform sample time throughout the model and to unary functions on unnested non-cyclical control paths. An extension to relax the former limitation would make use of factorization of component-specific sample times to support multi-rate models, i.e., models utilizing elements with non-uniform sample times. As, in signal-flow-oriented languages, each model element must use a sample time which is divisible by the simulation step size, our definition of a fixed-delay dependency, shown in Definition 5.3, could be adapted to hold a delay length not in absolute simulation time steps, but expressed as a coefficient of the simulation step size. The latter limitation is twofold. We chose the limitation to unary functions on unnested non-cyclical control paths, as shown in Figure 4.3a, due to the absence of control paths containing model elements of higher complexity in our case studies from the automotive domain. A relaxation of this limitation to unary modeling elements, however, is possible by extending our generation of global timed path conditions and the corresponding subsequent translation into constraint satisfaction problems. An extension of Definition 5.9 to support an arbitrary number of block input signals would be sufficient. Subsequently, it would be possible to extend our implementation to support additional translation rules for additional model elements on the control path. The model elements available for translation would, at this point, be limited by the underlying constraint solver, such that complex elements, e.g., integrators or division blocks, would not be possible to be translated into constraint operators directly. An extension of our methodology to support control paths containing n-ary blocks and non-uniform sample times would widen the range of control system models supported by our methodology. A relaxation of the limitation to unnested control paths, on the other hand, would require an analysis step to identify whether a stable fixed point in the control path behavior exists.

Additional Verification Back-Ends for State-Machine-Based Components. An additional limitation of our approach and its implementation is that in its current instantiation, it is restricted to the analysis of state-machine-based controllers developed in MATLAB Stateflow. The extension to controllers developed using Modelica StateGraph would require the development of a translation of such controllers to networks of UPPAAL timed automata. Due to the, in comparison to MATLAB Stateflow, limited expressiveness of Modelica StateGraph controllers, we expect this translation to be less complex than the solution presented in Jiang et al. [2016] and Yang et al. [2016].

Precise Limitations of Cyclical Control Path Solution Technique. Furthermore, an interesting direction to consider is the exploration of the *precise* limitations of our process to analyze information flow through models containing cyclical control paths. As we have presented in Section 5.6.3, we use the CAS Mathematica to (1) identify a non-recursive solution to the difference equation we extract from the cyclical control signal path components, and (2) analyze whether the identified non-recursive function, if found, is able to satisfy the timed path condition at the control flow element in question. As we cannot precisely determine which techniques Mathematica employs to obtain solutions to the family of difference equations extracted from cyclical paths, as future work, we could devise a range of case studies utilizing varying degrees of difference equations and arithmetical model elements to better understand the capabilities of the CAS employed by our process.

Model Repair Techniques. The utilization of our system as a basis for a number of analysis and verification techniques for control system models is a highly promising endeavor. One interesting direction, for example, is the development of methods to automatically repair safety and security policy violations. To this end, we plan to use our concept of timed path conditions as well as our backward propagation scheme for control signal paths to identify those signal sequences which lead to an information leak. Subsequently, it would be possible to develop an technique which proposes and, if possible, automatically repairs these leaks based on the information extracted by our information flow analysis. Consider the signal-flow-oriented components of our motivating example presented in Figure 4.1. As the data stored in the buffer is delayed in relation to the state signal, confidential information is potentially leaked through the public data output. An automated technique would be able to, e.g., insert a time-dependent memory element to delay the state signal on its path to the set of switches on the right. A similar repair

technique could be imagined for the state-machine-based components of the control system models our technique is able to analyze. Using the UPPAAL timed model checker, it is possible to identify the precise conditions and sequence of states as well as input signals leading to an arbitrary point in the model checking process. A repair technique could use this information to modify the state-machine-based controller accordingly.

Model Partitioning. One promising utilization of our method is the *partitioning* of control system models beyond the scope of syntactical features. In essence, our information flow analysis method could be extended as a semantic slicing tool and, thus, as a preparatory step for further analysis and verification systems, which decreases the size and complexity of the source model by partitioning. The required analysis effort of systems which rely on a verification of a control system model as a whole, such as presented in Reicherdt [2015] and Liebrenz et al. [2018], increases dramatically with increasing model size. To leverage the possibilities of our method, a possible extension could be the identification of the precise conditions of coupling between parts of a control system model. For a given model, such an analysis could identify conditions that must hold on model components and signals as well as on input data in order for model parts to be executed. To calculate such conditions, we could leverage our concept of timed path conditions. A prerequisite to extend our technique to support partitioning of control system models could be the development of a classification of data inside a model using a security type system. Such systems, which are the basis of various information control systems, could be extended by a notion of timing to be applicable to the domain of control system models based on signal-flow-oriented semantics. Based on such a system, our technique could automatically and semantically-aware classify model components into safety and security levels. For example, a `Memory` element, as shown in our motivating example in Figure 4.1, would be a member of a high as well as a low security level, depending on the current state of the system. For analysis methods based on verification, such a partitioning and classification technique could be highly beneficial as it has the possibility to greatly reduce the verification effort.

List of Definitions

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
BMBF	Bundesministerium Für Bildung Und Forschung
CAN	Controller Area Network
CAS	Computer Algebra System
CFG	Control Flow Graph
CIM	Computation-Independent Model
CISMo	Change Impact Analyses For Software Models
CLP	Constraint Logic Programming
COP	Constraint Optimization Problem
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
CTL	Computation Tree Logic
DSL	Domain-specific Language
ECoSMo	Effective Complexity Of Software Models
ECU	Electronic Control Unit
FCSP	Finite Constraint Satisfaction Problem
FIFO	First In, First Out
IFA	Information Flow Analysis
IFC	Information Flow Control
ISO	International Organization For Standardization
JIR	Java Intermediate Representation
MDD	Model-driven Development
MES	Model Engineering Solutions GmbH
ML	MATLAB

NFC	Near Field Communication
PAT	Process Analysis Toolkit
PDG	Program Dependence Graph
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QM	Quality Management
RTL	Register-transfer Level
SESE	Software And Embedded Systems Engineering
SF	Stateflow
SFG	Signal-flow Graph
SIL	Safety Integrity Level
SL	Simulink
SLDV	Simulink Design Verifier
SMT	Satisfiability Modulo Theory
TA	Timed Automaton
TPC	Timed Path Condition
TTA	Time-triggered Architecture
UP	UPPAAL
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Bibliography

Articles, Theses & Books

P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund (2004). "Designing Safe, Reliable Systems using SCADE". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, pp. 115–129 (cit. on p. 37).

J. R. Abrahams (1965). *Signal Flow Analysis*. Oxford: Pergamon Press. ISBN: 978-0-08-010677-9 (cit. on p. 10).

A. Albers, K. Gschweitl, C. Schyr, and S. Kunzfeld (2006). "Methods and Tools for Model-Based Validation of Hybrid Powertrains". In: *ATZ worldwide* 108.11, pp. 26–28 (cit. on p. 8).

R. Alur and D. L. Dill (1994). "A Theory of Timed Automata". In: *Theoretical Computer Science* 126.2, pp. 183–235 (cit. on p. 30).

G. Andersson, P. Bjesse, B. Cook, and Z. Hanna (2002). "A Proof Engine Approach to Solving Combinational Design Automation Problems". In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No. 02CH37324)*. IEEE, pp. 725–730 (cit. on p. 39).

M. Antoniali, M. Girotto, and A. M. Tonello (Aug. 2013). "In-Car Power Line Communications: Advanced Transmission Techniques". In: *International Journal of Automotive Technology* 14.4, pp. 625–632. DOI: 10.1007/s12239-013-0067-2 (cit. on p. 102).

Assystem (Jan. 2019). *Assystem Germany GmbH*. URL: https://www.assystem-germany.com/ (cit. on p. 102).

G. Barany and J. Signoles (2017). “Hybrid Information Flow Analysis for Real-World C Code”. In: *Tests and Proofs*. Ed. by S. Gabmeyer and E. B. Johnsen. Cham: Springer International Publishing, pp. 23–40. ISBN: 978-3-319-61467-0 (cit. on p. 34).

M. Barnett, K. R. M. Leino, and W. Schulte (2004). “The Spec# Programming System: An Overview”. In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, pp. 49–69 (cit. on p. 38).

S. Basu (2006). *Algorithms in Real Algebraic Geometry*. Berlin New York: Springer. ISBN: 978-3-540-33098-1 (cit. on p. 112).

G. Behrmann, A. David, and K. G. Larsen (2004). “A Tutorial on UPPAAL”. In: *Formal Methods for the Design of Real-Time Systems*. Springer, pp. 200–236 (cit. on pp. 30, 88).

E. Bell and L. LaPadula (1973). *Secure Computer Systems: Mathematical Foundations*. Tech. rep. DTIC Document (cit. on p. 22).

R. Bell (2006). “Introduction to IEC 61508”. In: *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*. Vol. 55. SCS ’05. Sydney, Australia: Australian Computer Society, Inc., pp. 3–12. ISBN: 1-920-68237-6 (cit. on p. 8).

L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara (Feb. 2019). “Recent Advances and Trends in On-Board Embedded and Networked Automotive Systems”. In: *IEEE Transactions on Industrial Informatics* 15.2, pp. 1038–1051. DOI: 10.1109/tii.2018.2879544 (cit. on p. 43).

J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi (1996). “UPPAAL - A Tool Suite for Automatic Verification of Real-Time Systems”. In: *Hybrid Systems III: Verification and Control*. Ed. by R. Alur, T. A. Henzinger, and E. D. Sontag. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 232–243. ISBN: 978-3-540-68334-6. DOI: 10.1007/BFb0020949 (cit. on p. 54).

J. Bengtsson and W. Yi (2004). “Timed Automata: Semantics, Algorithms and Tools”. In: *Lectures on Concurrency and Petri Nets*. Springer, pp. 87–124 (cit. on pp. 30, 92).

