Service-Oriented Design and Verification of Hybrid Systems

vorgelegt von
Timm Liebrenz,
Master of Science in Computer Engineering

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. Stephan Kreutzer Technische Universität Berlin
Gutachterin :	Prof. Dr. Sabine Glesner Technische Universität Berlin
Gutachter :	Prof. Dr. Holger Giese Hasso-Plattner-Institut
Gutachterin :	Prof. Dr. Paula Herber Westfälische Wilhelms-Universität Münster
Gutachter :	Prof. Dr. Holger Schlingloff Humboldt-Universität zu Berlin

Tag der wissenschaftlichen Aussprache: 5. Juli 2022

Berlin, 2023

Abstract

Nowadays, cyber-physical systems find application in many areas. These systems consist of multiple control components that interact with each other and with the physical environment. Model-driven development methods are used to handle their complexity and to ease their development. Furthermore, variability in the model design enables to customize a system for different environments. In safety-critical areas, where faulty behavior can injure or even kill people, it is desirable that the development and quality assurance of these systems is performed rigorously.

Cyber-physical systems have properties that make their development and quality assurance challenging. They contain hybrid behavior, which is the interaction of discrete changes and continuous behavior that is described by differential equations. Data flow-oriented modeling languages that are used in industry aim at enabling the comprehensible and fast development of hybrid control systems. However, they lack formal semantics, which precludes the application of rigorous verification methods and thus makes it impossible to provide guarantees about the behavior of the modeled systems. Formal modeling languages enable us to precisely model the mathematical processes and to formally verify that the system fulfills its requirements. However, correct behavior for each system variant must be ensured, which requires high effort, and formal approach often have scalability issues when handling industrially used models. Two major barriers prevent formal methods from being applied in practice: The lack of formal semantics of industrially used modeling languages, but also the high cost of rigorous formal verification.

In this thesis, we present an approach for the service-oriented design and verification of hybrid control systems. We define the concept of a *Service* for hybrid systems, with which we can formally define the functionality of the system and customize it for different contexts. As representative for data flow-oriented modeling languages, we present a formalization for Simulink into differential dynamic logic ($\mathsf{d}\mathcal{L}$), which is a formal language to design and verify hybrid systems. Our formalization enables the formal verification of hybrid Simulink models. With the verification results, we create *hybrid contracts* that describe the interface behavior of these services containing hybrid behavior. Additionally, we present an abstraction mechanism for services with hybrid contracts that enables the scalable verification of systems consisting of interacting services. With the addition of a feature model, we enable the customization for services for the reuse in different contexts. We have created a framework that implements our service-oriented design in Simulink and provides an automatic transformation of Simulink models into $\mathsf{d}\mathcal{L}$ to enable the service-oriented design and verification of Simulink models. The hierarchical nature of services enables the development of larger models that are verifiable.

Our approach combines the strengths of model-driven development of hybrid systems with the power of formally ensuring the correct behavior of modeled systems under all circumstances. We provide a formal foundation for hybrid Simulink models, for which we have developed an automatic transformation of Simulink models into $\mathsf{d}\mathcal{L}$. With our abstraction with hybrid contracts, we can provide safety guarantees for larger systems that consist of multiple interacting services. With a feature model and automatic service generation, we enable easy customization of services and their reuse. We demonstrate the applicability of our approach with different experimental results.

Zusammenfassung

In der heutigen Zeit finden cyber-physische Systeme in vielen Bereichen Anwendung. Diese bestehen aus Steuerelementen, die miteinander und mit der Umgebung interagieren. In der Praxis werden modellgetriebene Entwicklungsmethoden verwendet, um die Entwicklung zu vereinfachen. Durch das Bereitstellen von Variabilität im Design des Modells kann das System zur Verwendung in verschiedenen Umgebungen vorbereitet werden. In sicherheitskritischen Bereichen, bei welchen fehlerhaftes Verhalten zu Verletzungen oder Tod führen kann, ist ausgiebige Qualitätssicherung während der Entwicklung wichtig.

Die Eigenschaften von cyber-physischen Systemen erschweren deren Entwicklung und Qualitätssicherung. Sie weisen hybrides Verhalten auf, welches diskretes und kontinuierliches Verhalten kombiniert. Datenfluss-orientierte Sprachen, welche in der Industrie verwendet werden, ermöglichen das schnelle Erstellen von hybriden Steuersystemen mithilfe von nachvollziehbaren Modellen. Jedoch haben diese Sprachen keine formale Semantik und ermöglichen es nicht Garantien über das korrekte Systemverhalten zu liefern. Formale Modellierungssprachen ermöglichen es präzise die mathematischen Prozesse im Modell dazustellen und ermöglichen es formal zu prüfen, ob das modellierte System die Anforderungen erfüllt. Beim Bereitstellen von Variabilität muss zusätzlich für jede Variante korrektes Verhalten sichergestellt werden. Zusätzlich skalieren formale Ansätze schwer für Modellgrößen, die in der Praxis relevant sind. Die folgenden Barrieren behindern die Anwendung von Formalen Methoden in der Praxis: Die fehlende formale Semantik für in der Industrie verwendete Modellierungssprachen und hohe Kosten der formalen Verifikation.

In dieser Dissertation stellen wir einen Ansatz zur Service-Orientierten Entwicklung und Verifikation für hybride Steuerungssystem vor. Wir definieren den Begriff eines *Service* für Hybride Systeme, der es ermöglicht formal die Funktionalität von Komponenten zu erfassen und an verschiedene Kontexte anzupassen. Wir nehmen beispielhaft die datenflussorientierte Modellierungssprache Simulink und präsentieren eine Formalisierung in der formalen Semantik von Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$). Wir erstellen *hybrid contracts*, welche formal das Interfaceverhalten der Services erfassen. Zudem präsentieren wir einen Service-Abstraktionsmechanismus, der es ermöglicht Eigenschaften von größeren Systemen mit integrierten Services zu verifizieren. Außerdem beinhaltet unsere Definition von Services ein Feature Modell, das es ermöglicht Teile des Services zu modifizieren, um diesen für andere Anwendungsfälle anzupassen. Wir haben unsere Konzepte in einem Ansatz für Simulink implementiert und ermöglichen damit die Service-Orientierte Entwicklung und Verifikation für Simulink. Die hierarchische Natur von unseren Services ermöglicht die Entwicklung und Verifikation von größeren Systemen, die aus interagierenden Services bestehen und die wiederum formal verifiziert werden können.

Unser Ansatz kombiniert Stärken modellgetriebener Entwicklung von hybriden Systemen mit den formalen Garantien, die korrektes Systemverhalten unter allen Umständen sicherstellen. Wir stellen eine formale Basis für hybride Simulink Modelle bereit, für welche wir eine automatische Transformation von Simulink nach $\mathsf{d}\mathcal{L}$ entwickelt haben. Wir ermöglichen Sicherheitsgarantien mithilfe unserer Abstraktion über *hybrid contracts* für größere Systeme, die aus interagierenden Services bestehen. Über ein Feature Modell und einer automatischen Erstellung von Services, ermöglichen wir die Anpassung von Services für andere Umgebungen und erhöhen ihre Wiederverwendbarkeit. Wir demonstrieren die Anwendbarkeit unseres Ansatzes mithilfe von verschiedenen experimentellen Ergebnissen.

Contents

1 Introduction

In current times, the prevalence of cyber-physical systems (CPS) [BG11] is ever increasing. One important property of these systems is the strong interconnections between the digital components and the physical environment. Such systems find application in many areas, like the automotive industry or the medical context, and they fulfill a wide range of tasks. In safety critical applications, faulty behavior can cause injury and threaten human lives. Therefore, their correct behavior under all circumstances has to be ensured. However, the special properties of CPS make it difficult to ensure a smooth development process and still ensure their correct behavior.

First, CPS have a strong connection to the physical environment. Therefore, during the development it is desirable to consider the interplay of the digital components and the physical environment. As a result, cyber-physical systems contain hybrid behavior, i.e., they comprise continuous behavior, which is described by differential equations, additionally to traditional discrete behavior, like the change between control states. Second, hybrid control systems often consist of interacting components that work together and are executed in parallel. These properties impede the development of CPS with a traditional implementation in a sequential programming language. Model-driven development enables us to model a system on a higher abstraction level and provide better comprehensibility than written code. Furthermore, the executable code is often created by automatic code generation. To reduce the development costs and shorten the development time, parts of existing models or even whole models can be reused in the development of new systems. At the same time, model-driven development languages enable early system validation by simulation in early development steps. While this allows for testing a design for given input scenarios, the results are only applicable for a limited number of input scenarios. The continuous evolving values in the behavior of the modeled hybrid systems produce continuous trajectories and an infinite state space. While systems modeled in industrial used languages, like MATLAB Simulink [Matb], allow for fast development and reuse of developed parts, they are limited in their capabilities to provide guarantees about the system behavior. Formal verification can provide guarantees for all possible inputs. However, formal verification of hybrid control systems is impeded by the problem that many industrially used languages do not come with formal semantics. The languages that are typically used in model-driven development do not provide formal models and the creation of formal models requires high expertise. To ensure correct behavior of a CPS, the interaction between components and the environment must be considered. Changes in a part of the system often require performing the verification process again and a manual system-wide verification is a time intensive task.

This thesis addresses the problem that there is a gap between the commonly used modeling tools and the verification techniques used in hybrid system design. This problem consists of four major parts. First, the modeling tools do not provide formal models for systems with hybrid behavior. Second, the correct behavior of the system is often determined by the behavior of multiple interacting components and the behavior of the environment. This also means, components that are present multiple times in one system are considered multiple times in the verifica-

tion. Third, changes in one part of the system require a new verification for the whole system. Fourth, knowledge about verification results could help in the design and verification of new systems. In the context of hybrid control systems, these problems concern components that are executed in parallel and contain hybrid behavior, which consists of discrete changes and continuous evolutions.

To address these problems in this thesis, we aim at developing an approach that integrates formal verification in model-driven development and enables systematic reuse for hybrid systems. We require our approach to fulfill the following criteria:

1. *Hybrid behavior*
   In the design of hybrid control systems, the resulting system has elements that contain discrete behavior as well as continuous behavior. Therefore, we require our approach to support system models that contain hybrid behavior.

2. *Reuse capabilities and variability*
   We aim at providing components that can be reused in system design. These components should not only consist of exact copies but contain variability by providing means to change them in a structured manner. This enables more possibilities to reuse given components.

3. *Formal foundation*
   Many industrially used system design languages are only informally defined. This impedes formal verification. We require our approach to provide a formal foundation that enables us to integrate formal verification into existing industrial design processes.

4. *Compositionality*
   CPS can consist of multiple interacting components, which also can consist of other components. We aim at exploiting this compositional and hierarchical character of systems and providing means for compositional design and verification.

5. *Automation*
   Formal verification is a time-consuming task. This starts with the creation of formal models. Domain experts in the modeling area of hybrid systems are often not simultaneously experts in formal modeling, which creates a gap between system model and formal model. To increase the applicability of our approach, we intend to provide an automatic transformation of an informal model into a formal representation. Furthermore, the verification should be as automated as possible.

6. *Practical Applicability*
   Hybrid control systems are utilized in many different domains. We aim at supporting models that can be used in different application domains. To show the practical usability of our approach, we apply our approach to case studies from different domains. Namely, we use our approach with a temperature control system, a generic infusion pump, a distance warner that is provided by an automotive partner, and a robot in a factory.

We propose a service-oriented design and verification approach for the industrially widely used and powerful modeling language Simulink to solve the aforementioned problems. One of the main ideas of our approach is to define reusable services that fulfill contracts and incorporate it into model-driven development to tackle the problem of reusability of components and system verification. At its core, we define a service notation for Simulink that encapsulates customizable functionality, interface behavior, and verification results. We develop our own service representation in model-driven development that conforms with the ideas of services. This enables us to bridge the gap between design and verification by providing reusable components, which contain contracts that enable the reuse of verification results. Our approach is based on three key ideas.

First, we present an automatic transformation from Simulink models into the formal semantics of differential dynamic logic ($\mathsf{d}\mathcal{L}$). Our transformation is not limited to Simulink blocks that have discrete behavior, but it is also able to transform blocks that describe continuous behavior. Our transformation creates a $\mathsf{d}\mathcal{L}$ model that precisely captures the hybrid behavior of the given Simulink model including continuous trajectories and discrete changes. This provides us with a formal foundation for models of hybrid control systems. Furthermore, the use of $\mathsf{d}\mathcal{L}$ as formal representation enables us to use the interactive theorem prover KeYmaera X [Ful+15] to verify system properties semi-automatically.