B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, P. Mckenzie, and P. McKenzie (2001). “UPPAAL – Timed Systems”. In: *Systems and Software Verification: Model-Checking Techniques and Tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 153–159. ISBN: 978-3-662-04558-9. DOI: 10.1007/978-3-662-04558-9_15 (cit. on p. 112).

J.-F. Bergeretti and B. A. Carré (Jan. 1985). “Information-Flow and Data-Flow Analysis of **while**-Programs”. In: *ACM Trans. Program. Lang. Syst.* 7.1, pp. 37–61. ISSN: 0164-0925. DOI: 10.1145/2363.2366 (cit. on p. 23).

G. Berry (1989). “Real-Time Programming: Special Purpose or General Purpose Languages”. PhD thesis. Le Chesnay, France: INRIA (cit. on p. 9).

K. J. Biba (1977). *Integrity Considerations for Secure Computer Systems*. Tech. rep. DTIC Document (cit. on p. 20).

F. Bonchi, J. Holland, D. Pavlovic, and P. Sobocinski (2017a). “Refinement for Signal-Flow Graphs”. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 85. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (cit. on p. 2).

F. Bonchi, P. Sobociński, and F. Zanasi (Feb. 2017b). “The Calculus of Signal Flow Diagrams I: Linear relations on streams”. In: *Information and Computation* 252, pp. 2–29. DOI: 10.1016/j.ic.2016.03.002 (cit. on pp. 73, 165).

P. Boström (2011). “Contract-Based Verification of Simulink Models”. In: *Lecture Notes in Computer Science*, pp. 291–306. DOI: 10.1007/978-3-642-24559-6_21 (cit. on p. 38).

P. Boström, L. Morel, and M. Waldén (2007). “Stepwise Development of Simulink Models Using the Refinement Calculus Framework”. In: *International Colloquium on Theoretical Aspects of Computing*. Springer, pp. 79–93 (cit. on p. 11).

P. Braun, M. Broy, F. Houdek, M. Kirchmayr, M. Müller, B. Penzenstadler, K. Pohl, and T. Weyer (Feb. 2014). “Guiding Requirements Engineering for Software-Intensive Embedded Systems in the Automotive Industry”. In: *Computer Science - Research and Development* 29.1, pp. 21–43. ISSN: 1865-2042. DOI: 10.1007/s00450-010-0136-y (cit. on p. 2).

H. Brückmann, J. Strenkert, U. Keller, B. Wiesner, and A. Junghanns (2009). “Model-Based Development of a Dual-Clutch Transmission Using Rapid Prototyping and SiL”. In: *International VDI Congress Transmissions in Vehicles* (cit. on p. 19).

R. E. Bryant (June 2004). “System Modeling and Verification with UCLID”. In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04.* Pp. 3–4. DOI: 10.1109/MEMCOD.2004.1459805 (cit. on p. 38).

P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis (2003). "Translating Discrete-Time Simulink to Lustre". In: *Embedded Software*. Ed. by R. Alur and I. Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 84–99. ISBN: 978-3-540-45212-6 (cit. on pp. 37, 38).

H. Chalé, O. Taofifenua, T. Gaudré, A. Topa, N. Lévy, and J.-L. Boulanger (June 2011). "Reducing the Gap Between Formal and Informal Worlds in Automotive Safety-Critical Systems". In: *INCOSE International Symposium* 21.1, pp. 1306–1320. DOI: 10.1002/j.2334-5837.2011.tb01287.x (cit. on p. 43).

S. Checkoway et al. (2011). "Comprehensive Experimental Analyses of Automotive Attack Surfaces." In: *USENIX Security Symposium* (cit. on p. 1).

C. Chen, J. Sun, Y. Liu, J. S. Dong, and M. Zheng (2012). "Formal Modeling and Validation of Stateflow Diagrams". In: *International Journal on Software Tools for Technology Transfer* 14.6, pp. 653–671. DOI: 10.1007/s10009-012-0235-0 (cit. on p. 40).

Z. Chen, L. Chen, and B. Xu (Sept. 2014). "Hybrid Information Flow Analysis for Python Bytecode". In: *2014 11th Web Information System and Application Conference*. DOI: 10.1109/wisa.2014.26 (cit. on p. 34).

Y. Cherdantseva and J. Hilton (2013). "A Reference Model of Information Assurance & Security". In: *8th International Conference on Availability, Reliability and Security*. IEEE, pp. 546–555 (cit. on p. 20).

J. Choi and S.-i. Jin (2019). "Security Threats in Connected Car Environment and Proposal of In-Vehicle Infotainment-Based Access Control Mechanism". In: *Advanced Multimedia and Ubiquitous Engineering*. Ed. by J. J. Park, V. Loia, K.-K. R. Choo, and G. Yi. Singapore: Springer Singapore, pp. 383–388. ISBN: 978-3-662-47487-7 (cit. on p. 1).

E. Chrisofakis, A. Junghanns, C. Kehrer, and A. Rink (2011). "Simulation-Based Development of Automotive Control Software with Modelica". In: *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*. 063. Linköping University Electronic Press, pp. 1–7 (cit. on p. 19).

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella (2002). "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *International Conference on Computer Aided Verification*. Springer, pp. 359–364 (cit. on p. 38).

A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri (1999). “NuSMV: A New Symbolic Model Verifier”. In: *International conference on computer aided verification*. Springer, pp. 495–499 (cit. on p. 38).

E. M. Clarke, E. A. Emerson, and A. P. Sistla (1986). “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2, pp. 244–263 (cit. on p. 112).

E. M. Clarke and E. A. Emerson (1981). “Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic”. In: *Workshop on Logic of Programs*. Springer, pp. 52–71 (cit. on p. 29).

E. M. Clarke and O. Grumberg (1999). *Model Checking*. Cambridge: MIT Press. ISBN: 0-262-03270-8 (cit. on pp. 30, 94).

M. Conrad (2004). “Modell-basierter Test eingebetteter Software im Automobil (Model-based Testing of Embedded Automotive Software), ser”. PhD thesis. Universität Wiesbaden (cit. on p. 15).

S. Dajani-Brown, D. Cofer, and A. Bouali (2004). “Formal Verification of an Avionics Sensor Voter Using SCADE”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Y. Lakhnech and S. Yovine. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 5–20. ISBN: 978-3-540-30206-3 (cit. on p. 37).

A. Danziger (2016). “Implementierung einer grafischen Lösung zur Darstellung von MATLAB/Simulink Modellen”. Supervised by: Sebastian Schlesinger and Sabine Glesner. BA thesis. Technische Universität Berlin (cit. on p. 98).

N. De Francesco and L. Martini (Sept. 2007). “Instruction-Level Security Analysis for Information Flow in Stack-Based Assembly Languages”. In: *Information and Computation* 205.9, pp. 1334–1370. DOI: 10.1016/j.ic.2007.04.002 (cit. on p. 34).

L. De Moura and N. Bjørner (2008). “Z3: An Efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 337–340 (cit. on p. 38).

L. De Moura, H. Rueß, and M. Sorea (2003). “Bounded Model Checking and Induction: From Refutation to Verification”. In: *International Conference on Computer Aided Verification*. Springer, pp. 14–26 (cit. on p. 38).

R. Dechter (2003). *Constraint Processing*. San Francisco: Morgan Kaufmann Publishers. ISBN: 9780080502953 (cit. on p. 111).

D. E. Denning and P. J. Denning (1977). "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20.7, pp. 504–513 (cit. on p. 21).

D. E. Denning (1976). "A Lattice Model of Secure Information Flow". In: *Communications of the ACM* 19.5, pp. 236–243 (cit. on p. 22).

U. Donath, J. Haufe, T. Blochwitz, and T. Neidhold (Mar. 2008). "A New Approach for Modeling and Verification of Discrete Control Components Within a Modelica Environment". In: *Proceedings of the 6th International Modelica Conference*. Ed. by B. Bachmann. The Modelica Association, pp. 269–276 (cit. on p. 19).

J. S. Fenton (1973). "Information Protection Systems." PhD thesis. University of Cambridge (cit. on p. 22).

R. Frevert, J. Haase, R. Jancke, U. Knochel, P. Schwarz, R. Kakerow, and M. Darianian (2006). *Modeling and Simulation for RF System Design*. Springer Science & Business Media (cit. on p. 8).

P. Fritzson (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Piscataway, New Jersey: IEEE Press Wiley-Interscience. ISBN: 0-471-471631 (cit. on p. 17).

P. Fritzson and V. Engelson (1998). "Modelica — A Unified Object-Oriented Language for System Modeling and Simulation". In: *European Conference on Object-Oriented Programming*. Springer, pp. 67–90 (cit. on p. 17).

T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace (1992). "Constraint Logic Programming". In: *Logic Programming in Action*. Ed. by G. Comyn, N. E. Fuchs, and M. J. Ratcliffe. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 3–35. ISBN: 978-3-540-47312-1 (cit. on p. 27).

T. Gerlitz and S. Kowalewski (2016). "Flow Sensitive Slicing for MATLAB/Simulink Models". In: *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, pp. 81–90 (cit. on p. 36).

J. A. Goguen and J. Meseguer (1982). "Security Policies and Security Models". In: *IEEE Symposium on Security and Privacy*, pp. 11–20 (cit. on p. 25).

— (1984). “Unwinding and Inference Control”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, pp. 75–75 (cit. on p. 25).

D. Goswami, R. Schneider, A. Masrur, M. Lukasiewycz, S. Chakraborty, H. Voit, and A. Annaswamy (July 2012). “Challenges in Automotive Cyber-Physical Systems Design”. In: *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 346–354. DOI: 10.1109/SAMOS.2012.6404199 (cit. on p. 102).

R. M. Graham (1967). “Protection in an Information Processing Utility”. In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP ’67. New York, NY, USA: ACM. DOI: 10.1145/800001.811674 (cit. on p. 33).

B. Hailpern and P. Tarr (2006). “Model-Driven Development: The Good, the Bad, and the Ugly”. In: *IBM Systems Journal* 45.3, pp. 451–461. ISSN: 0018-8670. DOI: 10.1147/sj.453.0451 (cit. on p. 2).

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud (1991). “The Synchronous Data Flow Programming Language Lustre”. In: *Proceedings of the IEEE* 79.9, pp. 1305–1320 (cit. on p. 37).

C. Hammer (July 2009). *Information Flow Control for Java. A Comprehensive Approach Based on Path Conditions in Dependence Graphs*. Karlsruhe: Universitätsverlag Karlsruhe. ISBN: 978-3-86644-398-3 (cit. on pp. 20, 22, 33).

C. Hammer, J. Krinke, and G. Snelting (2006). “Information Flow Control for Java Based on Path Conditions in Dependence Graphs”. In: *IEEE International Symposium on Secure Software Engineering*, pp. 87–96 (cit. on p. 26).

C. Hammer, R. Schaade, and G. Snelting (2008). “Static Path Conditions for Java”. In: *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. ACM, pp. 57–66 (cit. on p. 26).

C. Hammer and G. Snelting (2009). “Flow-Sensitive, Context-Sensitive, and Object-Sensitive Information Flow Control Based on Program Dependence Graphs”. In: *International Journal of Information Security* 8.6, pp. 399–422 (cit. on pp. 25, 33).

G. Hamon (2005). “A Denotational Semantics for Stateflow”. In: *ACM International Conference on Embedded Software*. ACM, pp. 164–172 (cit. on p. 40).

G. Hamon and J. Rushby (2004). “An Operational Semantics for Stateflow”. In: *Fundamental Approaches to Software Engineering*. Springer, pp. 229–243 (cit. on p. 40).

R. M. Haralick and L. G. Shapiro (Feb. 1979). “The Consistent Labeling Problem: Part I”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1.2, pp. 173–184. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1979.4766903 (cit. on p. 28).

D. Harel (1987). “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming*. Vol. 8. 3. Elsevier, pp. 231–274 (cit. on pp. 15, 41).

D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld (2014). “JSFlow: Tracking information flow in JavaScript and its APIs”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, pp. 1663–1671 (cit. on p. 34).

T. A. Henzinger, P.-H. Ho, and H. Wong-Toi (1997). “HyTech: A Model Checker for Hybrid Systems”. In: *International Conference on Computer Aided Verification*. Springer, pp. 460–463 (cit. on p. 39).

P. Herber, R. Reicherdt, and P. Bittner (2013). “Bit-Precise Formal Verification of Discrete-Time MATLAB/Simulink Models using SMT Solving”. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10. DOI: 10.1109/EMSOFT.2013.6658586 (cit. on pp. 38, 40).

C. A. R. Hoare (1985). *Communicating Sequential Processes*. Englewood Cliffs, N.J: Prentice/Hall International. ISBN: 978-0-13-153271-7. (Cit. on p. 39).

J. Hutchinson, J. Whittle, and M. Rouncefield (Sept. 2014). “Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure”. In: *Science of Computer Programming* 89, pp. 144–161. DOI: 10.1016/j.scico.2013.03.017 (cit. on p. 43).

IEC (Apr. 2010). *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Standard. International Electrotechnical Commission (cit. on p. 8).

ISO (Nov. 1993). *ISO 11898*. Tech. rep. International Organization for Standardization, p. 65 (cit. on pp. 1, 45).

— (July 2009). *ISO/DIS 26262 - Road Vehicles - Functional Safety*. Tech. rep. Geneva, Switzerland: International Organization for Standardization (cit. on pp. 1, 2, 8, 9).

— (2015). *ISO 11898-1*. Tech. rep. International Organization for Standardization (cit. on p. 45).

J. Jaffar and J.-L. Lassez (1987). “Constraint Logic Programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’87. Munich: ACM, pp. 111–119. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41635 (cit. on p. 27).

Y. Jiang, Y. Yang, H. Liu, H. Kong, M. Gu, J. Sun, and L. Sha (Apr. 2016). “From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–11. DOI: 10.1109/RTAS.2016.7461337 (cit. on pp. 41, 42, 85, 88, 90, 99, 112, 119, 161, 162, 163).

A. Joshi and M. P. E. Heimdahl (2005). “Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier”. In: *Computer Safety, Reliability, and Security*. Ed. by R. Winther, B. A. Gran, and G. Dahll. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 122–135. ISBN: 978-3-540-32000-5 (cit. on pp. 37, 40).

J. C. King (1976). “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7, pp. 385–394 (cit. on p. 26).

M. Klenk, D. G. Bobrow, J. d. Kleer, and B. Janssen (2014). “Making Modelica Applicable for Formal Methods”. In: *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. 96. Linköping University Electronic Press; Linköpings universitet, pp. 205–211 (cit. on p. 39).

A. G. Kleppe, J. B. Warmer, and W. Bast (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional (cit. on p. 8).

P. Kocher, J. Jaffe, and B. Jun (1999). “Differential Power Analysis”. In: *Advances in Cryptology—CRYPTO’99*. Springer, pp. 388–397 (cit. on p. 21).

P. C. Kocher (1996). “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology — CRYPTO ’96*. Ed. by N. Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 104–113. ISBN: 978-3-540-68697-2 (cit. on p. 21).

B. Köpf and D. Basin (2006). “Timing-Sensitive Information Flow Analysis for Synchronous Systems”. In: *Computer Security – ESORICS 2006*. Ed. by D. Gollmann, J. Meier, and A. Sabelfeld. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 243–262. ISBN: 978-3-540-44605-7 (cit. on p. 35).

K. Koscher et al. (May 2010a). “Experimental Security Analysis of a Modern Automobile”. In: *2010 IEEE Symposium on Security and Privacy*, pp. 447–462. DOI: 10.1109/SP.2010.34 (cit. on p. 102).

K. Koscher et al. (2010b). “Experimental Security Analysis of a Modern Automobile”. In: *IEEE Symposium on Security and Privacy*. IEEE, pp. 447–462 (cit. on p. 1).

K. Kuchcinski and R. Szymanek (2013). “Jacop - Java Constraint Programming Solver”. In: *International Conference on Principles and Practice of Constraint Programming*. unpublished (cit. on p. 78).

B. Kuo and F. Golnaraghi (2009). *Automatic Control Systems*. 9th ed. Hoboken, N.J: Wiley. ISBN: 978-0470048962 (cit. on p. 9).

B. W. Lampson (1969). “Dynamic Protection Structures”. In: *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*. AFIPS ’69 (Fall). Las Vegas, Nevada: ACM, pp. 27–38. DOI: 10.1145/1478559.1478563 (cit. on p. 33).

B. W. Lampson (1973). “A Note on the Confinement Problem”. In: *Communications of the ACM* 16.10, pp. 613–615 (cit. on p. 22).

E. A. Lee and S. Neuendorffer (Mar. 2005). “Concurrent Models of Computation for Embedded Software”. In: *IEE Proceedings - Computers and Digital Techniques* 152.2, pp. 239–250. ISSN: 1350-2387. DOI: 10.1049/ip-cdt:20045065 (cit. on p. 11).

P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire (1991). “Programming Real-Time Applications with SIGNAL”. In: *Proceedings of the IEEE* 79.9, pp. 1321–1336 (cit. on p. 37).

F. Leitner (May 2008). *Evaluation of the* MATLAB *Simulink Design Verifier versus the Model Checker Spin*. Tech. rep. University of Konstanz (cit. on p. 39).

H. W. Lenstra Jr. (1983). “Integer Programming with a Fixed Number of Variables”. In: *Mathematics of operations research* 8.4, pp. 538–548 (cit. on p. 111).

T. Liebrenz, P. Herber, and S. Glesner (2018). “Deductive Verification of Hybrid Control Systems Modeled in Simulink with KeYmaera X”. In: *Lecture Notes in Computer Science*, pp. 89–105. DOI: 10.1007/978-3-030-02450-5_6 (cit. on pp. 38, 39, 40, 120).

Y. Liu, J. Sun, and J. S. Dong (2008). “An Analyzer for Extended Compositional Process Algebras”. In: *Companion of the 30th international conference on Software engineering*. ACM, pp. 919–920 (cit. on p. 40).

Z. Liu, J. Woodcock, H. Zhu, A. Cavalcanti, and A. Mota (2013). "Simulink Timed Models for Program Verification". In: *Theories of Programming and Formal Methods*. Vol. 8051. Springer Berlin Heidelberg, pp. 82–99. ISBN: 978-3-642-39697-7. DOI: 10.1007/978-3-642-39698-4_6. URL: http://dx.doi.org/10.1007/978-3-642-39698-4_6 (cit. on p. 39).

A. Lomuscio and F. Raimondi (2006). "The Complexity of Model Checking Concurrent Programs against CTLK Specifications". In: *International Workshop on Declarative Agent Languages and Technologies*. Springer, pp. 29–42 (cit. on p. 112).

H. Lundvall, P. Bunus, and P. Fritzson (2004). "Towards Automatic Generation of Model Checkable Code from Modelica". In: *Proceedings of the 45th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2004)*, pp. 23–24 (cit. on p. 39).

A. K. Mackworth and E. C. Freuder (1993). "The Complexity of Constraint Satisfaction Revisited". In: *Artificial Intelligence* 59.1-2, pp. 57–62 (cit. on p. 27).

K. Marriott, P. Stuckey, and P. Stuckey (1998). *Programming with Constraints: An Introduction*. Adaptive Computation and Machine. MIT Press. ISBN: 9780262133418 (cit. on p. 28).

S. J. Mason (1953). "Feedback Theory-Some Properties of Signal Flow Graphs". In: *Proceedings of the IRE* 41.9, pp. 1144–1156 (cit. on p. 10).

J. Matoušek (2007). *Understanding and Using Linear Programming*. Berlin: Springer. ISBN: 3-540-30697-8 (cit. on p. 111).