Second, we define Simulink *services*, which provide means to reuse and customize components in system design. A service may comprise discrete as well as continuous behavior, extends a component by structural variability and by a defined interface. We enhance services with a *feature model* to enable systematic structuring of variants. A feature model is a tree structure that we use to model possible changes in the internal service structure and their dependencies. With adaption of feature modeling to services, we provide means to adjust the behavior of a service in a well-structured and well-defined way. In particular, feature modeling enables the addition and deletion of functionality. Users can access a service via a defined interface to obtain results for their own uses. A large number of different services can be provided in service-libraries. The main benefit of our services is that they are reusable in different areas.

Third, we provide the concept of *hybrid contracts* that capture the dynamic interface behavior of services. They consider discrete changes of values as well as continuous evolutions to capture the hybrid nature of the underlying service. With that, they are not limited to static properties, but also describe the dynamic changes and the evolution of values. A hybrid contract abstracts the inner structure of a service by capturing its hybrid behavior of its interface and can be used in compositional system verification.

In summary, we present an approach to integrate verification and reuse in the design of hybrid control systems modeled in Simulink. Our main contributions are as follows:

1. With our automatic transformation from Simulink models into $\mathsf{d}\mathcal{L}$, we provide a formal foundation for hybrid Simulink models and enable their verification with the interactive theorem prover KeYmaera X. This enables the design of hybrid systems in the commonly used modeling language Simulink and

still enables formal verification without the need for the designer to provide a formal model.

2. We introduce *hybrid contracts* that capture the dynamic behavior of the interface of a service. This enables us to describe the hybrid behavior of a service in a formal way and can be used for compositional verification by abstracting from the inner block structure of a service. This verification is more scalable than a monolithic verification. Our compositional approach is able to reduce the verification effort significantly and enables the verification of more complex hybrid systems that contain interacting services.

3. Our *services* for Simulink encapsulate defined functionality, interface behavior and customization capabilities. By integrating a *feature model* into the service design, we provide a systematic way to customize a service and still enable the reuse of interface and verification information. We show that the use of our service-oriented design reduces the development effort of systems that contain variable components.

4. We demonstrate the applicability of our approach with four case studies. Namely, a temperature control system, a general infusion pump, a distance warner model provided by an industrial partner, and an autonomous robot in a factory setting. The interactive verification for our case studies is performed with a reasonable amount of human effort and calculation time.

We have published our work as follows: An overview of our approach is presented in [Lie18], where we motivate the use of services to enable reuse in Simulink and sketch our idea of a formal foundation for compositional verification. In [Lie+17], we introduce our service-oriented design in Simulink by presenting a methodology to reuse component information and encapsulate variability into the service model. In [LHG18], we present our automatic transformation of Simulink models into $\mathsf{d}\mathcal{L}$. We introduce transformation rules that capture the behavior of Simulink blocks in $\mathsf{d}\mathcal{L}$. We demonstrate the use of the interactive theorem prover KeYmaera X. This transformation is the foundation of our formalization of Simulink models in $\mathsf{d}\mathcal{L}$ and enables us to formally define properties that hold for Simulink models. In [LHG19], we present our hybrid contracts for compositional verification. We use our hybrid contracts and abstraction technique to prove properties of systems that consist of multiple interacting services. We have discussed what we have learned about the formalization of informally defined languages with automated transformations in [HLA21]. In [LHG21], we present our feature modeling to generate customizable Simulink services. In [LHG20], we evaluate the manual verification effort during the verification of services with the case study of a generic infusion pump. Furthermore, we present techniques to reduce the manual input during verification by providing means for automatic invariant generation and automatic creation of assumption and guarantees for hybrid contracts. In [ALH21, HAL21], we present the application of our approach for a case study of an autonomous factory robot. We use our hybrid contracts to prove that no collisions occur, despite that the system contains a black-box component for which we do not know the inner structure. With a global system guarantee, we infer hybrid contracts that need to hold for the black box component.

The rest of this thesis is structured as follows: In Chapter 2, we introduce necessary background of this thesis. We discuss related work in Chapter 3. Chapter 4 provides an overview of our approach. In Chapter 5, we introduce our transformation of Simulink models into dℒ and perform the verification of system properties for a temperature control service. To enable the verification of interacting systems, we extend our transformation by hybrid contracts that are introduced in Chapter 6. In Chapter 7, we present our services that encapsulate the obtained verification results and enable their reuse in system development by providing variability with feature modeling. In Chapter 8, we evaluate our approach with four different case studies. Chapter 9 concludes our work.

2 Background

In this chapter, we introduce the preliminaries of this thesis. First, we introduce the principles of *model-driven development* (MDD). Furthermore, as part of MDD, we introduce the concepts of *feature modeling*. Second, we introduce *hybrid systems* and the challenges that occur due to their behavior. To do this, we briefly introduce representations that capture the behavior of hybrid systems in a formal way and the basic ideas of existing analysis techniques for hybrid systems. Third, we briefly introduce general verification approaches. Afterwards, we present the modeling languages that are used in the remainder of this thesis. We introduce MATLAB/Simulink [Matb] as a development language for hybrid systems. Furthermore, we present a Simulink model of a temperature control system that we use as a running example in the remainder of this thesis. As a formal language for the representation of hybrid systems, we introduce differential dynamic logic (d$\mathcal{L}$) [Pla08] in detail. Lastly, we briefly discuss the concepts of *Service-Oriented-Architectures* as design paradigm.

2.1 Model-Driven Development

The development of cyber-physical systems requires to consider interacting components and their interactions to the continuous environment. This is difficult to implement in traditional sequential code. To cope with the complexity of these systems, model-driven development (MDD) [Sel03] is used.

In MDD, models are used as central artifacts in the development process. In this development technique, a model is created instead of directly developing the executable code of these systems out of their specifications. MDD increases the comprehensibility of the developed system and reduces the development effort. Furthermore, a system designer can analyze and often even simulate the intended behavior of the system and check its behavior and the interplay between the system behavior and its environment. In the following, we give a brief introduction of the concept of models, MDD and feature modeling.

Models

A model in general is a depiction with basic components of a real world object, system or process. Since a model is only an abstraction of the real world and in general cannot contain all details of the real world, the accuracy of models is also limited. There are different kinds of models that can be used: static models and dynamic models. Static models depict the components of a system and their possible interactions and interfaces. These models are independent of time. Dynamic models represent the evolution of the system over time. It captures the changes that occur in the system states during the execution of the system. Some models only require a predecessor-successor relation of system events. Other systems require a distinct representation of time. These models often allow for the simulation of the system behavior by calculating the system behavior at specific times.

In [Sel03], five characteristics to measure the quality of a model are suggested:

1. Abstraction. A model is an abstraction of real world behavior. Therefore, a model does not represent the real world in every detail but captures the behavior that is important for the system.
2. Understandability. The model should convey the underlying behavior in an intuitive way.
3. Accuracy. A good model represents the behavior that is observed in real life. Often, the accuracy is limited since the real world is too complex or there can occur unknown events.
4. Predictiveness. Models should allow to calculate the behavior of the system, which can be used to generate relevant data. This can be either through experiments and simulation, or with formal analysis of the model.
5. Inexpensiveness. The more real world details are added to the model the higher is the quality of the results that can be obtained by the analysis of the model. At the same time, the construction and analysis of the model gets more complicated with higher quality of the included real world details. The construction and analysis of a model should be cheaper than the implementation and testing of the system.

The basic components to model systems are adapted to the domain that should be modeled. By including more real world behavior in a model, it is more accurate, but at the same time more components and bigger models are less comprehensible and impede an analysis of their behavior. Therefore, for the creation of models there is a trade-off of comprehensibility and accuracy. In general, models can be created in many different ways. Mathematical equations can be used to accurately describe the underlying physical processes. In computer science and MDD in general, models are often created with visual elements.

Model-driven development

Figure 2.1 shows a possible workflow of system development with MDD. The basic idea of MDD is that the developer first creates a model that captures the behavior of the system instead of directly writing executable code. Furthermore, a model of the environment, which interacts with the system, is included in the system model. Therefore, MDD introduces an intermediate step between the specification of a system and its implementation in the real world. The model can be used to analyze the interactions between the environment and the system or even between different components of the system. This also enables the design of components in different levels of abstraction. Some MDD approaches can differ from this approach, e.g., the step of code generation or hardware synthesis can be omitted.

The use of MDD has benefits over the direct development of code.

- Comprehensibility. By providing basic components to model behavior and using design patterns, the comprehensibility of the model is increased and the design is simplified. Models and their building blocks are adapted for the domain that should be modeled. Therefore, the resulting models illustrate the domain and can be read by domain-experts.

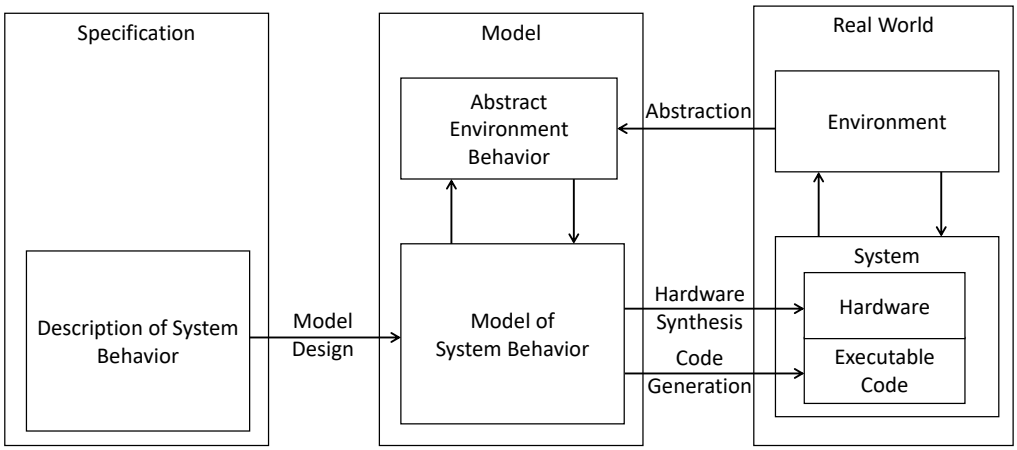


Figure 2.1: Model-Driven Development Flow

- Reuse. Standardized models allow for reuse of model parts in new systems and provide compatibility to other models.

- Simple representation of complex tasks. The models often represent parallel tasks, which can be designed via visual representations instead of sequential code. Furthermore, models can abstract from the target architecture (e.g., memory) and allow a designer to only include relevant behavior of the developed system.

- System analysis or simulation. Means to simulate or even formally analyze a system allow to check whether the designed system fulfills its specification. Finding possible faults before the system is implemented reduces the costs for error correction dramatically.

- Code generation and hardware synthesis. By providing means for automatic code generation and the synthesis of hardware components, no manual effort in the creation of executable code or hardware development is necessary.

For all its benefits, it should be noted that MDD also comes with some challenges.

- Adequate abstraction. The model can only capture a chosen part of the real world behavior. If the abstraction is too coarse, the accuracy of the model is limited and it can exhibit behavior that does not occur in the real world. On the other hand, a detailed model is difficult to create and can impede the analysis of the system behavior.

- Tool support. To enable efficient analysis capabilities, tools are required that explore the behavior of the modeled systems. This is challenging, especially in the context of hybrid systems, where the interactions between discrete and continuous behavior must be considered.