S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy (2016). "A Security Analysis of an In-Vehicle Infotainment and App Platform." In: *WOOT* (cit. on p. 1).

B. Meenakshi, A. Bhatnagar, and S. Roy (2006). "Tool for Translating Simulink Models into Input Language of a Model Checker". In: *International Conference on Formal Engineering Methods*. Springer, pp. 606–620 (cit. on p. 38).

MES (June 2016). *Model Engineering Solutions GmbH*. URL: model-engineers.com (cit. on p. 102).

S. Messaoud (2014). "Translating Discrete-Time Simulink to SIGNAL". PhD thesis. Virginia Tech (cit. on p. 38).

M. Mikulcak, P. Herber, T. Göthel, and S. Glesner (Sept. 2016). "Towards Identifying Spurious Paths in Combined Simulink/Stateflow Models". In: *INFORMATIK 2016*. Ed. by M. P. Heinrich C. Mayr. Vol. P-259. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik. GI Bonn, pp. 1495–1508. URL: https://dl.gi.de/20.500.12116/1037 (cit. on pp. 5, 87).

— (2017). "Timed Path Conditions in MATLAB/Simulink". In: *System Level Design from HW/SW to Memory for Embedded Systems*. Ed. by M. Götz, G. Schirner, M. A. Wehrmeister, M. A. Al Faruque, and A. Rettberg. Cham: Springer International Publishing, pp. 64–76. ISBN: 978-3-319-90023-0. DOI: 10.1007/978-3-319-90023-0_6 (cit. on p. 5).

— (June 2018). "Information Flow Analysis of Combined Simulink/Stateflow Models". In: *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 223–228. DOI: 10.1109/WETICE.2018.00050 (cit. on pp. 5, 84).

— (June 2019). "Information Flow Analysis of Combined Simulink/Stateflow Models". In: *Journal of Information Technology And Control* 48.2, pp. 299–315. DOI: 10.5755/j01.itc.48.2.21759 (cit. on pp. 5, 84).

C. Miller and C. Valasek (Aug. 2015). "Remote Exploitation of an Unaltered Passenger Vehicle". In: *Black Hat USA* (cit. on p. 1).

S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl (2005). "Formal Verification of Flight Critical Software". In: *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, pp. 15–18 (cit. on pp. 38, 40).

D. K. Misra (2004). "Signal-Flow Graphs and Their Applications". In: *Radio-Frequency and Microwave Communication Circuits*. Wiley, pp. 392–421. ISBN: 9780471653769. DOI: 10.1002/0471653764.ch10 (cit. on pp. 2, 9).

H. B. Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J.-M. Roussel (2010). "Verification of a Timed Multitask System with UPPAAL". In: *IEEE Transactions on Automation Science and Engineering* 7.4, pp. 921–932. DOI: 10.1109/etfa.2005.1612699 (cit. on p. 90).

N. Navet and F. Simonot-Lion (2009). *Automotive Embedded Systems Handbook*. Ed. by R. Zurawski. Vol. 1. Industrial Information Technology Series. Boca Raton: CRC Press. ISBN: 978-0-8493-8026-6 (cit. on p. 2).

J. Nellen, T. Rambow, M. T. B. Waez, E. Ábrahám, and J.-P. Katoen (2018). “Formal Verification of Automotive Simulink Controller Models: Empirical Technical Challenges, Evaluation and Recommendations”. In: *Lecture Notes in Computer Science*, pp. 382–398. DOI: 10.1007/978-3-319-95582-7_23 (cit. on pp. 39, 44).

A. Niederliński (2011). *A Quick and Gentle Guide to Constraint Logic Programming Via ECLiPSe*. 3rd ed. Gliwice: Jacek Skalmierski Computer Studio. ISBN: 978-83-62652-08-2 (cit. on p. 27).

F. Nielson, H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*. Springer (cit. on pp. 23, 25, 34).

S. Osder (Jan. 1999). “Practical View of Redundancy Management Application and Theory”. In: *Journal of Guidance, Control, and Dynamics* 22.1, pp. 12–21. DOI: 10.2514/2.4363 (cit. on p. 37).

M. Otter, K.-E. Årzén, and I. Dressler (2005). “Stategraph - A Modelica Library for Hierarchical State Machines”. In: *Modelica 2005 proceedings* (cit. on p. 19).

M. Otter et al. (Sept. 2015). “Formal Requirements Modeling for Simulation-Based Verification”. In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. DOI: 10.3384/ecp15118625 (cit. on p. 39).

A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner (2007). “Software Engineering for Automotive Systems: A Roadmap”. In: *2007 Future of Software Engineering*. FOSE ’07. Washington, DC, USA: IEEE Computer Society, pp. 55–71. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.22 (cit. on p. 2).

B. Pussig, J. Denil, P. De Meulenaere, and H. Vangheluwe (Jan. 2014). “Generation of Functional Mock-up Units for Co-Simulation from Simulink R, using Explicit Computational Semantics”. In: vol. 46 (cit. on p. 83).

J.-P. Queille and J. Sifakis (1982). “Specification and Verification of Concurrent Systems In CESAR”. In: *International Symposium on programming*. Springer, pp. 337–351 (cit. on p. 29).

R. Reicherdt (July 2015). “A Framework for the Automatic Verification of Discrete-Time Matlab Simulink Models Using Boogie”. PhD thesis. Technische Universität Berlin (cit. on pp. 36, 38, 39, 40, 55, 57, 97, 98, 120).

R. Reicherdt and S. Glesner (2012). “Slicing MATLAB/Simulink Models”. In: *34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 551–561. DOI: 10.1109/ICSE.2012.6227161 (cit. on p. 36).

C. Robinson-Mallett, R. M. Hierons, and P. Liggesmeyer (2006). “Achieving Communication Coverage in Testing”. In: *ACM SIGSOFT Software Engineering Notes* 31.6, pp. 1–10. DOI: 10.1145/1218776.1218786 (cit. on p. 88).

A. Rodrigues da Silva (Oct. 2015). “Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model”. In: *Computer Languages, Systems & Structures* 43, pp. 139–155. DOI: 10.1016/j.cl.2015.06.001 (cit. on p. 43).

A. Sabelfeld and A. Myers (2003). “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications* 21.1, pp. 5–19 (cit. on pp. 25, 33).

J. Saltzer and M. Schroeder (1975). “The Protection of Information in Computer Systems”. In: *Proceedings of the IEEE* 63.9, pp. 1278–1308 (cit. on p. 20).

I. Sander (2003). “System Modeling And Design Refinement In ForSyDe”. PhD thesis. Stockholm, Sweden: Royal Institute of Technology (cit. on p. 9).

N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi (2004). “Defining and Translating a Safe Subset of Simulink/Stateflow into Lustre”. In: *International Conference on Embedded Software*. ACM, pp. 259–268 (cit. on p. 41).

P. Schnoebelen (2002). “The Complexity of Temporal Logic Model Checking”. In: *Advances in Modal Logic* 4.393-436, p. 35 (cit. on p. 112).

A. Schrijver (1998). *Theory of Linear and Integer Programming*. Chichester New York: Wiley. ISBN: 0471982326 (cit. on p. 111).

J. Schroeder, C. Berger, T. Herpel, and M. Staron (2015). “Comparing the Applicability of Complexity Measurements for Simulink Models During Integration Testing: An Industrial Case Study”. In: *Proceedings of the Second International Workshop on Software Architecture and Metrics*. SAM ’15. Florence, Italy: IEEE Press, pp. 35–40 (cit. on p. 3).

C. Schulte, M. Lagerkvist, and G. Tack (2009). “Gecode: Generic Constraint Development Environment”. In: *INFORMS Annual Meeting* (cit. on p. 78).

F. Selmke (Feb. 2018). “Analyse und Optimierung des Übersetzungsprozesses von MATLAB/Stateflow-Automaten in Systeme von zeitbehafteten Automaten”. Supervised by: Marcus Mikulcak and Sabine Glesner. BA thesis. Technische Universität Berlin (cit. on p. 101).

A. Singh, M. Kar, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay (2017). “Improved Power Side Channel Attack Resistance of a 128-Bit Aes Engine with Random Fast Voltage Dithering”. In: *ESSCIRC 2017-43rd IEEE European Solid State Circuits Conference*. IEEE, pp. 51–54 (cit. on p. 21).

G. Stavrakakis (Jan. 2018). “Information Flow Analysis for Simulink Models with Cyclic Control Signals”. Supervised by: Marcus Mikulcak, Sebastian Schlesinger, Paula Herber, and Sabine Glesner. MA thesis. Technische Universität Berlin.

G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas (2004). “Secure Program Execution via Dynamic Information Flow Tracking”. In: *ACM Sigplan Notices*. Vol. 39. 11. ACM, pp. 85–96 (cit. on p. 25).

J. Sun, Y. Liu, J. S. Dong, and C. Chen (2009). “Integrating Specification and Programs for System Modeling and Verification”. In: *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, pp. 127–135 (cit. on p. 40).

J. Sutherland, K. Oizumi, K. Aoyama, N. Takahashi, and T. Eguchi (2016). “System-Level Design Trade Studies by Multi Objective Decision Analysis (MODA) utilizing Modelica”. In: *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*. 124. Linköping University Electronic Press, pp. 61–69 (cit. on pp. 3, 19).

N. Tanković, D. Vukotić, and M. Žagar (June 2012). “Rethinking Model Driven Development: analysis and opportunities”. In: *Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces*, pp. 505–510. DOI: 10.2498/iti.2012.0414 (cit. on p. 2).

M. Tiller, P. Bowles, and M. Dempsey (2003). “Development of a Vehicle Model Architecture in Modelica”. In: *3rd International Modelica Conference* (cit. on p. 19).

A. Tiwari (2002). *Formal Semantics and Analysis Methods for Simulink/Stateflow Models*. Tech. rep. SRI International. URL: http://www.csl.sri.com/users/tiwari/html/stateflow.html (cit. on p. 15).