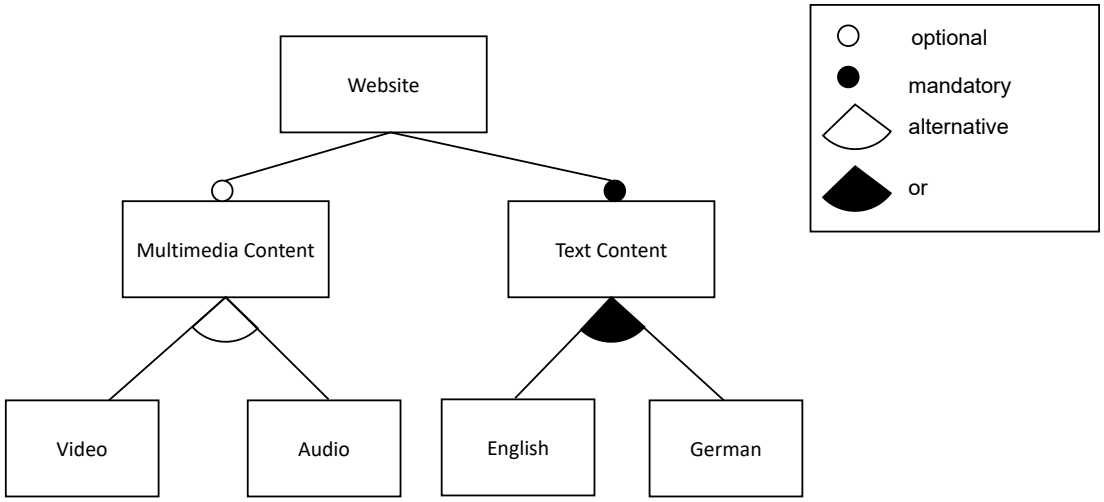


Figure 2.2: Example Feature Model for a Website

Feature Modeling

In MDD, often a large number of models is created. Some of them are variants of other existing models. To ease the development of new variants and enable a designer to choose between existing models, feature modeling [Kan+90] can be used to capture the relations between different model variants.

Feature modeling [Kan+90] is a concept to illustrate different variants of one system in a hierarchical model. It is a medium between a user and a product provider. In Software Product Lines (SPL) a feature model allows an end-user to choose between given properties that a product should fulfill.

Definition 2.1 (Feature model)**.** A *feature model* is a tree structure

$$FM = (F, D)$$

where F is a set of features that depict design alternatives and D is a set of edges that determine the dependencies between features. A feature is an end-user visible characteristic of a system. By choosing a set of features, a configuration for the final product can be determined.

The combinations and dependencies of features can be used to create different variants of a design. A variant is an instance of a design where different features are activated or deactivated according to the edge dependencies. In general, a child feature can only be chosen if its parent feature is also active. The relations determine further conditions when child features must be active. We consider the following four dependency types: *mandatory*, *optional*, *alternative*, and *or* relations. A *mandatory* dependency means that the child feature must be present in a variant if the parent feature is active. In an *optional* dependency, the child feature can be active or not. An *alternative* dependency is between one parent and multiple child features. It states that exactly one child is active and all the others must be deactivated. An *or* dependency also considers multiple child nodes and states that at least one of the child nodes must be active but is not restricted to only one.

Figure 2.2 illustrates a feature model for the design of a website. In this example the root feature *website* describes the product that is designed in this model. In this

example, the website must contain *text content*. This text content can be present in different languages that are present as its child features. The two languages *English* and *German* are in an *or* relation, which means that the text content must be present in at least one of these two languages or in both. The website in figure 2.2 can have optional *Multimedia content*. In this example, the multimedia content is defined more precise in its child features. These are in an *alternative* relationship. In this case, the website can have either video or audio content, or neither since the multimedia content is optional.

2.2 Hybrid Systems

In this thesis, we consider systems that combine discrete and continuous behavior. Only if we include the continuous behavior of physical processes in the system behavior, we can obtain a refined model of the overall system behavior. By introducing continuous behavior into the discrete behavior of digital systems, we obtain *hybrid systems* [Gro+93], which combine both behavior types in one system. The discrete behavior of these systems is described by *jumps* in the system behavior. These jumps can be caused for example, by the change between different operating modes of a controller, or operations that are executed at specified time steps. They can be represented by sudden changes in the values of system variables. Continuous behavior is described by the *flow* of the system. The flow contains all changes of the system that evolve over time. Typically, differential equations are used to model the flows in a hybrid system. The resulting flows create continuous trajectories.

There are different methodologies to model hybrid systems. To give an intuitive understanding of hybrid systems, we first introduce hybrid automata [Hen00] in this section as general introduction to hybrid systems and use them to create a simple model of a bouncing ball. Afterwards, we introduce differential dynamic logic (Section 2.5) as a more advanced formal representation for hybrid systems.

Hybrid automata are finite state machines that describe the evolutions of continuous variables and discrete state changes. We use the following definition of a hybrid automaton, which is a slight variation of the definition given in [Hen00].

Definition 2.2 (Hybrid Automaton)**.** A hybrid automaton H is defined as tuple:

$$H = \{X, V, E, init, inv, flow, jump\}$$

X is a set of real-valued variables. Additionally, we define two sets of variables $\dot{X}, X'$. These contain variable symbols that represent discrete and continuous changes. The set $\dot{X} = \{\dot{x}_1, ..., \dot{x}_n\}$ with $\{x_1, ..., x_n\} \subseteq X$ denotes values after occurrences of discrete changes. $X' = \{x'_1, ..., x'_m\}$ with $\{x_1, ..., x_m\} \subseteq X$ denotes continuous changes of values, i.e., the symbols define the derivatives of the variables. V are control modes and E are control switches. Together they form a finite directed multigraph (V, E). The control modes are labeled with three vertex labeling functions $init, inv, flow$, which assign to each control mode $v \in V$ three predicates. Each initial condition $init(v)$ assigns values to variables in X. Each invariant condition $inv(v)$ is a predicate over variables in X. It holds as long as the automaton remains in the given control mode. Each flow condition $flow(v)$ over

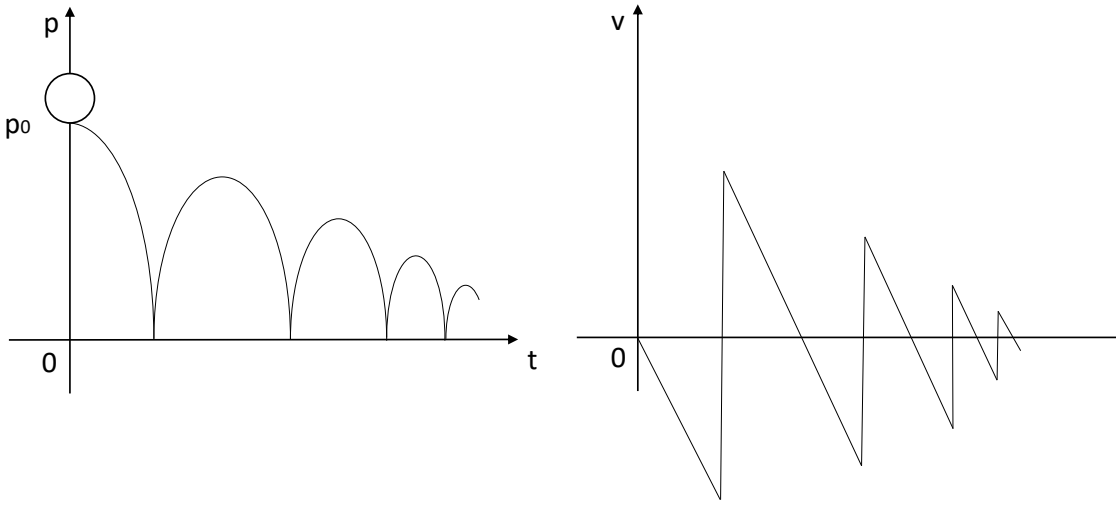


Figure 2.3: Bouncing Ball

variable symbols in X' describes the continuous change of the according variables in X. The edges between vertices are labeled with an edge labeling function. Each jump condition $jump(e)$ provides jump conditions on variables in X that must be fulfilled in order that the respective jump can be taken and defines the resulting values of variables after the jump is taken by assigning values to variable symbols in $\dot{X}$.

The key idea of hybrid automata to model hybrid behavior is to link each discrete system state to differential equations that describe how the continuous variables of the system evolve in this state. The system can be in one of its states, during which all differential equations are executed in parallel.

Bouncing ball

In the following, we model a bouncing ball as hybrid system. The ball starts at a height p_0 and falls down. Whenever it hits the floor, it jumps back up with a dampening factor of f. The change of the position and speed of the ball are values that are determined by the physical behavior of the ball and its environment. Additionally, the system contains sudden changes during each bounce when the ball changes its direction. In the following, we present a one-dimensional model of a bouncing ball, where p is its position above the ground, v is its velocity, and a is a constant acceleration, which is equal to the negative gravitation constant g.

Figure 2.3 represents the evolution of the position p of the ball and its velocity v over time t. In this example, we only consider the height above the ground to describe its position, i.e., its movement is one-dimensional. Initially, the position of the ball is at p_0 and its velocity is 0. The position and velocity change over time and can be described with differential equations:

$$p'(t) = v(t)$$
$$v'(t) = a(t)$$

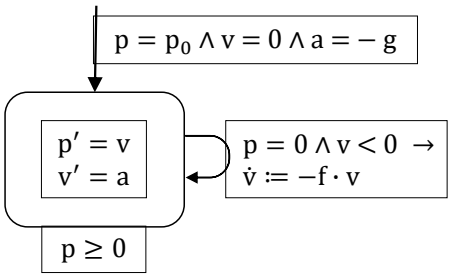


Figure 2.4: Hybrid Automaton of a Bouncing Ball

The acceleration is a constant that is given by the gravitational acceleration g. Whenever the ball hits the ground at $p = 0$, it bounces and inverts its velocity. At each bounce, the ball loses some of its speed. The loss is given by a dampening factor f.

Figure 2.4 shows a bouncing ball that is modeled as a hybrid automaton. The automaton consists of one state and one transition. During the evolution of the state, the position and velocity change according to the given differential equations.

In the case that the value of p falls below 0, the system changes in the state of the increasing p value.

2.3 Verification

In safety-critical systems, a failure can cause high financial damage and even harm human lives. It is crucial to ensure that such systems behave according to defined specifications under all circumstances. In the context of MDD, formal analysis techniques can be used to ensure the correctness of the system model. In this section, we give a brief introduction to the formal verification of safety-critical systems.

First, we briefly introduce the general idea of *Model-Checking* and the extensions to hybrid automata and the resulting challenges. These approaches suffer from a so-called state-space explosion problem. Therefore, we look at methods that aim at coping with this problem. Namely, the relevant methods are *deductive verification* where system properties are inferred from the system specifications and *contract-based verification* where the interactions between system components are abstracted to reduce the possible state-space.

Model checking

One of the most famous approaches for formal verification is model checking [Cla+99]. The key idea of model checking is to check whether a model of a finite state system fulfills a given requirement specification that is given as logical formula. The exploration of the state-space is done fully-automatically by a model checker, however the state-space explosion is a major problem for interacting con-

current systems.

A model, which is used in model checking, can be represented by a Kripke Structure:

Definition 2.3 (Kripke Structure)**.** A *Kripke structure* over a set of atomic propositions AP is a tuple

$$M = \{S, s_0, R, L\}$$

where S is a set of states, with $s_0 \in S$ as initial state. $R : S \times S$ is a transition relation. $L : S \to 2^{AP}$ is a labeling function.

The idea of model checking is to check whether a model fulfills a given requirement specification:

$$M \models \Phi$$

The model M is a Kripke structure and Φ is the requirement specification, which is given as temporal logic formula. The general approach is not directly applicable for the verification of hybrid systems since no continuous time and no continuous changes of values are considered.

In the verification of hybrid systems, the possible state-space does not only consist of discrete state changes but also consists of continuous evolutions. The continuous evolutions define the change of real numbers where there are infinite values between two different numbers. The verification is further impeded since the verification of general hybrid automata, i.e., systems that incorporate continuous evolutions, is an undecidable problem [Hen+98].

Deductive Verification

Instead of constructing the whole state-space to check whether a system property is fulfilled, *deductive verification* is based on axioms and proof rules that define how each step changes system wide properties [Hoa69]. Axioms are the foundation of deductive reasoning and are chosen according to the underlying behavior of the variables in the system. In the following, we use illustrating examples to introduce the basics of deductive verification. First, we showcase a simple calculus for non-negative integers. Second, we have a brief look at the Hoare calculus.

Table 2.1 shows an illustrating example for a simple calculus with a small set of axioms relevant to non-negative integers with addition and multiplication. Axioms are used to construct logical proofs to show that a given theorem holds.