T. K. Tolstrup, F. Nielson, and H. R. Nielson (2005). “Information Flow Analysis for VHDL”. In: *Parallel Computing Technologies*. Ed. by V. Malyshkin. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 79–98. ISBN: 978-3-540-31826-2 (cit. on p. 34).

S. Tripakis, C. Sofronis, P. Caspi, and A. Curic (2005). “Translating Discrete-Time Simulink to Lustre”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 4.4, pp. 779–818 (cit. on p. 37).

F. Truyen (Jan. 2006). “The Fast Guide to Model Driven Architecture”. In: *Cephas Consulting Corp* (cit. on p. 7).

P. J. Van Rensburg and H. C. Ferreira (2003). “Automotive Power-Line Communications: Favourable Topology for Future Automotive Electronic Trends”. In: *Proceedings of the 7th International Symposium on Power-Line Communications and Its Applications (ISPLC'03)*, pp. 103–108 (cit. on p. 102).

G. Walde and R. Luckner (2015). *Automatic Translation of Complex Flight Control Systems from Simulink/Stateflow to SCADE - An Experience Report*. Tech. rep. Deutsches Zentrum für Luft- und Raumfahrt (cit. on p. 40).

X.-Y. J. Wang and J. Yuko (May 2010). *Orion Active Thermal Control System Dynamic Modeling Using Simulink/MATLAB*. Tech. rep. NASA Glenn Research Center (cit. on p. 44).

M. Weiser (1981). “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, pp. 439–449 (cit. on p. 36).

— (1982). “Programmers Use Slices when Debugging”. In: *Communications of the ACM* 25.7, pp. 446–452 (cit. on p. 36).

M. W. Whalen, D. Hardin, and L. G. Wagner (2010). “Model Checking Information Flow”. In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, pp. 381–428. ISBN: 978-1-4419-1538-2. DOI: 10.1007/978-1-4419-1539-9_13 (cit. on pp. 35, 36, 42).

J. Wiik and P. Boström (2014). “Contract-Based Verification of MATLAB and Simulink Matrix-Manipulating Code”. In: *Lecture Notes in Computer Science*, pp. 396–412. DOI: 10.1007/978-3-319-11737-9_26 (cit. on p. 38).

S. Wolfram (1999). *The Mathematica Book*. 4th ed. Champaign, IL & New York: Wolfram Media & Cambridge University Press. ISBN: 1579550045 (cit. on pp. 73, 74, 165).

Y. Yang, Y. Jiang, M. Gu, and J. Sun (2016). "Verifying Simulink Stateflow Model: Timed Automata Approach". In: *31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, pp. 852–857. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970293 (cit. on pp. 41, 42, 85, 88, 90, 99, 112, 119, 161, 163).

J. Zander-Nowicka (2009). "Model-Based Testing of Real-Time Embedded Systems in the Automotive Domain". PhD thesis. Technische Universität Berlin. ISBN: 9783816779742 (cit. on p. 11).

L. Zou, N. Zhan, S. Wang, and M. Fränzle (2015). "Formal Verification of Simulink/Stateflow Diagrams". In: *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*. Ed. by B. Finkbeiner, G. Pu, and L. Zhang. Cham: Springer International Publishing, pp. 464–481. ISBN: 978-3-319-24953-7. DOI: 10.1007/978-3-319-24953-7_33 (cit. on pp. 39, 40).

R. Zurawski, ed. (2009). *Embedded Systems Handbook: Embedded Systems Design and Verification*. 2nd ed. Boca Raton: CRC Press. ISBN: 9781439807552 (cit. on p. 43).

Technical Documentation

P. Boström and L. Morel (2007). *Formal Definition of a Mode-Automata Like Architecture in Simulink/Stateflow* (cit. on p. 11).

D. Hedin and A. Sabelfeld (2012). *A Perspective on Information-Flow Control.* (Cit. on p. 33).

K. Marriott and P. J. Stuckey (2013). *A Minizinc Tutorial* (cit. on pp. 28, 78).

Modelica Association (Apr. 2017). *Modelica - A Unified Object-Oriented Language for Systems Modeling: Language Specification*. Version 3.4. Modelica Association. URL: https://www.modelica.org/documents/ModelicaSpec34.pdf (visited on 04/06/2018) (cit. on p. 18).

NICTA (2014). *The MiniZinc Constraint Programming Language*. http://www.minizinc.org/ (cit. on p. 78).

Prover Technologies AB (2019). *Formal Verification*. URL: https://www.prover.com/software-solutions-rail-control/formal-verification/ (cit. on p. 35).

C. Schulte, G. Tack, and M. Lagerkvist (2010). *Modeling and Programming with Gecode*. URL: http://www.gecode.org/doc/4.2.0/MPG.pdf (cit. on pp. 28, 78).

The MathWorks (Sept. 2017a). *Algebraic Loops*. r2017b. URL: https://de.mathworks.com/help/simulink/ug/algebraic-loops.html (cit. on p. 50).

— (Sept. 2017b). MATLAB. r2017b. URL: www.mathworks.com/products/matlab.html (cit. on p. 12).

— (Sept. 2017c). *Simulink Design Verifier*. r2017b. 1 Apple Hill Drive, Natick, MA. URL: www.mathworks.com/products/sldesignverifier (cit. on p. 39).

— (Sept. 2017d). *Stateflow*. r2017b. 1 Apple Hill Drive, Natick, MA. URL: www.mathworks.com/products/stateflow/ (cit. on pp. 15, 16, 161).

— (Sept. 2017e). *Types of Solvers*. r2017b. URL: https://de.mathworks.com/help/simulink/ug/types-of-solvers.html (cit. on p. 15).

— (2018a). *Block Libraries*. r2018b. URL: https://de.mathworks.com/help/simulink/block-libraries.html (cit. on p. 49).

— (2018b). *Execution of a Stateflow Chart*. r2018b. URL: https://de.mathworks.com/help/stateflow/ug/chart-during-actions.html (cit. on pp. 86, 161).

— (2019). *Execution Order for Parallel States*. r2018b. URL: https://de.mathworks.com/help/stateflow/ug/execution-order-for-parallel-states.html (cit. on p. 161).

The Modelica Association (Jan. 2019a). *Overview of Modelica Libraries*. URL: https://www.modelica.org/libraries/ModelicaLibrariesOverview (cit. on p. 49).

— (Jan. 2019b). *The* StateGraph *Library*. URL: https://www.maplesoft.com/documentation_center/online_manuals/modelica/Modelica_StateGraph.html (cit. on p. 19).

Wolfram Research (2008). *RSolve*. URL: https://reference.wolfram.com/language/ref/RSolve.html (cit. on p. 50).

— (2014). *Simplify*. URL: https://reference.wolfram.com/language/ref/Simplify (cit. on p. 167).

— (2018). *Wolfram Mathematica*. URL: www.wolfram.com/mathematica (visited on 02/15/2018) (cit. on p. 49).

— (2019a). *J/Link User Guide*. URL: http://www.wolfram.com/solutions/mathlink/jlink/ (cit. on pp. 167, 168).

— (2019b). *Some Notes on Internal Implementation*. URL: https://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html (cit. on pp. 73, 112).

A Examples

A.1 Information Flow Analysis of Signal-Flow-Oriented Models

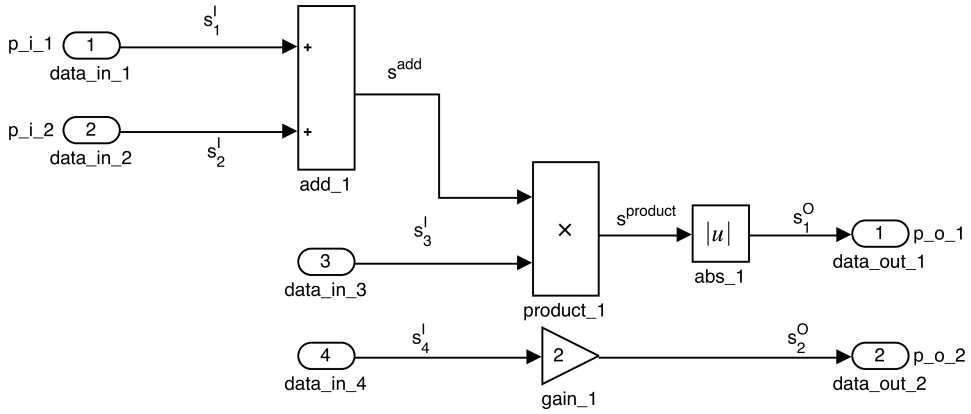
In this section, we utilize a number of generic model examples to demonstrate the functionality of our algorithm to analyze information flow in signal-flow-oriented software models.

A.1.1 Example 1: Unconditional Flow

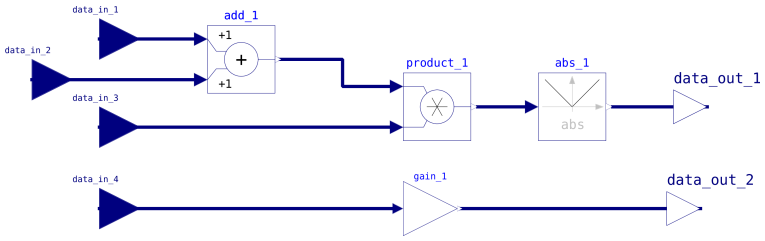
The example, shown in Figure A.1a as a Simulink implementation and in Figure A.1b as the equivalent Modelica implementation, uses a model consisting of four inputs and two outputs with arithmetic data manipulations connecting them. To demonstrate our algorithm, we assume that the model developer is interested in the existence of information flow between two pairs of inputs and outputs, namely (p_1^i, p_1^o) and (p_2^i, p_2^o). Starting with the first pair of interest (p_1^i, p_1^o), our algorithm performs the following steps:

Path Detection. After loading the model and translating it into our Java Intermediate Representation, the first step is the detection of path between the blocks of interest. If no paths can be found, information flow is impossible and all subsequent steps of our algorithm are superfluous. Starting from the block p_1^i, we perform a breadth-first search for possible paths leading the block p_1^O and detect the path:

$$\Phi_{(p_1^i \rightarrow p_1^o)} = \{\phi_1\}$$
$$\phi_1 = \langle p_1^i, b_{\text{add\_1}}, b_{\text{product\_1}}, b_{\text{abs\_1}}, p_1^o \rangle$$



(a) Simulink implementation



(b) Modelica implementation

Figure A.1: Example 1: Unconditional flow

Identifying Timing Dependencies. In this step, our algorithm analyzes the set of detected paths $\Phi_{(p_1^i \to p_1^o)}$ for time-dependent model elements splits the paths into *time slices* whenever it detects a block type that holds information between simulation steps to establish the precise timing dependency between the blocks along the path. On iterating over path ϕ_1, our algorithm detects that neither block is part of B^{mem} and therefore establishes a single time slice spanning the whole path between the blocks of interest. Note that here we denote a dependency between the output port of the first block in the path and the input port of the last block on the path:

$$\phi_0^{\text{ts}} = \langle p_1^i, b_{\text{add\_1}}, b_{\text{product\_1}}, b_{\text{abs\_1}}, p_1^o \rangle$$

Also, the matching untimed dependency between the first and last block on the path is established:

$$p_1^o \, \text{dep}_0^{\phi_1} \, p_1^i$$

The results of this step show that as time-dependent model elements are present on the path, information leaving the input block p_1^I enters the output block p_1^O in the same simulation step t.

Extracting Local Timed Path Conditions. After identifying the required set of time slices for the path under analysis, this step of our algorithm iterates over the path and extracts the conditions for information flow along the path from identified routing blocks and annotates them according to the time slice the respective routing blocks are placed into. When iterating over ϕ_1, our algorithm does not detect any blocks that control the information flow along the path. Information flow is therefore always assumed to be possible and the path condition controlling the information flow takes the form:

$$C\left(\Phi_{p_1^i \to p_1^o}\right) = true$$

After completing this step, our algorithm concludes that non-interference between the selected blocks cannot be proven as the information flow is unconditional. When analyzing the second pair of blocks of interest (p_2^I, p_2^O), however, our algorithm is able to prove non-interference between the blocks as it does not detect a path between them, i.e., $\Phi_{(p_2^I \to p_2^O)} = \emptyset$.

A.1.2 Example 2: Conditional Path Execution

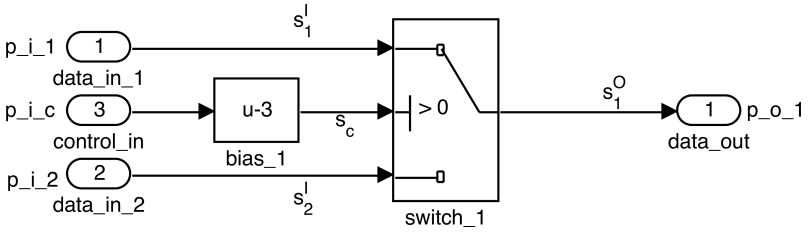


Figure A.2: Example 2: Conditional path execution

The example shown in Figure A.2 shows a MATLAB Simulink model consisting of a single routing block, three input blocks and a single output block. To demonstrate our algorithm, we analyze the information flow between the blocks (p^{i_2}, p^{o_1}).

Path Detection. Performing the path detection through the model, our algorithm detects the single path:

$$\Phi_{(p^{i_2} \to p^{o_1})} = \{\phi_1\}$$
$$\phi_1 = \langle p^{i_2}, b_{\text{switch\_1}}, p^{o_1} \rangle$$

Identifying Timing Dependencies. Similar to the previous example, our algorithm does not detect any time-dependent model elements along the path and therefore establishes the untimed dependency between the blocks of interest and single time slice ϕ_{ts} such that:

$$p^{o_1} \ \text{dep}_0^{\phi_1} \ p^{i_2}$$
$$\phi_0^{\text{ts}} = \langle p^{i_2}, b_{\text{switch\_1}}, p^{o_1} \rangle$$

Extracting Local Timed Path Conditions. On iterating over the detected path ϕ_1, our algorithm identifies the block $b_{\text{switch\_1}}$ as a routing blocks and proceeds to extract its parameters and consequently, the local path condition for information flow over ϕ_1. to detect the correct comparison operation, our algorithm first analyzes the port through which information flows into the routing block, which is port $p_{i,2}$. Hence, the extracted condition for information flow $>$ must be negated as the functionality a `Switch`-type block routes information from $p_{i,1}$ to the outgoing port p_o if the condition evaluates to true and to $p_{i,2}$ if it the condition does not hold. After extracting the comparison threshold 0, our algorithm therefore extracts the local path condition dependent on the block control signal input $p_{i,c}$ and annotates it with the timing information the routing block is sorted into:

$$c_0^{\phi}(b_{\text{switch\_1}}) = \neg\left(s_{c,0} > 0\right)$$

Evaluating Control Signals. In this step, our algorithm raises the scope of the identified routing control signals and path conditions from block-local to global. Using the identified control signal s_c as a starting point, we first use our path detection algorithm to extract all paths from the port to any global model inputs. This results in the set of control paths Φ^c with a single identified control path ϕ^{c_1}:

$$\Phi^c = \{\phi^{c_1}\} = \langle b_{\text{control\_in}}, b_{\text{bias\_1}}, s_c \rangle$$

```
var int: signal_out_input_3_t;
var int: signal_out_bias_1_t;
var int: signal_in_control_switch_1_t;

constraint signal_out_bias_1_t == signal_out_input_3_t - 3;
constraint signal_control_switch_1_t == signal_out_bias_1_t;
constraint not (signal_control_switch_1_t > 0);

solve satisfy;
```

Listing A.1: Translated constraint satisfaction problem for path ϕ_1 through example model 2

When raising the scope of the local path condition $c_{b_{\text{switch\_1}}}$ extracted in the previous step, our algorithm iterates over $\phi_{c,1}$ and identifies the signal manipulations along the path, which extracts the following functionalities for each block on the control path, annotated with the correct routing block timing information:

$$f_{\text{bias\_1}} = p_i - 3$$
$$p_{o,k}(b_{\text{bias\_1}}) = f_{\text{bias\_1}}(p_{i,k})$$
$$p_{o,k}(b_{\text{bias\_1}}) = p_{i,k} - 3$$

This information yields the global path condition for information flow through the control signal input of block $b_{\text{switch\_1}}$:

$$g_0^{\phi}(b_{\text{switch\_1}}) = \neg\big((p_0^{i_c} - 3) > 0\big)$$

and the global path condition for the execution of ϕ_1, which states that the information leaving the input block p_c^i at an arbitrary simulation step t must be greater than 3 for the switch to execute path ϕ_1:

$$C(\phi_1) = \neg\big((p_0^{i_c} - 3) > 0\big)$$

Translating and Solving Path Conditions. In the final step, our algorithm translates the extracted set of global path conditions into a constraint satisfaction problem and solves it using a constraint solver. During the translation process, our algorithm unfolds each path condition and defines decision variables for each global model input and required intermediate signal. The translated CSP is shown in Listing A.1.

When unfolding $C(\phi_1)$, our algorithm first defines a variable for the signal leaving the innermost block $p_0^{i,c}$, called `signal_out_input_3_t`, which encodes the outgoing direction of the signal, the block it originates from as well as the timing information. Subsequently, for each arithmetic operation in the global path conditions, we define a matching decision variable and add a constraint, such as the variable `signal_out_bias_1_t` and the constraint shown in Line 5. Finally, when the outermost function in the global path condition, i.e., the local path conditions, is reached, a variable for the switch control signal, here `signal_in_control_switch_1_t` is defined and the corresponding constraint is output, shown in Line 7. Additionally, a constraint connecting the switch control signal and the output of the final block on the path is added, as shown in Line 6.

Finally, we indicate the type of decision problem in Line 9 and instruct the solver to identify a solution to the constructed satisfaction problem.

A.1.3 Example 3: Complex Path Execution through Multiple Switches

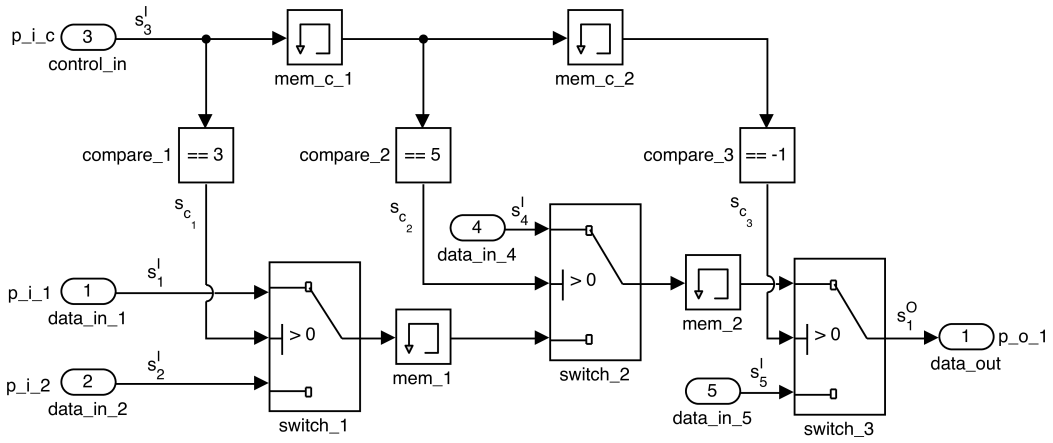


Figure A.3: Example model demonstrating the behavior of our algorithm in the presence of multiple switches and time slices

The example shown in Figure A.3 shows a MATLAB Simulink more complex model consisting of a set of routing blocks controlled by a single control signal. The control signal is manipulated through a number of `Compare` type blocks and held in `Memory` elements. To demonstrate our algorithm, we analyze the information flow between the blocks (p^{i_2}, p^{o_1}).