For programs, the rules of logical deduction are extended to executable code. In the following, we look at the Hoare calculus as an example for logical deduction. We introduce an excerpt of the functionality to give a brief understanding for the use of logical deduction. The intended function of a program or part of it can be specified by making assertions of the values of relevant variables after the execution of the program. These assertions are often general properties or relations between variables and not concrete values. The proof for program behavior begins with a set of assumptions that define the initial behavior of the variables. The connection

Table 2.1: Showcase Axioms [Hoa69]

A1	$x + y = y + x$	addition is commutative
A2	$x \cdot y = y \cdot x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	multiplication is associative
A5	$x \cdot (y + z) = x \cdot y + x \cdot z$	multiplication distributes through addition
A6	$(x - y) + y = x$ for $y \leq x$	addition cancels substraction
A7	$x + 0 = x$	neutral element of addition
A8	$x \cdot 0 = 0$	multiplication with zero
A9	$x \cdot 1 = x$	neutral element of multiplication

between preconditions P, a program Q and a description of its result R is written as a Hoare triple:

$$\{P\}Q\{R\}$$

The Hoare triple describes that any execution of program Q that starts in a state where P holds, terminates in a state where R will hold. The axioms describe how the execution of line of code changes the program state that consists of the values of all variables.

Rules of consequence. Deductive verification applies inference rules to axioms and existing theorems to enable the deduction of new theorems. Inference rules describe that if assertions of the form X and Y have already been proven as theorems, then Z is thereby proven as a theorem. A simple example of an inference rule states that if the execution of a program Q ensures the truth of the assertion R, then it also ensures the truth of every assertion which logically implies P. Formally we write the rules of consequence as:

$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash R \implies S \text{ then } \vdash \{P\}Q\{S\}$$
$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash S \implies P \text{ then } \vdash \{S\}Q\{R\}$$

Rule of composition. Programs consist of program lines that are executed one after the other, which can be written as $Q_1; Q_2; ...; Q_n$. The rule of composition is formally defined as:

$$\text{if } \vdash \{P\}Q_1\{R_1\} \text{ and } \vdash \{R_1\}Q_2\{R\} \text{ then } \vdash \{P\}(Q_1; Q_2)\{R\}$$

The inference rule for composition states that if the proven result of the first part of a program is identical with the preconditions under which the second part produces its intended results, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied. These rules allow us to infer new properties for a system.

Rule of iteration. A program can repeat a portion of a program S multiple times as long as a condition B holds. This can be written with the while notation as **while** B **do** S. If the condition B is *true* the program S is executed. Afterwards the condition B is checked again. This repeats as long as B holds. If one check returns that B is false, the program terminates. The rule of iteration can be formally written as:

$$\text{if } \vdash \{P \wedge B\}S\{P\} \text{ then } \vdash \{P\}(\textbf{while } B \textbf{ do } S)\{\neg B \wedge P\}$$

Note that this rule assumes that the program terminates eventually.

These rules of logical deduction can be used to create formal proofs for executable codes.

Contract-based Verification

Systems can consist of multiple components that interact with each other. This violates a core assumption of the previously introduced program verification, namely that no side effects occur and the program lines are executed sequentially. Interacting components are executed in parallel and can interact by different means, like shared variables. This requires other methods to verify properties for such systems. One way to analyze the system behavior is to consider all possible sequential interleavings of the interacting components. During the verification, all possible system states are checked and the interactions between the components cause the state-space to grow exponentially.

Contract-based verification [Mey92] aims at abstracting the interactions between components by introducing an *assume-guarantee paradigm*. Instead of considering all components in one overall system description that models all concrete component behaviors, contracts are used that define the interface behavior of the components. For a component that contains inputs Σ_{in} and outputs Σ_{out}, a contract c is a tuple

$$c = \{\Phi_{in}, \Phi_{out}\}$$

where Φ_{in} are assumptions, which are properties on the elements of Σ_{in}, and Φ_{out} are guarantees on the elements of Σ_{out}.

The verification of a system via contract-based verification consists of multiple steps. Each component is verified individually. The assumptions Φ_{in} are added as prerequisites to the verification and the guarantees Φ_{out} are the safety properties that need to be shown. In the system verification, the interactions between all components are checked via the contracts, i.e., it is checked whether the guarantees of the components fulfill the corresponding assumption of interacting components.

Figure 2.5 shows an example system consisting of three components. This system has one input and one output signal. All signals in the system are of the type integer. As overall system contract, the following should hold: If the input signal is less than 10, then the output signal is less than 60. Each component of the system has its own contract that it fulfills. To ensure the overall system contract, it is necessary to consider the interplay of the components. First, we consider the output of component A. Its contract states that the output is less than two times

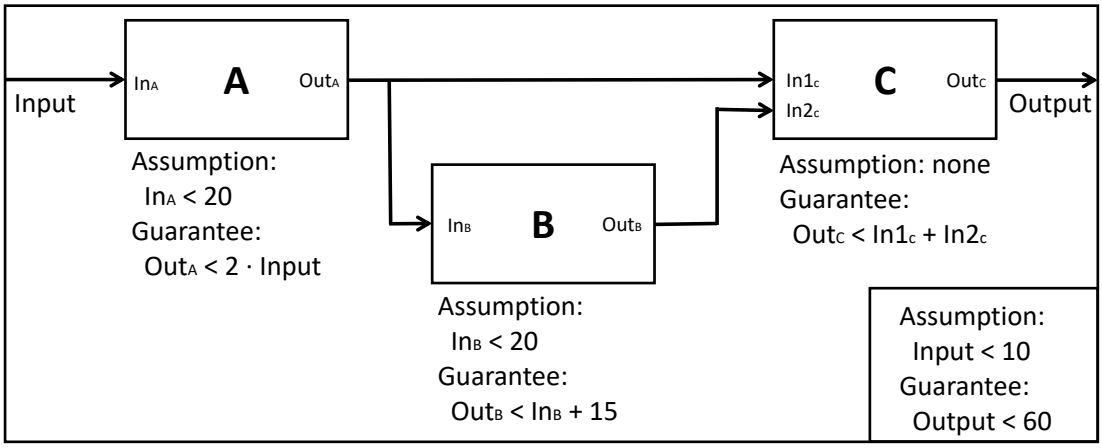


Figure 2.5: Components with Contracts, adapted from [Mur+13]

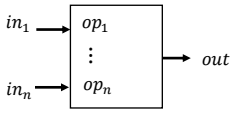
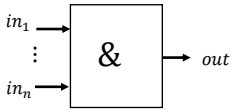
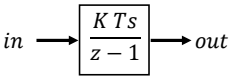
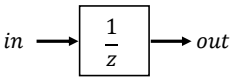
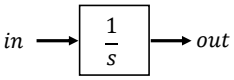
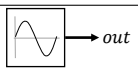
its input if the input is less than twenty. With the overall system assumption that the system input is less than 10 this is ensured. Furthermore, we can use the value 10 for the output guarantee of component A and obtain $Out_A < 2 \cdot 10$. This fulfills the input assumption of component B and therefore its output guarantee can be ensured. We insert the upper bound in the guarantee and obtain for its output value $Out_B < 20 + 15$. The last component has no assumptions, therefore, its guarantee always holds. To obtain an upper bound for its output signal, we consider the guarantees of the other two components, since $Out_C < Out_A + Out_B$. After inserting the upper bounds, we obtain $Out_C < 2 \cdot 10 + 20 + 15 = 55$. Lastly, we check whether the output of component C fulfills the overall system guarantee. The system guarantee $Output < 60$ can be ensured with $Out_C < 55$. Therefore, the overall system contract holds.

2.4 MATLAB/Simulink

MATLAB/Simulink [Matb] is a modeling language and integrated modeling tool. MATLAB is a numerical computing environment and Simulink is a MATLAB extension. Simulink provides a system description language and a graphical development environment that allows for the design and simulation of hybrid systems. Furthermore, Simulink allows for the automatic generation of executable code from modeled systems.

In Simulink, systems are modeled in a data-flow oriented system description language. Simulink models are directed graphs, where the nodes are *blocks* and edges are *signal lines*. Blocks are the basic building elements. They have input and output ports, and perform calculations on the data that is provided at their input ports and write the resulting data to their output ports. Signal lines define how the blocks are connected and transfer data that is written to the output ports of blocks to the input ports of following blocks. There are different kinds of blocks, e.g., direct feed-through, time-discrete, time-continuous, and control flow blocks. Each block defines the calculation that it performs. The timing behavior and calculation step times are defined by a *solver*.

Table 2.2: First Part of Showcase Simulink Blocks

Simulink block			**Behavior**
feed-through	Sum		Takes the incoming signals in_1 to in_n and writes the sum of the values to its output. For each incoming signal a positive or negative sign can be assigned.
	Logical		Takes the incoming signals in_1 to in_n and determines the output according to the internal logical operator.
time-discrete	Discrete Integrator		Takes the current value at each time step and adds it to its state variable. The state variable is written to the output.
	Unit Delay		At each discrete time step, stores the incoming value into an internal variable and writes the last stored value to its output.
time-continuous	Integrator		Takes the incoming value and integrates it over time. Writes the result to its output. This represents a continuous evolution. As its parameters an upper and lower bound for its output can be defined.
	Sine Wave		Generates the trajectory of a sine wave.

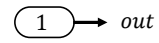
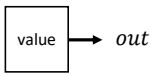
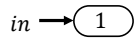
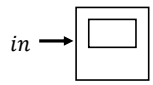
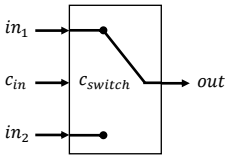
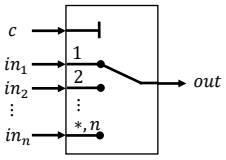
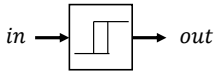
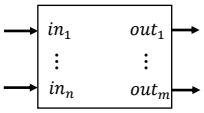
The Simulink block library provides a large set of predefined blocks with simple or complex functionality, ranging from simple arithmetic and logic blocks over control flow blocks to integrators or derivatives and complex transformations. Signals can carry discrete or continuous values. Special S-function blocks enable the definition of arbitrary functions. Blocks are categorized in libraries and external libraries can be included to extend the scope of usable blocks. Together with the Matlab library, linear and non-linear differential equations and complex mathematical functions can be modeled and simulated.

Blocks

The main building elements in Simulink are the blocks that define the calculations that are performed on the data in the system. Blocks can be assigned to one or more of several classes that enable modeling of different behaviors. In general, the behavior of blocks can be further defined by block parameters, e.g., initial values and default outputs.

Table 2.2 and Table 2.3 showcase basic Simulink blocks that represent important block classes. *Feed-through* blocks take the incoming signals to perform calculations and directly write the results to their output ports. They do not contain an inner state. Some example blocks for this class are arithmetic formulas like a *Sum* block, or logical gates. *Time-discrete* blocks are evaluated at discrete time steps. The step size is defined by the Simulink solver or can be defined for each block

Table 2.3: Second Part of Showcase Simulink Blocks

Simulink block			Behavior
sources & sinks	Inport		Provides values of a signal that is connected at an external input.
	Constant		Provides a constant value at its output.
	Outport		Takes a signal and writes it to a connected external output.
	Scope		Creates a 2D plot of the incoming signal value over time.
control flow	Switch		Checks whether the value of the second input signal C_{in} fulfills the condition c_{switch}. If the condition is fulfilled, the first input in_1 is written to the output, otherwise in_2 is written to its output.
	Multi-port Switch		The value at the first input port determines the input that is written to the output.
discontinuities	Relay		Whenever its input reaches a defined upper bound the output is set to a given on-value and when a lower bound is reached the output is set to a given off-value. If the input is in between the two bounds, the current output does not change.
layout	Sub-system		Encapsulates other Simulink blocks. Incoming signals at input ports are passed to internal *Inports* and signals of internal *Outports* are provided at output ports.

individually. The output of these blocks is constant between these steps. *Unit delay* and *Discrete Integrator* are blocks that are part of this category. *Time-continuous* blocks change their output signal continuously over time. For example, an *Integrator* integrates the incoming signal and provides the current state at its output, or the *Sine Wave* provides a continuous sine signal. *Source* blocks generate signal forms, while *sink* blocks take incoming signals for evaluation, e.g., a *Scope* plots incoming signals. *Control flow* blocks pass one of multiple inputs to its output without changing the value. Depending on the value of a special control input port the input signal to pass through is chosen. The *Switch* and *Multiport Switch* are examples of blocks in this class. *Discontinuities* can change between different output trajectories depending on the incoming signal. Their exact behavior is given by their internal parameters. The *Relay* block is part of this category. A special kind of block is the *Subsystem* block. Its behavior is modeled by an embedded Simulink model. Each subsystem contains other Simulink blocks. Such subsystems are used to encapsulate behavior of a model into subcomponents.