Path Detection. Performing the path detection through the model, our algorithm detects the single path:

$$\Phi_{(p_2^I \to p_1^O)} = \{\phi_1\}$$
$$\phi_1 = \langle p^{i_2}, b_{\text{switch\_1}}, b_{\text{mem\_1}}, b_{\text{switch\_2}}, b_{\text{mem\_2}}, b_{\text{switch\_3}}, p^{o_1} \rangle$$

Identifying Timing Dependencies. When iterating over ϕ_1, our algorithm detects a number of time-dependent model elements and therefore establishes the following time slices and dependency:

$$p^{o_1} \quad \text{dep}_2^{\phi_1} \quad p^{i_2}$$
$$\phi^{\text{ts}_0} = \langle p^{i_2}, b_{\text{switch\_1}} \rangle$$
$$\phi^{\text{ts}_1} = \langle b_{\text{mem\_1}}, b_{\text{switch\_2}} \rangle$$
$$\phi^{\text{ts}_2} = \langle b_{\text{mem\_2}}, b_{\text{switch\_3}}, p^{o_1} \rangle$$

The extracted *fixed delay* dependency illustrates that the information leaving through the output p^{o_1} has entered the system via the input p^{i_2} precisely two simulation steps prior.

Extracting Local Timed Path Conditions. On iterating over the detected path ϕ_1, our algorithm identifies the blocks $b_{\text{switch\_1}}$, $b_{\text{switch\_2}}$, and $b_{\text{switch\_3}}$ as routing blocks and proceeds to extract their parameters and timing information corresponding to the time slices they are placed into:

$$c_0^{\phi_1}\Big(b_{\text{switch\_1}}\Big) = \quad \big(s_0^{c_1} > 0\big)$$
$$c_1^{\phi_1}\Big(b_{\text{switch\_2}}\Big) = \neg\big(s_1^{c_2} > 0\big)$$
$$c_2^{\phi_1}\Big(b_{\text{switch\_3}}\Big) = \quad \big(s_2^{c_3} > 0\big)$$

Evaluating Control Signals. Using the identified control signal inputs s^{c_1}, s^{c_1} and s^{c_3} of the `Switch` blocks ($b_{\text{switch\_1}}$), ($b_{\text{switch\_2}}$) and ($b_{\text{switch\_3}}$), respectively, as starting points, our algorithm detects the control path sets $\Phi_c(b_{\text{switch\_1}})$, $\Phi_c(b_{\text{switch\_2}})$ and $\Phi_c(b_{\text{switch\_3}})$

with a single identified control path each:

$$\phi^{c_{\text{switch\_1}}} = \langle p^{i_3}, b_{\text{compare\_1}}, s^{c_1} \rangle$$
$$\phi^{c_{\text{switch\_2}}} = \langle p^{i_3}, b_{\text{mem\_c\_1}}, b_{\text{compare\_2}}, s^{c_2} \rangle$$
$$\phi^{c_{\text{switch\_3}}} = \langle p^{i_3}, b_{\text{mem\_c\_1}}, b_{\text{mem\_c\_2}}, b_{\text{compare\_3}}, s^{c_3} \rangle$$

To raise the scope of the local path conditions, our algorithm extracts the following functionalities for each block along the paths:

$$p_{o,k}(b_{\text{compare\_1}}) = p_{i,k}(b_{\text{compare\_1}}) == 3$$
$$p_{o,k}(b_{\text{compare\_2}}) = p_{i,k}(b_{\text{compare\_2}}) == 5$$
$$p_{o,k}(b_{\text{compare\_3}}) = p_{i,k}(b_{\text{compare\_3}}) == -1$$
$$p_{o,k}(b_{\text{mem\_c\_1}}) = p_{i,k-1}(b_{\text{mem\_c\_1}})$$
$$p_{o,k}(b_{\text{mem\_c\_2}}) = p_{i,k-1}(b_{\text{mem\_c\_2}})$$

Subsequently, our algorithm uses the extracted functionalities to construct a global path condition from each local path condition:

$$g_0^{\phi_1}(b_{\text{switch\_1}}) = \left(s_0^{i_3} == 3\right)$$
$$g_1^{\phi_1}(b_{\text{switch\_2}}) = \neg\left(s_0^{i_3} == 5\right)$$
$$g_2^{\phi_1}(b_{\text{switch\_3}}) = \left(s_0^{i_3} == -1\right)$$

Finally, the global path condition for the execution of ϕ_1 is constructed:

$$C(\phi_1) = \left(s_0^{i_3} == 3\right) \wedge \neg\left(s_0^{i_3} == 5\right) \wedge \left(s_0^{i_3} == -1\right)$$

Translating and Solving Path Conditions. In the final step, our algorithm translates the extracted set of global path conditions into a constraint satisfaction problem and solves it using a constraint solver. When instructed to solve the constructed CSP shown in Listing A.2, the constraint solver correctly recognizes the problem as unsatisfiable. Our algorithm is therefore able to prove non-interference between the blocks (p^{i_2}, p^{o_1}) as the path $\Phi_{(p^{i_2} \to p^{o_1})}$ can never be executed due to the time-dependent behavior of the routing blocks in the model under analysis.

```
%
var int: signal_out_input_3_t;
var int: signal_in_compare_1_t;
var int: signal_in_compare_2_t_plus_1;
var int: signal_in_compare_3_t_plus_2;
var int: signal_out_compare_1_t;
var int: signal_out_compare_2_t_plus_1;
var int: signal_out_compare_3_t_plus_2;
var int: signal_in_mem_c_1_t;
var int: signal_in_mem_c_2_t_plus_1;
var int: signal_out_mem_c_1_t_plus_1;
var int: signal_out_mem_c_2_t_plus_2;
var int: signal_in_control_switch_1_t;
var int: signal_in_control_switch_2_t_plus_1;
var int: signal_in_control_switch_3_t_plus_2;

%
function var int: compare(var int: x, var int: y) =
        if x == y then 1 else 0 endif;

%
constraint signal_in_control_switch_1_t > 0;
constraint signal_in_control_switch_1_t == signal_out_compare_1_t;
constraint signal_out_compare_1_t == compare(signal_in_compare_1_t, 3);
constraint signal_in_compare_1_t == signal_out_input_3_t;

%
constraint not (signal_in_control_switch_2_t_plus_1 > 0);
constraint signal_in_control_switch_2_t_plus_1 == signal_out_compare_2_t_plus_1;
constraint signal_out_compare_2_t_plus_1 == compare(signal_in_compare_2_t_plus_1,
↪  5);
constraint signal_in_compare_2_t_plus_1 == signal_out_mem_c_1_t_plus_1;
constraint signal_out_mem_c_1_t_plus_1 == signal_in_mem_c_1_t;
constraint signal_in_mem_c_1_t == signal_out_input_3_t;

%
constraint signal_in_control_switch_3_t_plus_2 > 0;
constraint signal_in_control_switch_3_t_plus_2 == signal_out_compare_3_t_plus_2;
constraint signal_out_compare_3_t_plus_2 == compare(signal_in_compare_3_t_plus_2,
↪  -1);
constraint signal_in_compare_3_t_plus_2 == signal_out_mem_c_2_t_plus_2;
constraint signal_out_mem_c_2_t_plus_2 == signal_in_mem_c_2_t_plus_1;
constraint signal_out_mem_c_1_t_plus_1 == signal_in_mem_c_2_t_plus_1;

solve satisfy;
```

Listing A.2: Translated constraint satisfaction problem for path ϕ_1 between blocks (p^{i_2}, p^{o_1}) through our third example model

B Stateflow Semantics

As discussed in Chapter 2, the semantics of state-machine-based controllers developed using Stateflow are only defined informally in the manuals provided in The MathWorks [2017d]. In this section, we present a formalization of the Stateflow semantics as presented in Jiang et al. [2016] and Yang et al. [2016].

While similar in syntax, the execution semantics of Stateflow and UPPAAL timed automata are inherently dissimilar. The key differences lie in the mode of execution and the structure of both types of automata: (1) In Stateflow, transitions are driven by events which are placed on an *event stack* in a deterministic sequential order, depending on explicit and implicit event specifications and the layout of the automaton itself [The MathWorks 2018b]. The same holds true for substates designed in a parallel fashion, which are sequentialized during evaluation of the Stateflow automaton according to either an explicit specification or, implicitly, to the geometry of the automaton [The MathWorks 2019]. UPPAAL timed automata, on the other hand, employ a non-deterministic, concurrent execution of networks of automata that synchronize via explicit messages over channels. (2) Stateflow supports the design of hierarchical states, which are activated and deactivated in a recursive fashion according to the validity of transitions entering and leaving states and their substates. A similar mechanism does not exist in UPPAAL, as during execution of a single timed automaton, there only exists a single active state.

In order to overcome this gap in the execution semantics, Jiang et al. [2016] and Yang et al. [2016] introduce an array-based data structure, which holds the possible events extracted from the Stateflow automaton, ordered according to the semantics of the Stateflow automaton, thereby acting as the event stack found in Stateflow. A *controller automaton* is responsible for pushing events onto this stack-like structure and, if valid, popping them to activate the appropriate timed automata via channels.

B.1 State Transformation

When encountering states during the transformation process, two situations can occur [Jiang et al. 2016]: (1) The state has no substates and no `entry`, `during`, or `exit` state actions are utilized in its design, or (2) the state utilizes decomposition into substates or state actions. In the first case, the translation simply creates a UPPAAL state and maps it onto the Stateflow state it originates from. For the second case, a number of timed automata are created, emulating the functionality of the Stateflow state and its interaction with the Stateflow automaton:

1. A *controller automaton*, simulating the activation and deactivation of substates of the current state,

2. an *action automaton*, emulating the functionality of state actions using three self-loops corresponding to the state action type,

3. a *condition automaton*, evaluating conditions of transitions between substates of the current state and storing the results of this evaluation in a data structure, and finally,

4. a *common automaton*, executing transitions between substates according to the results of the condition automaton.