Signal Routing

Signals are used to connect output ports of blocks and input ports of blocks to each other. The restriction on the signal connections is that each input port can only have one incoming signal. To choose between various incoming signals and change the control flow, blocks like a *Switch* are used. There is no restriction how many ports an output port can be connected to. An output port can also be connected to the input ports of the same block. We write

$$BlockName1.OutputPortNumber \rightarrow BlockName2.InputPortNumber$$

to explicitly denote signal connections. Note that signals in Simulink are always directional, therefore, the left-hand side always denotes an output port and the right-hand side always denotes an input port. Note that an Inport block consists of one output port. Therefore, signals that represent the resulting input signal have the output port of this Inport block as their left-hand side.

It is possible to create loops that can contain one or more blocks. If the loop contains at least one block that stores an inner state, the resulting loop is called a feedback loop. If the loop only contains blocks that do not store an inner state, i.e., only consists of feed-through blocks, an *algebraic loop* is produced. An algebraic loop does not contain an inner variable. For some algebraic loops, it is possible to provide a value for outgoing signals. This can be achieved by constructing the underling equation system and solving it to obtain the output variables. However, this is not always possible and the use of algebraic loops is discouraged in the development of Simulink models.

Simulink allows the use of vector signals. A vector signal consists of multiple scalar signals, where each individual signal has its own trajectory. There are two possibilities to create vector signals. The *Mux* block takes n input signals and provides one output signal that is a vector with n elements. With a *Demux* block, an incoming vector signal can be split into single signals. A *Bus* block takes a number of named signals and provides a single output vector. A *BusSelector* block can access signals of this vector by their names. A Simulink model that contains vector signals can be represented by an equivalent model without vector signals. Each vector signal of the size n is replaced by n individual signal lines.

Solver

The *solver* in Simulink determines how the blocks in a model are evaluated during simulation. In general, all calculations are numerical solutions. To model continuous evolutions in Simulink, the solver performs approximations. The chosen solver determines the accuracy of the solution. The main categories of solvers are *discrete step size* solvers and *variable step size* solvers.

Discrete step size solvers perform discrete-time steps between the evaluation of blocks. Therefore, the time that elapsed between two consecutive evaluations is the same. This behavior simplifies the calculations. However, this behavior is imprecise. Events, like the change of a switch block, can occur between time steps. These events are not evaluated until the next time step or they can be missed entirely.

A variable step size solver adapts the time step size to achieve more accurate

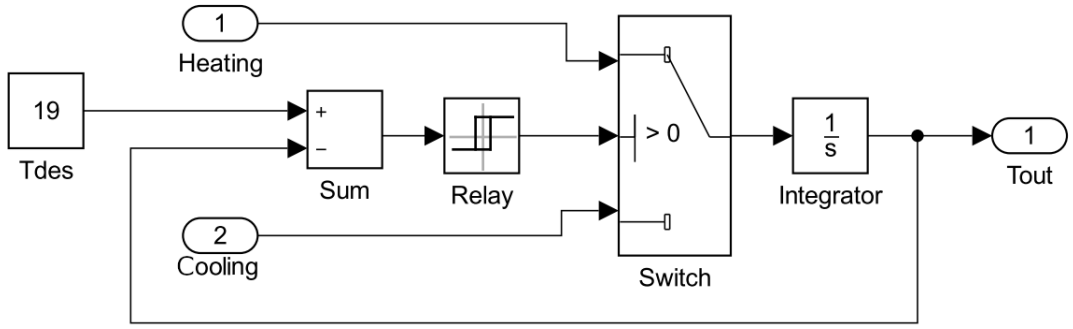


Figure 2.6: Simulink Model of a Temperature Control System

results. While the results are more accurate, the analysis of models that use variable step size solvers is more complex. During simulation, the solver changes the step size depending on the current dynamics of the model. If only small changes in the values are present, the solver executes larger time steps. If the signals in the system change quickly, smaller time steps are executed. Furthermore, the solver executes smaller time steps if control flow blocks or discontinuities are near the values that cause a change at their outputs. This change is called zero-crossing. Depending on the step size, the system behavior can change and the higher the delay before a zero-crossing is detected the less precise is the resulting behavior.

Temperature Control System modeled in Simulink

In this section, we present a temperature control system that is modeled in Simulink. We use this model as a running example in the remainder of this thesis. The purpose of this system is to keep a temperature value in defined bounds. To achieve this, heating or cooling can be activated. The level of heating or cooling are given as inputs and the current temperature is provided at an output port.

Figure 2.6 shows our temperature control system modeled in Simulink. It takes a heating value (*Heating*) and a cooling value (*Cooling*) as input, which are real values that represent the temperature gradient, and adjusts the current temperature accordingly. The desired temperature is defined as a constant block *Tdes*, which we have set to 19 in this model. In a feedback loop, the following decision is made at the *Switch* block: If the current temperature is lower than desired, *Heating* is forwarded, otherwise the switch passes *Cooling* through. The output temperature (*Tout*) is calculated by integrating the input values over (continuous) time with the *Integrator* block, and the result is used to control the *Switch*. We use a *Relay* block to prevent rapid switching, i.e., a fast change between heating and cooling. We have set the parameter for the *Relay* block as follows: If the current temperature is more than one degree smaller than desired, it yields *true*, if the temperature is more than one degree greater than desired, it yields *false* and otherwise, it keeps the last value. Note that this is a simple example of a hybrid system, as it contains the temperature as a value that evolves continuously over time and also discrete state changes that are given by switching between the heating and cooling.

2.5 Differential Dynamic Logic and KeYmaera X

Differential dynamic logic ($\mathsf{d}\mathcal{L}$) [Pla08] provides a formal foundation to model hybrid systems. It can be used to capture the discrete and continuous system behavior of hybrid systems in one representation and enables formal analysis of the modeled systems. In $\mathsf{d}\mathcal{L}$, hybrid systems are modeled via *hybrid programs* and modal operators allow the designer to formulate system properties. To facilitate the analysis of systems that are modeled in $\mathsf{d}\mathcal{L}$, the models can be used as input for the theorem prover KeYmaera X [Ful+15]. KeYmaera X is an interactive theorem prover for deductive verification of hybrid systems that are modeled in differential dynamic logic ($\mathsf{d}\mathcal{L}$).

Syntax of $\mathsf{d}\mathcal{L}$

In the following, we introduce the syntax of $\mathsf{d}\mathcal{L}$ based on [Pla17].

Variables. In $\mathsf{d}\mathcal{L}$, the set of all *variables* is $\mathscr{V}$. For some variables $x \in \mathscr{V}$, there exist variables of the form $x' \in \mathscr{V}'$. These are called *differential symbols*. It is assumed that the set of all variables contains all of its differential symbols $\mathscr{V}' \subseteq \mathscr{V}$. Therefore, it is also possible to have differential symbols in the form of x''.

Function symbols. Functions are represented by *function symbols*. In the following, we use f, g, h as function symbols. Furthermore, number literals, like 0 or 1, are used to represent function symbols without arguments. In these cases they are interpreted as the number that they denote.

Predicate symbols. Predicate symbols can evaluate to *true* or *false*. We use p, q, r as predicate symbols in the following examples.

Variable symbols. Variables in the program are denoted by *variable symbols*. Variables can change their value according to the program. In the following, we use x, y, z to represent variables.

Program constants. The values of program constants are fixed. In the following, we use a, b, c to represent program constants.

Definition 2.4 (Terms)**.** Terms are defined by the following grammar (where θ, η, $\theta_1, ...\theta_k$ are terms, $x \in \mathscr{V}$ is a variable, and f is a function symbol)

$$\theta ::= x \mid f(\theta_1, ...\theta_k) \mid \theta + \eta \mid \theta \cdot \eta \mid (\theta)'$$

The differential $(\theta)'$ describes the local change of the term θ, which is determined by the change of its inner variables x.

Definition 2.5 (Hybrid Programs)**.** Hybrid programs are defined by the following grammar (where α, β are hybrid programs, $x \in \mathscr{V}$ is a variable, θ is a term (possibly containing x), and ψ is a $\mathsf{d}\mathcal{L}$ formula that can evaluate to *true* or *false*)

$$\alpha, \beta ::= x := \theta \mid x := * \mid ?\psi \mid (x_1' = \theta_1, ...x_n' = \theta_n \,\&\, \psi) \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

An evaluation of a hybrid program is called a *run*. With a *discrete assignment* $x := \theta$, a value given by θ is assigned to variable x. A *nondeterministic assignment* $x := *$ assigns an arbitrary value to the variable x. A *test formula* $?\psi$ takes a condition ψ and behaves as follows: If ψ evaluates to *true*, the test formula is handled like a *skip* and the run continuous without further changes. If ψ does not hold, the run is aborted. In a *continuous evolution* $(x_1' = \theta_1, ...x_n' = \theta_n \,\&\, \psi)$, the variables x_i evolve according to the given gradients θ_i as long as the invariant ψ holds, which is also called the evolution domain. Note that the evolution may stop at any time. The *nondeterministic choice*[1] $\alpha \cup \beta$ allows for a branch in the execution, where either α or β can be executed. The *sequential composition* $\alpha; \beta$ connects two hybrid programs so that α is executed before β. A *nondeterministic repetition* α^* repeats an inner program α an arbitrary number of times.

To give a better understanding for nondeterministic repetitions and test formulas, we present a while loop that is written as hybrid program.

$$\{?(\psi); \alpha\}^*; ?(\neg\psi)$$

In this hybrid program the formula ψ represents the loop condition and the hybrid program α represents the loop body. The nondeterministic repetition can be repeated an arbitrary number of times but only runs where the loop condition ψ evaluates to *true* are evaluated further. After the nondeterministic repetition is executed an arbitrary number of times, only runs are evaluated where ψ does not hold anymore. Together, only runs are evaluated where α is repeated as long as ψ holds and the loop is exited if ψ does not hold anymore.

With these hybrid programs, shorthand notations for the following control flow statements can be given.

$$\text{if } (\phi)\ \alpha \text{ else } \beta \equiv ?\phi; \alpha \cup ?(\neg\phi); \beta$$
$$\text{if } (\phi)\ \alpha \equiv ?\phi; \alpha \cup ?(\neg\phi)$$
$$\text{skip} \equiv ?true$$

An if (ϕ) α else β can be represented as a nondeterministic choice and test formulas that ensure the conditions on the *if* branch and the *else* branch. Without an *else* branch, an empty *else* branch is added in $\mathsf{d}\mathcal{L}$ to model the behavior for the cases where the condition is not fulfilled. Furthermore, a *skip* statement can be written as $?true$.

Definition 2.6 ($\mathsf{d}\mathcal{L}$ Formulas)**.** Formulas are defined by the following grammar (where ϕ, ψ are $\mathsf{d}\mathcal{L}$ formulas, $\theta, \eta, \theta_1, ...\theta_k$ are terms, p is a predicate symbol, $x \in \mathcal{V}$ is a variable, and α is a hybrid program)

$$\phi, \psi ::= \theta \geq \eta \mid p(\theta_1, ...\theta_k) \mid \neg\phi \mid \phi \wedge \psi \mid \forall x\phi \mid \exists x\phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

The box modality $[\alpha]\phi$ expresses that the formula ϕ holds after all possible runs of α, and the diamond modality $\langle\alpha\rangle\phi$ expresses that ϕ holds after some runs of α. Note that $[\alpha]\phi$ is equivalent to $\neg\langle\alpha\rangle\neg\phi$ and $\forall x\phi$ is equivalent to $\neg\exists x\neg\phi$.

[1] In written $\mathsf{d}\mathcal{L}$ code, a nondeterministic choice is represented as α `++` β.

Example hybrid program. Listing 2.1 shows a hybrid program that models a person on an escalator [MP17]. In *ProgramVariables*, we define the variables that are used in the model. The R indicates that the variables are real numbers. The variable x denotes the position of the person and v the speed of the escalator. The model of the system is given in *Problem*. Initially the person starts at the second or a higher step and the escalator goes up. The person can take one step down ($x := x - 1$) if they are not already at the bottom ($x \geq 1$), or move up with speed v for an arbitrary time ($x' = v$). The choice between the step down or the movement up can be repeated an arbitrary number of times ($\{\cdot\}^*$). The specification $[\cdot]$ requires that under the precondition $x \geq 2 \land v > 0$, after all possible executions, $x \geq 0$, i.e., the person does not step lower than the step at position 0.

```
ProgramVariables.
  R x.
  R v.
End.

Problem.
  x >= 2 & v > 0
-> {
  [
    {
      ?(x >= 1);
      x := x - 1;
    ++
      {x' = v}
    }*
  ] x >= 0}
End.
```

Listing 2.1: A Hybrid Program of an Escalator

Dynamic Semantics

The dynamic semantics defines the evaluation of the language constructs in $\mathsf{d}\mathcal{L}$ according to [Pla17]. The basic idea of the dynamic semantics of $\mathsf{d}\mathcal{L}$ is to define how a given state ν is changed by the execution of a hybrid program to a new state ω. It is possible to obtain more than one resulting state and the execution can start from a set of states. Each execution of hybrid programs from a set of starting states to a set of resulting states is called a run. To define these transition semantics, we first present how the interpretation of basic $\mathsf{d}\mathcal{L}$ constructs is defined. Afterwards, we present the interpretation of hybrid programs.

Terms evaluate to real values, formulas evaluate to truth-values and hybrid programs evaluate to reachable states. The values of variables and differential symbols can change over time, therefore their values are given by the state.

A state maps variables to real values:

$$\nu : \mathcal{V} \to \mathbb{R}$$

A state includes the mappings of differential symbols $\mathcal{V}' \subseteq \mathcal{V}$ to $\mathbb{R}$. The set of states is denoted by $\mathcal{S}$. For a set $X \subseteq \mathcal{S}$, the complement set is given by $X^C = \mathcal{S} \backslash X$. ν_x^r denotes the state that matches with state ν except for the value of variable x, which is changed to $r \in \mathbb{R}$.

The interpretation of a function symbol f with n arguments in interpretation I is a function $I(f) : \mathbb{R}^n \to \mathbb{R}$ with n arguments. The set of interpretations is given by $\mathcal{I}$. The semantics of a term θ is a mapping $[\![\theta]\!] : \mathcal{I} \to (\mathcal{S} \to \mathbb{R})$, i.e. interpretations are mapped to a real number.

Definition 2.7 (Semantics of terms)**.** The semantics of a term θ in interpretation I and state $\nu \in \mathcal{S}$ is its value $I\nu[\![\theta]\!] \in \mathbb{R}$. That means, for a given state ν of the model, we map all variables and symbols to real numbers. The semantics of terms is defined as follows:

$$
\begin{aligned}
I\nu[\![x]\!] &= \nu(x) \text{ for variable } x \in \mathcal{V} \\
I\nu[\![f(\theta_1, ..., \theta_k)]\!] &= I(f)(I\nu[\![\theta_1]\!], ..., I\nu[\![\theta_k]\!]) \\
I\nu[\![\theta + \eta]\!] &= I\nu[\![\theta]\!] + I\nu[\![\eta]\!] \\
I\nu[\![\theta \cdot \eta]\!] &= I\nu[\![\theta]\!] \cdot I\nu[\![\eta]\!] \\
I\nu[\![(\theta)']\!] &= \sum_{x \in \mathcal{V}} \nu(x') \frac{\delta I[\![\theta]\!]}{\delta x}(\nu) = \sum_{x \in \mathcal{V}} \nu(x') \frac{\delta I\nu[\![\theta]\!]}{\delta x}
\end{aligned}
$$

$I\nu[\![x]\!]$ describes that the interpretation of a variable x is its value $\nu(x)$. The interpretation of a function of terms $I\nu[\![f(\theta_1, ..., \theta_k)]\!]$ is the interpretation of the function where each parameter $\theta_1, ..., \theta_k$ is interpreted accordingly. A sum of two terms $I\nu[\![\theta + \eta]\!]$ is interpreted by adding the interpretations of the two individual terms. Analogous, the product of two terms $I\nu[\![\theta \cdot \eta]\!]$ is the product of the interpretations of the terms. The interpretation of a derivative $I\nu[\![(\theta)']\!]$ is the sum of its partial derivatives.

Definition 2.8 (Semantics of formulas)**.** Semantics of a $\mathsf{d}\mathcal{L}$ formula ϕ, for each interpretation I with a corresponding set of states $\mathcal{S}$, is the subset $I[\![\phi]\!] \subseteq \mathcal{S}$ of states in which ϕ is true. It is defined inductively as follows:

$$
\begin{aligned}
I[\![\theta \geq \eta]\!] &= \{\nu \in \mathcal{S} : I\nu[\![\theta]\!] \geq I\nu[\![\eta]\!]\} \\
I[\![p(\theta_1, ..., \theta_k)]\!] &= \{\nu \in \mathcal{S} : (I\nu[\![\theta_1]\!], ..., I\nu[\![\theta_k]\!]) \in I(p)\} \\
I[\![\neg\phi]\!] &= (I[\![\phi]\!])^C \\
I[\![\phi \wedge \psi]\!] &= I[\![\phi]\!] \cap I[\![\psi]\!] \\
I[\![\exists x \phi]\!] &= \{\nu \in \mathcal{S} : \nu_x^r \in I[\![\phi]\!] \text{ for some } r \in \mathbb{R}\} \\
I[\![\forall x \phi]\!] &= \{\nu \in \mathcal{S} : \nu_x^r \in I[\![\phi]\!] \text{ for all } r \in \mathbb{R}\} \\
I[\![\langle\alpha\rangle\phi]\!] &= I[\![\alpha]\!] \circ I[\![\phi]\!] = \{\nu \in \mathcal{S} : \omega \in I[\![\phi]\!] \\
&\qquad \text{for some } \omega \text{ such that } (\nu, \omega) \in I[\![\alpha]\!]\} \\
I[\![[\alpha]\phi]\!] &= I[\![\neg\langle\alpha\rangle\neg\phi]\!] = \{\nu \in \mathcal{S} : \omega \in I[\![\phi]\!] \\
&\qquad \text{for all } \omega \text{ such that } (\nu, \omega) \in I[\![\alpha]\!]\}
\end{aligned}
$$

The interpretation $I[\![\theta \geq \eta]\!]$ of a relation symbol is the set of states, where the interpretations of the terms θ and η fulfill the relations. The interpretation of a predicate symbol $I[\![p(\theta_1, ..., \theta_k)]\!]$ is the set of states where the interpretations of its parameter terms fulfill the predicate symbol. The interpretation of a negation of a formula $I[\![\neg\phi]\!]$ is the complement of the interpretation of the formula ϕ. A conjunction of two formulas $I[\![\phi \wedge \psi]\!]$ is interpreted by creating the intersection of the interpretations of the individual formulas. The interpretation of the *exist* quantifier $I[\![\exists x\phi]\!]$ evaluates to true if there exists an assignment of the variable x where the formula ϕ is interpreted as true. The interpretation of the all quantifier $I[\![\forall x\phi]\!]$ evaluates to true if all assignments of the variable x evaluate the formula ϕ to true. The diamond operator $I[\![\langle\alpha\rangle\phi]\!]$ is interpreted as follows: The evaluation of the hybrid program α creates a set of states (variable assignments). If there exists at least one state where the interpretation of the formula ϕ is true, then the diamond operator is interpreted as true. The interpretation of the box operator $I[\![[\alpha]\phi]\!]$ is reduced to an interpretation of the diamond operator. It means, that the box operator evaluates to true if all states produced by α fulfill the formula ϕ.

A formula in $\mathsf{d}\mathcal{L}$ evaluates to *true at state* ν *in* I ($I, \nu \models \phi$) iff $\nu \in I[\![\phi]\!]$. A formula in $\mathsf{d}\mathcal{L}$ is *valid in* I ($I \models \phi$) iff $I[\![\phi]\!] = \mathcal{S}$, i.e. $\nu \in I[\![\phi]\!]$ for all states ν. A formula in $\mathsf{d}\mathcal{L}$ is *valid* ($\models \phi$), iff $I \models \phi$ for all interpretations I.

Definition 2.9 (Transition semantics of Hybrid Programs)**.** For each interpretation I, each hybrid program α is interpreted semantically as a binary transition relation $I[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ on states. They are defined inductively as follows:

$$
\begin{aligned}
I[\![x := \theta]\!] &= \{(\nu, \nu_x^r) : r = I\nu[\![\theta]\!]\} = \{(\nu, \omega) : \omega = \nu \text{ except } \omega(x) = I\nu[\![\theta]\!]\} \\
I[\![?\psi]\!] &= \{(\nu, \nu) : \nu \in I[\![\psi]\!]\} \\
I[\![x' = \theta \;\&\; \psi]\!] &= \{(\nu, \omega) : \nu = \phi(0) \text{ on } \{x'\}^C \text{ for some function } \phi : [0, r] \to \mathcal{S} \\
&\quad \text{of some duration } r \text{ satisfying } I, \phi \models x' = \theta \wedge \psi\} \\
&\quad \text{where } I, \phi \models x' = \theta \wedge \psi \text{ iff } \phi(\xi) \in I[\![x' = \theta \wedge \psi]\!] \text{ and } \phi(0) = \phi(\xi) \\
&\quad \text{on } \{x, x'\}^C \text{ for all } 0 \leq \xi \leq r \text{ and if } \frac{d\phi(t)(x)}{dt}(\xi) \text{ exists and is equal} \\
&\quad \text{to } \phi(\xi)(x') \text{ for all } 0 \leq \xi \leq r. \\
I[\![\alpha \cup \beta]\!] &= I[\![\alpha]\!] \cup I[\![\beta]\!] \\
I[\![\alpha; \beta]\!] &= I[\![\alpha]\!] \circ I[\![\beta]\!] = \{(\nu, \omega) : (\nu, \mu) \in I[\![\alpha]\!], (\mu, \omega) \in I[\![\beta]\!] \text{ for some } \mu\} \\
I[\![\alpha^*]\!] &= (I[\![\alpha]\!])^* = \bigcup_{n \in \mathbb{N}} I[\![\alpha^n]\!] \text{ with } \alpha^{n+1} \equiv \alpha^n; \alpha \text{ and } \alpha^0 \equiv ?true
\end{aligned}
$$

A discrete assignment $I[\![x := \theta]\!]$ creates a new state that is equal to the initial state except for the value of the variable x that is assigned to the evaluation of the term θ. A test formula $I[\![?\psi]\!]$ can only be evaluated if the initial state fulfills the formula ψ. In this case, the new state is identical to the initial state. Therefore, the test formula removes all initial states from the evaluation that violate the formula ψ. A continuous evolution $I[\![x' = \theta \;\&\; \psi]\!]$ creates a new state, where the value of the evolving variable x is changed and all other variables keep their values. For all

new states, the possible values for x are on the trajectory that is given by θ. The trajectory can only evolve as long as it fulfills the evolution domain ψ. That means that we do not only reach a state in which ψ holds but along the whole trajectory of x the formula ψ does also hold. All possible values on the trajectory from the initial value to the violation of ψ are in the set of new states. A nondeterministic choice $I[\![\alpha \cup \beta]\!]$ creates the union of the individual interpretations of α and β. The sequential operator $I[\![\alpha;\beta]\!]$ creates a set of final states where an intermediate state exists that is created by the interpretation of α and is the initial state for the interpretation of β. The final states of the sequential operator are the final states of β. A nondeterministic repetition $I[\![\alpha^*]\!]$ is interpreted as sequential repetition of α where it can be executed any number of times.

Static Semantics

The static semantics of $\mathsf{d}\mathcal{L}$ defines aspects concerning the usage of variables. We use the static semantics later on, to find all variables in a hybrid program that are written to, i.e., all variables that change their value in the execution of a given hybrid program. The aspects of the static semantics follow from the syntactic structure without running the programs or evaluating their dynamical effects. The static semantics identifies *free variables* and *bound variables*.

Free variables are all variables that the value of an expression depends on. *Bound variables* can change their value during the evaluation of an expression.

Variables of $\mathsf{d}\mathcal{L}$ formula ϕ, whether free or bound, are $V_F(\phi) = FV_F(\phi) \cup BV_F(\phi)$. The variables of hybrid program α, whether free or bound, are given by $V_{HP}(\alpha) = FV_{HP}(\alpha) \cup BV_{HP}(\alpha)$.

Bound variables. Bound variables are variables that change their value in the execution of hybrid programs. The set $BV_{HP}(\alpha)$ is the smallest set for which the following bound effect property holds: If $(\nu,\omega) \in I[\![\alpha]\!]$, then $\nu = \omega$ on $BV_{HP}(\alpha)^C$. That means that all variables that are not in the set $BV_{HP}(\alpha)$ have the same value after the execution of α that they had at the beginning.