B.2 Transition Transformation

After translation of the states in the originating Stateflow model, transitions are converted to the UPPAAL language and integrated into the translated states presented above. As described in Chapter 2, a transition can be annotated with four conditions and actions, of which each is optional: (1) an *event*, that specifies the transition to execute whenever the (2) *conditional action* is true; (3) a *conditional action* that is executed after the conditional action is evaluated as true; and (4) a *transitional action* that executes when the transition is taken and the target state is activated.

When encountering a Stateflow transition, these characteristics are embedded into the automata created by the state translation logic, such that:

1. each event is transformed into a unique integer to be placed onto the event stack;

2. each condition is translated to a guard at the corresponding transition in the UPPAAL *condition automaton*;

3. each conditional action is similarly translated to a condition action the corresponding transition in the UPPAAL *condition automaton*; and

4. each transitional action is transformed to a transition action at the corresponding transition of the *common automaton* of the state.

B.3 Simulation Time Representation

In UPPAAL, there is no direct equivalent to the global simulation time present in the Simulink model. While it is possible to represent this concept using *clocks* acting as transition guards, Jiang et al. [2016] and Yang et al. [2016] implement the passing of simulation using two global integer variables. These variables, `mDrivenTime` and `mTotalTime` represent the simulation time that has passed since entering a state and since starting the simulation, respectively. The first variable, `mDrivenTime` is reset by a global controller every time the simulation enters a new state is utilized to transition guards modeled using temporal logic, as shown in Figure 2.4. The second variable, `mTotalTime` is updated by the same global controller, i.e., incremented by 1, every time the event stack for the current simulation step has been completely processed.

Based on these translation rules, we are confident that the translation of Stateflow found embedded into combined signal-flow-based and state-machine-based controller models to the UPPAAL timed automata language is sound, as it provides a direct mapping of each Stateflow state and edge into a semantically equivalent timed automata representation. Further, it explicitly models the execution semantics of Stateflow, including the event queue, concurrent states and temporal logic conditions.

C Translating Discrete Control System Models to Mathematica

In this chapter, we present our translation of the signal-flow-oriented components of discrete embedded control system models to the CAS Mathematica. As presented in Section 5.6.3, we utilize this translation as part of our methodology to analyze information flow through signal-flow-oriented model components, specifically to identify global timed path conditions on cyclical control paths. The input to this translation process is a set of blocks representing the cyclical control path modifying the control flow condition of a single routing block under analysis.

As presented in Section 5.6.3, cyclical paths describe a recurrence relation [Bonchi et al. 2017b], i.e., a relation whose result at an arbitrary time depends on current as well as prior inputs. To identify non-recursive solutions to the recurrence relations described by cyclical control paths, we perform the following steps:

1. To extract the recurrence relation of a given control path, our method iterates backwards over the path while recording the functionality of each block until an input block or a previously recorded model element is encountered.

2. From this extracted control path representation, we contruct a Mathematica-compatible set of equations.

3. We utilize `RSolve` [Wolfram 1999, p. 96], a Mathematica functionality, to identify a non-recursive solution to the recurrence relation. If the CAS is able to identify a solution, the result of this operation is a pure, non-recursive function which enables us to calculate the value of the control signal at arbitrary simulation steps.

4. To identify the possibility of information flow in the presence of cyclical control paths, we utilize the `Reduce` functionality built into Mathematica [ibid., p. 86]. Given a function, in our case the output of the previous analysis step, a domain limiting the independent variable, and a set of conditions, it returns the range of function inputs in the given domain which fulfill the condition.

C.1 Representation of Signal-Flow-Oriented Block Functionalities in Mathematica

In the following we present our set of rules to translate Simulink blocks utilized in cyclical control paths into their respective Mathematica description. As discussed in Sections 4.3 and 5.6.3, our approach supports the utilization of unary and 2-ary block functionalities.

2-ary Arithmetical Block Types. Arithmetical block types using two inputs signals, such as the `Add` block type, are represented by the following functionality, as presented in Section 5.6.3:

$$\begin{aligned} p_{o,k} &= f(p_{i_1,k}, p_{i_2,k}) \\ f(p_{i_1,k}, p_{i_2,k}) &= p_{i_1,k} \bowtie p_{i_2,k} \qquad \text{with} \quad \bowtie \in \{+,-,\cdot\} \end{aligned}$$

The corresponding representation in Mathematica is as follows:

$$\mathtt{o} = \mathtt{i}_1 \bowtie \mathtt{i}_2$$

Stateful Model Elements. Stateful model elements, such as `Mem` blocks, are represented as follows:

$$\begin{aligned} p_{o,k} &= f_b(p_{i,k-l}), \quad \text{with} \ \ b \in B^{\text{Mem}} \\ f_b(p_{i,k}) &= p_{i,k-l} \end{aligned}$$

The representation of stateful modeling elements in Mathematica consists of two elements. A definition of the initial value, read from the properties of the modeling element, and a definition of the output signal depending on block state from a previous time slice, such as:

$$\begin{aligned} &\mathtt{mem[0]} == 0 \\ &\mathtt{mem[k]} == \mathtt{mem[k-l]} \end{aligned}$$

Input Block Types. Additionally, our translation supports a number of blocks without input signals, such as the `Const` block type or input blocks, such as:

$$p_{o,k} = r$$

Their Mathematica-compatible representation is as follows:

$$\mathtt{o} = \mathtt{r}$$

Based on these rules, our approach translates a given cyclical control path expressed using our intermediate representation into a Mathematica-compatible form, which we transmit to Mathematica. We achieve this connection between Mathematica and Java using the J/Link toolkit [Wolfram Research 2019a], which offers a scripting interface to Mathematica to be used directly from Java programs.

C.2 Identifying Non-Recursive Solutions to Cyclical Control Signals

Based on the Mathematica-compatible representation of the cyclical control path extracted and transmitted to Mathematica,in this step, we utilize Mathematica to identify a non-recursive solution to the recurrence relation represented by the path. To obtain such a pure-function solution to the recurrence relation, we use the `RSolve` functionality built into Mathematica.

Revisiting the example presented in Section 5.6.3, the extracted recurrence relation prepared for Mathematica takes the form shown below. Note that we apply the `Simplify` functionality [Wolfram Research 2014] to the extracted recurrence relation to obtain its minimal representation:

$$\mathtt{add[k]} == \mathtt{add[k-1]} - 1$$
$$\mathtt{add[0]} == 4$$

Based on this representation, we call the `RSolve` functionality according to the syntax described in Wolfram Research [2019a]. The first argument is the extracted equation set, which includes set of boundary conditions, i.e., the initial condition of the `Add` block used in our example. The second and third arguments describe the function for which a non-recursive solution is to be obtained, and the independent variable, respectively. For our example, the command takes the following form:

$$\texttt{RSolve}\Big[\,\{\texttt{a[k]} == \texttt{a[k-1]} - 1, \texttt{a[0]} == 4\,\}, \texttt{a[k]}, \texttt{k}\,\Big]$$

The output of this step is non-recursive representation of the functionality of the cyclical control path. Note that it is not guaranteed that a solution, even if existing, can be obtained automatically by Mathematica. We discuss an approach to identify the precise limitations of this solution process in Section 8.2.

For our example model, Mathematica returns the following solution:

$$\texttt{a[k]} \rightarrow 4 - \texttt{k}$$

The solution corresponds to the following pure function, which only depends on the current simulation step k:

$$p_{o,k}(add) = 4 - k$$

C.3 Calculating the Validity of Timed Path Conditions on Cyclical Control Paths

The pure-function representation identified in the previous step describes the precise value of the control signal at the control block at a given simulation step, possibly depending on the value of a connected input signal. In this final step of our approach, we combine this representation with the timed path condition corresponding to the control block under analysis by using the `Reduce` functionality built into Mathematica.

For the purpose of our approach, we use the `Reduce` functionality to identify those domains on the independent control signal function variable on which the timed path condition holds. We achieve this by calling the function as shown below:

$$\texttt{Reduce}\Big[\,\{\text{expr}\},\text{var},\text{dom}\,\Big]$$

The first argument *expr* combines the timed path condition corresponding to the current control block and path under analysis and the pure-function expression we have identified in the previous step. The second and third argument *var* define the independent variable over which we aim to identify a solution as well as its domain.

For our example model, the function call takes the following form:

$$\texttt{Reduce}\Big[\,\{4-\texttt{k}<0\ \&\&\ \texttt{k}>0\},\texttt{k},\texttt{k}\geq 0\ \&\&\ \texttt{Element}\big[\texttt{k},\texttt{Integers}\big]\,\Big]$$

There, the pure-function expression of the control signal $p_{o,k}(b_{\text{add}}) = 4-k$ is combined with the timed path condition $p_{o,k}(b_{\text{add}}) < 0$ and the limitation of the simulation step value to the domain of natural numbers, $k \in \mathbb{N}$ including 0.

The output of this function call describes the set of simulation steps for which the timed path condition on the control block under analysis holds. For our example model, the output $k > 4$ describes that the path condition only holds after four simulation steps have passed. Beginning with simulation step $k \geq 5$, input s_2^i is routed through the control block indefinitely.

C.4 Summary

In this chapter we have presented an integral part of our approach to analyze information flow through discrete embedded control systems models: the solution of cyclical control paths. As we have shown in Section 4.2, cyclical control paths through signal-flow-oriented components cannot be solved using static analysis techniques. To analyze information flow through model components containing such control paths, we have developed a translation and solution mechanism based on the computer algebra system Mathematica. Using the symbolic equation solution capabilities built into Mathematica, we obtain a non-cyclical, pure-function representation of the recurrence relation repre-

sented by the cyclical control path and, using this representation, are able to calculate the validity of the timed path condition of the path and control block under analysis. The results of this step are used as part of our overall methodology as described in Chapter 4.