Definition 2.10 (Bound variables of $\mathsf{d}\mathcal{L}$ formulas)**.** The set $BV_F(\phi) \subseteq \mathcal{V}$ of (syntactically) bound variables of $\mathsf{d}\mathcal{L}$ formula ϕ is defined inductively as:

$$\begin{aligned} BV_F(p(\theta_1,...,\theta_k)) &= \emptyset \\ BV_F(\neg\phi) &= BV_F(\phi) \\ BV_F(\phi \wedge \psi) &= BV_F(\phi) \cup BV_F(\psi) \\ BV_F(\forall x\phi) = BV_F(\exists x\phi) &= \{x\} \cup BV_F(\phi) \\ BV_F([\alpha]\phi) = BV_F(\langle\alpha\rangle\phi) &= BV_{HP}(\alpha) \cup BV_F(\phi) \end{aligned}$$

A formula θ that only consists of simple predicates has no bound variables, since the evaluation of such a formula only reads variables and does not change them. Quantifier symbols create a set of bound variables that consists of the referenced variable x and the bound variables of the bound variables of the term ϕ. The set

of bound variables for box or diamond modality consists of all bound variables of the internal hybrid program α and the bound variables of the formula ϕ.

Definition 2.11 (Bound variables of $\mathsf{d}\mathcal{L}$ hybrid programs)**.** The set $BV_{HP}(\alpha) \subseteq \mathcal{V}$ of (syntactically) bound variables of hybrid program α, i.e. all variables that may potentially be written to, is defined inductively as:

$$\begin{aligned} BV_{HP}(x := \theta) &= \{x\} \\ BV_{HP}(x := *) &= \{x\} \\ BV_{HP}(?\psi) &= \emptyset \\ BV_{HP}(x' = \theta \& \psi) &= \{x, x'\} \\ BV_{HP}(\alpha \cup \beta) = BV_{HP}(\alpha; \beta) &= BV_{HP}(\alpha) \cup BV_{HP}(\beta) \\ BV_{HP}(\alpha^*) &= BV_{HP}(\alpha) \end{aligned}$$

For discrete assignments and nondeterministic assignments, the set of bound variables contains the variables of the assignment. A test formula only reads values and has an empty set of bound variables. Note that x and x' are bound by a differential equation $x' = \theta$, since both may change their value.

Definition 2.12 (Must-bound variables of $\mathsf{d}\mathcal{L}$ hybrid programs)**.** The set of (syntactically) must-bound variables $MBV_{HP}(\alpha) \subseteq BV_{HP}(\alpha) \subseteq \mathcal{V}$ of hybrid program α, i.e. all those that must be written to on all paths of α, is defined inductively as:

$$\begin{aligned} MBV_{HP}(\alpha) &= BV_{HP}(\alpha) && \text{for atomic hybrid programs } \alpha \\ MBV_{HP}(\alpha \cup \beta) &= MBV_{HP}(\alpha) \cap MBV_{HP}(\beta) \\ MBV_{HP}(\alpha; \beta) &= MBV_{HP}(\alpha) \cup MBV_{HP}(\beta) \\ MBV_{HP}(\alpha^*) &= \emptyset \end{aligned}$$

An atomic hybrid program only consists of one path and therefore the set of must-bound variables is equal to the set of bound variables. Since a nondeterministic choice consists of two paths, the execution of α or the execution of β, the set of must-bound variables for a nondeterministic choice is the set of variables that are must-bound in both α and β. A sequential operator connects all possible paths of α to all possible paths of β and the set of must-bound variables of a sequential operator is the set of variables that are either written to in all paths of α or in all paths of β. Since one path of a nondeterministic repetition is the case that α is repeated zero times and this path does not change any variable, the set of must-bound variables for a nondeterministic repetition is empty.

Free variables. In the following, we first present a general intuition for free variables for $\mathsf{d}\mathcal{L}$ terms, $\mathsf{d}\mathcal{L}$ formulas and hybrid programs and afterwards present their definitions.

Free variables are variables that are read in a term, formula or hybrid program and influence the outcome of its evaluation. A variable $x \in \mathcal{V}$ is a free variable of a term, formula or hybrid program, if the interpretation of the term, formula or

hybrid program does change if the value of x changes and all other variables of the system stay the same. The complement set $\{x\}^C$ contains all system variables except x.

$$FV_T(\theta) = \bigcup\{x \in \mathcal{V} : \text{there are } I \text{ and } \nu = \tilde{\nu} \text{ on } \{x\}^C \text{ such that } I\nu[\![\theta]\!] \neq I\tilde{\nu}[\![\theta]\!]\}$$
$$FV_F(\phi) = \bigcup\{x \in \mathcal{V} : \text{there are } I \text{ and } \nu = \tilde{\nu} \text{ on } \{x\}^C \text{ such that } \nu \in I[\![\theta]\!] \not\ni \tilde{\nu}\}$$
$$FV_{HP}(\alpha) = \bigcup\{x \in \mathcal{V} : \text{there are } I, \nu, \tilde{\nu}, \omega \text{ such that } \nu = \tilde{\nu} \text{ on } \{x\}^C \text{ and } (\nu, \omega) \in I[\![\alpha]\!] \text{ but there is no } \tilde{\omega} \text{ with } \omega = \tilde{\omega} \text{ on } \{x\}^C \text{ such that } (\tilde{\nu}, \tilde{\omega}) \in I[\![\alpha]\!]\}$$

Definition 2.13 (Free variables of $\mathsf{d}\mathcal{L}$ terms)**.** The set $FV_T(\theta) \subseteq \mathcal{V}$ of (syntactically) free variables of term θ, i.e. those that occur in θ directly or indirectly, is defined inductively as:

$$\begin{aligned} FV_T(x) &= \{x\} && \text{hence } FV_T(x') = \{x'\} \\ FV_T(f(\theta_1, ..., \theta_k)) &= FV_T(\theta_1) \cup ... \cup FV_T(\theta_k) && \text{where } f \text{ can also be } + \text{ or } \cdot \\ FV_T((\theta)') &= FV_T(\theta) \cup FV_T(\theta)' \end{aligned}$$

The set of free variables for a term that just reads a variable x consists of just this variable. A function symbol can combine multiple terms $\theta_1, ..., \theta_k$. The free variables of this function symbol are all free variables of all these terms. The set of free variables for the derivative of a term θ consists of the free variables of the term and the derivatives of these variables.

Definition 2.14 (Free variables of $\mathsf{d}\mathcal{L}$ formulas)**.** The set $FV_F(\phi)$ of (syntactically) free variables of $\mathsf{d}\mathcal{L}$ formula ϕ, i.e. all that occur in ϕ outside the scope of quantifiers or modalities binding it, is defined inductively as:

$$\begin{aligned} FV_F(p(\theta_1, ..., \theta_k)) &= FV_T(\theta_1) \cup ... \cup FV_T(\theta_k) \\ FV_F(\neg\phi) &= FV_F(\phi) \\ FV_F(\phi \wedge \psi) &= FV_F(\phi) \cup FV_F(\psi) \\ FV_F(\forall x\phi) = FV_F(\exists x\phi) &= FV_F(\phi) \setminus \{x\} \\ FV_F([\alpha]\phi) = FV_F(\langle\alpha\rangle) &= FV_{HP}(\alpha) \cup (FV_F(\phi) \setminus MBV_{HP}(\alpha)) \end{aligned}$$

The set of free variables for a predicate symbol that combines the terms $\theta_1, ..., \theta_k$ consists of all free variables that occur in the terms. The free variables of a quantified formula are defined by removing its bound variables as $FV_F(\forall x\phi) = FV_F(\phi) \setminus \{x\}$, since all occurrences of x in ϕ are bound by $\forall x$. The bound variables of a program in a modality act in a similar way, except that the program itself may read variables during the computation, so its free variables need to be taken into account.

Definition 2.15 (Free variables of hybrid programs)**.** The set $FV_{HP}(\alpha) \subset \mathcal{V}$ of (syntactically) free variables of hybrid program α, i.e. all those that may potentially be read, is defined inductively as:

$$
\begin{aligned}
FV_{HP}(x := \theta) &= FV_T(\theta) \\
FV_{HP}(x := *) &= \emptyset \\
FV_{HP}(?\psi) &= FV_F(\psi) \\
FV_{HP}(x' = \theta \& \psi) &= \{x\} \cup FV_T(\theta) \cup FV_F(\psi) \\
FV_{HP}(\alpha \cup \beta) &= FV_{HP}(\alpha) \cup FV_{HP}(\beta) \\
FV_{HP}(\alpha; \beta) &= FV_{HP}(\alpha) \cup (FV_{HP}(\beta) \setminus MBV_{HP}(\alpha)) \\
FV_{HP}(\alpha^*) &= FV_{HP}(\alpha)
\end{aligned}
$$

The set of free variables for a discrete assignment consists of all variables that are read in the term for the assignment θ. A nondeterministic assignment reads no variables and the set of free variables is empty. The set of free variables for a continuous evolution consists of the evolving variable x, the variables that are read in the evolution term θ and the variables that are read in the evolution domain ψ. A sequential operator first executes a hybrid program α and afterwards a hybrid program β. All free variables of α are also in the set of free variables of the sequential operator. Free variables of β that are also must-bound in α, i.e., they are written to in all paths of α, are not in the set of free variables of the sequential operator. A variable is a free variable, if a change of its initial value changes the interpretation of the overall hybrid program. Since α overwrites this initial value for must-bound variables, this initial value has no influence of the interpretation of β in the sequential operator. Only free variables of β that are not in the set of must-bound variables of α are in the set of free variables of the sequential operator.

Coincidence for terms. The value of a term only depends on the values of its free variables. Therefore, a term θ that is evaluated in two different states $\nu, \tilde{\nu}$ that agree on its free variables $FV_T(\theta)$ provides the same values of θ in both states. Accordingly, the value of a term will agree for different interpretations I, J that agree on the symbols $\Sigma(\theta)$ that occur in θ.

The set $FV_T(\theta)$ is the smallest set with the coincidence property for θ: If $\nu = \tilde{\nu}$ on $FV_T(\theta)$ and $I = J$ on $\Sigma(\theta)$, then $I\nu[\![\theta]\!] = J\tilde{\nu}[\![\theta]\!]$.

In particular, the semantics of differentials is a sum over just the free variables:

$$
I\nu[\![(\theta)']\!] = \sum_{x \in FV_T(\theta)} \nu(x') \frac{\delta I[\![\theta]\!]}{\delta x}(\nu) = \sum_{x \in FV_T(\theta)} \nu(x') \frac{\delta I\nu[\![\theta]\!]}{\delta x}
$$

Coincidence for formulas. The evaluation of a $\mathsf{d}\mathcal{L}$ formula ϕ in two different states $\nu, \tilde{\nu}$ that agree on its free variables $FV_F(\phi)$ in $I = J$ on $\Sigma(\phi)$ provides the same truth-values of ϕ in both states.

The set $FV_F(\phi)$ is the smallest set with the coincidence property for ϕ: If $\nu = \tilde{\nu}$ on $FV_F(\phi)$ and $I = J$ on $\Sigma(\phi)$, then $\nu \in I[\![\phi]\!]$ iff $\tilde{\nu} \in J[\![\phi]\!]$.

Coincidence for programs. The runs of a hybrid program α only depend on the values of its free variables, because its behavior cannot depend on the values of variables that it never reads. If $\nu = \tilde{\nu}$ on $FV_{HP}(\alpha)$ and $I = J$ on $\Sigma(\phi)$ and $(\nu, \omega) \in I[\![\alpha]\!]$, then there is an $\tilde{\omega}$ such that $(\tilde{\nu}, \tilde{\omega}) \in J[\![\alpha]\!]$ and ω and $\tilde{\omega}$ agree on $FV_{HP}(\alpha)$. In fact, the final states $\omega, \tilde{\omega}$ continue to agree on any set $V \supseteq FV_{HP}(\alpha)$ that the initial states $\nu, \tilde{\nu}$ agreed on. The corresponding pairs of initial and final states of a run of hybrid program α already agree on the complement $BV_{HP}(\alpha)^C$.

The set $FV_{HP}(\alpha)$ is the smallest set with the coincidence property for α: If $\nu = \tilde{\nu}$ on $V \supseteq FV_F(\phi)$, $I = J$ on $\Sigma(\phi)$ and $(\nu, \omega) \in I[\![\alpha]\!]$, then there is an $\tilde{\omega}$ such that $(\tilde{\nu}, \tilde{\omega}) \in J[\![\alpha]\!]$ and $\omega = \tilde{\omega}$ on V.

In this section, we have first introduced $\mathsf{d}\mathcal{L}$ and its hybrid programs. Hybrid programs enable us to create formal models of systems that combine discrete and continuous behavior. These formal $\mathsf{d}\mathcal{L}$ models can be used as input for the interactive theorem prover KeYmaera X.

Furthermore, we have presented the dynamic and static semantics of $\mathsf{d}\mathcal{L}$. The dynamic semantics define the interpretations of hybrid programs, which can be seen as runs. The state of the system defined by the values of all variables at a time and the run of a hybrid program changes these values. The static semantics define the variables that influence the result of the execution of a hybrid program. Overall, this enables us to compare the runs of different hybrid programs and compare their behavior.

2.6 Service-Oriented Design

Existing implementations can be reused to reduce the development effort of new systems. However, there are challenges in the reuse of existing solutions. First, the development of solutions to subproblems does cost effort and money, so the results are often not simply put to public use. Second, many developers and businesses use their own interface standards and data types. Therefore, solutions that were developed in a different context cannot simply be inserted into another project.

Service-Oriented-Architectures (SOA) aim at providing reusable solutions for different users. The following ideas of SOA are taken from the OASIS Reference Model for SOA [OAS]. There are different definitions for SOA, which can differ in context, abstraction and concrete wordings. It should be noted that SOA does not describe one concrete architecture, but a paradigm to create concrete architectures. By providing rules how different participants interact when using a system, it helps to determine how to create a concrete software architecture. SOA enables access and usage of distributed competences. Services are reusable building blocks, which can be evaluated independently of the application and computing platform that is used to execute the implementation [PV06].

Figure 2.7 depicts a general workflow of a service-oriented architecture. The main common principle is that a *Service User* can access *Services* of a *Service Provider* that can be integrated into the *User Application*. The *Services* are provided via a *Service Library* that enables a user to choose the appropriate service for their application.

The three key principles are:

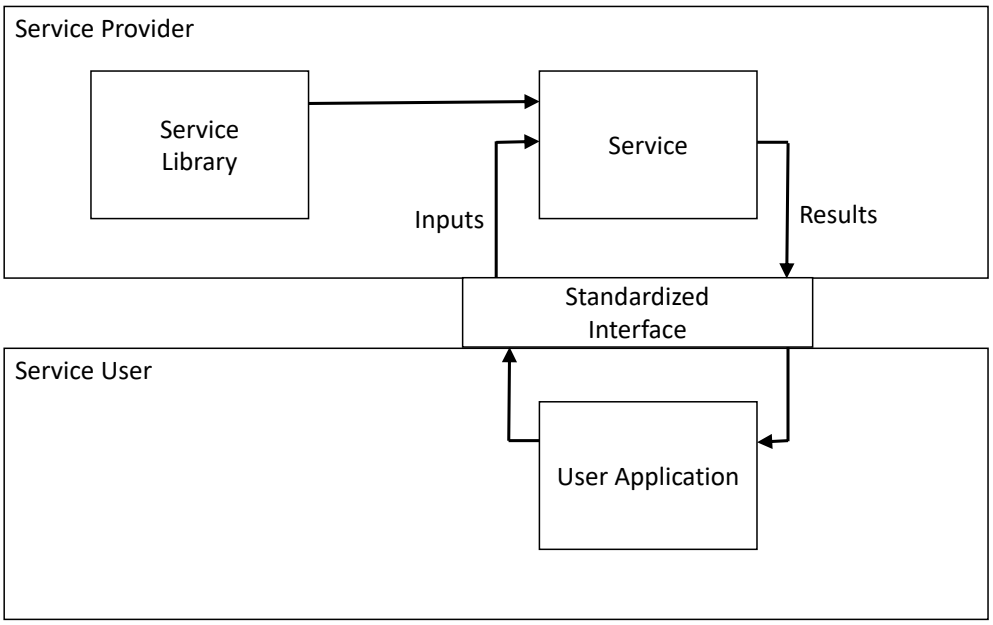


Figure 2.7: Service-Oriented Architecture

- a SOA is a paradigm for exchange of value between independently acting participants
- participants (and stakeholders in general) have legitimate claims of ownership of resources that are made available within the SOA ecosystem
- the behavior and performance of the participants are subject to rules of engagement which are captured in a series of policies and contracts

In the following, we explain basic SOA related terms, which are presented in [OAS].

SOA Ecosystem. To understand the interactions between participants it is not sufficient to look at a decomposition of the system. It is important to understand the context of the system functions and the contributions of the participants to the overall system. These interactions are considered in the *SOA ecosystem*. According to the OASIS reference model for SOA [OAS], SOA based systems assume the following interactions:

- use of resources that are distributed across ownership boundaries
- people and systems interacting with each other, also across ownership boundaries
- security, management and governance that are similarly distributed across ownership boundaries
- interaction between people and systems that is primarily through the exchange of messages with reliability that is appropriate for the intended uses and purposes

There are different participants that interact with each other and the system in a SOA ecosystem [OAS]. A *provider* offers a service that can be used by others. The *consumer* has a need and interacts with a service to fulfill this need. To facilitate the interaction and connectivity between participants and services, a *mediator* helps to offer services for use. Lastly, the *owner* claims the ownership over a service. To own a resource implies taking responsibility for creating, maintaining and provisioning the resource.

The participants in a SOA ecosystem interact with each other to achieve their own goals. The interactions are determined by the individual goals and the provided functionality. The following terms are used to define the interactions between participants. A *need* is a general statement that expresses something that is deemed necessary. A *requirement* is a formal statement of a desired result as real world effect. When a requirement is fulfilled, it satisfies a need. A *capability* is the ability to deliver a real world effect. Lastly, a *real world effect* is a measurable change to the shared state of pertinent entities that are relevant to and experienced by participants.

The typical approach to develop a new service can be described as follows:

1. A participant expresses a need. This can be a user that requires a specified real world effect or even a provider that intends to extend the amount of accessible functionality.
2. Designers and developers formalize the need as requirement.
3. A service with the capability to achieve the requirement is created.
4. The service can be accessed to deliver a real world effect. Mediators create service registries to facilitate the access of services and enable easy access by users.

The idea of a service in a SOA ecosystem combines business functionality with implementation, including the artifacts needed and made available as IT resources. By defining design rules that consider the use of *Services* in larger business tasks, the reuse of solutions can be facilitated. A system consists of different individuals, which intend to fulfill individual goals. These individuals act in the same environment, interact with each other and share resources to obtain their goals.

Services are provided in a *Service Library*, where a user can choose a service that fulfills the desired task. The interface of the service is given to the user. A user needs to adapt an application to access the interface of the service that is given by input parameters.

Service Quality Measures. In the following, we present means that measure service design quality, which are taken from [PV06]. These measures are *coupling*, *cohesion* and *granularity*.

Coupling. The coupling of systems determines the independence between their processes. By minimizing the coupling, the systems can be analyzed independent from each other. Furthermore, self-contained services contain high independence of

other system parts and, therefore, provide low coupling. A benefit of low coupling is that the amount of service redundancy and duplication in the system can be reduced. This aims to develop self-contained services that do not rely on knowledge of other services that are used by the system.

Cohesion. The degree of functional relatedness of operations within a service is measured as *cohesion*. High cohesion means that a system or service itself uses services that are highly related and have related responsibilities.

Granularity. The scope of functionality that is exposed by a system is described by its *granularity*. A fine-grained service provides a small amount of usefulness for the overall system. Whereas larger granularities provide compositions of smaller grained components.

2.7 Summary

In this chapter, we have presented background information to the topics that are used in the remainder of this thesis. We introduced model-driven development and its application to hybrid systems. Furthermore, we have introduced hybrid systems and ways to model their behavior. Afterwards, we gave an excerpt of verification techniques. We presented Simulink as a modeling language and design-tool that enables MDD for hybrid systems. Furthermore, we presented Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$) as a formal representation of hybrid systems. Thereafter, we presented the paradigm of service-oriented design, which we adapt to our design and verification approach.

3 Related Work

In this chapter, we discuss related work of this thesis. In our approach, we consider the verification and design of hybrid systems. This chapter contains the topics that are relevant for this thesis and is structured as follows: First, we discuss work on verification of hybrid systems. Therefore, we take a look at different representations of hybrid systems like timed automata, hybrid automata and logic-based approaches. Second, we summarize approaches that formalize Simulink models. We present other work in the area of the verification of Simulink models. We discuss approaches that consider a limited subset of the Simulink block library, approaches that consider hybrid system models and approaches that enable compositional verification. Third, we discuss approaches that introduce wider means to introduce reuse and variability in Simulink. Therefore, we summarize approaches that perform clone detection for Simulink models and other approaches that introduce variability capabilities.

3.1 Verification of Hybrid Systems

The models of hybrid systems capture the discrete state changes of the system as well as its continuous state evolutions. These two behavior types and their interactions need to be considered in the verification of these systems. Hybrid automata are a commonly used representation of these systems and there are many approaches that perform verification on them [Alu+93, CK99, Hen00, FHK04, Roe+16, RPV17] and it is still an ongoing research topic. Hybrid automata have gained wide attention in the hybrid systems community, and many verification approaches for hybrid systems have been based upon this formalism. It is known that the general reachability problem of hybrid automata is undecidable [Hen+98]. However, since their application in safety-critical areas require guarantees about their behavior, different approaches are developed to verify parts or approximations of their behavior. Therefore, verification approaches consider only a decidable subset of the hybrid behavior [HPR94, HHW95, Fre05] or perform over-approximations of the system behavior to determine the reachable states [CK99, RPV16].

A widely researched topic of hybrid automata are systems that only contain linear dynamics [HPR94, Fre05]. For systems that contain more complex behavior, approaches that over-approximate the continuous dynamics were developed [Roe+16, RPV17]. Besides hybrid automata, there are other representations of hybrid systems [Pla08]. Some of these introduce other methods in the verification of hybrid systems than reachability analysis. In [Pla08], the authors use deductive verification to prove that a system with hybrid behavior fulfills safety properties.

In the following, we discuss a special decidable subclass of hybrid automata, namely timed automata. Thereafter, we summarize approaches that perform reachability analysis on a wider application of hybrid automata. Lastly, we discuss logic-based approaches that perform deductive verification to prove the system behavior.

Timed Automata. As special case of hybrid systems, *Timed Automata* [AD94, BCM16, And19] are used to model continuous time and discrete state changes. They extend finite automata by clocks that evolve continuously and can be reset.

In contrast to general hybrid systems, they only model time as continuous value and not data. Due to this restriction, the reachability problem of timed automata becomes decidable [AD94]. The verification of timed automata is still present in current research [BCM16, And19].

UPPAAL [Ben+95, Dav+15, Eri+17] is a tool suite for the design and verification of real-time systems. The systems are modeled as networks of interacting timed automata. An integrated model checker is used to verify system properties. The properties are written in CTL-like expressions. The UPPAAL model checker is still being improved and allows for the modeling and verification of, e.g., stochastic timed automata [Dav+15]. However, the limiting factor in the use of timed automata is that they are restricted in the type of continuous variables that they use. Only clock variables can evolve continuously and only with a gain of 1. It is not possible to introduce continuous data variables.

Priced Timed Automata, an extension of UPPAAL, allows for using other rates for clock variables [Bul+12]. However, the resulting models represent linear hybrid automata, which are not decidable [Hen+98]. The resulting verification is done via stochastic model checking, which provides probabilities on safe behavior but cannot ensure that the system always behaves in such a way.

Hybrid Automata. A widely used formalism for the description and verification of hybrid systems are hybrid automata [Alu+93, CK99, Hen00, FHK04, Roe+16, RPV17]. Hybrid automata consider more continuous values than just time. The general idea in the verification of hybrid automata is to determine the reachable set of states. The continuous dynamics make this a challenging task since real valued variables possibly produce infinite reachable states. To determine the reachable continuous states, the continuous dynamics need to be used in their calculation. Since this is a challenging task, most approaches over-approximate the continuous evolutions.

In [FHK04] the tool PHAVer is presented, which enables an assume-guarantee reasoning based on simulation relations for hybrid systems. The systems are modeled as variation of hybrid input/output automata [Lyn+95], which extend hybrid automata by variables that represent inputs and outputs. The assume-guarantee reasoning enables compositional reasoning for these systems. The simulation relations are over-approximations of the underlying hybrid behavior and do not consider the interacting continuous dynamics. This can restrict the applicability, where the dynamics of the inputs are necessary to determine the behavior of the outputs.

In [Fre+11], the tool SpaceEx is presented. It contains a reachability algorithm for hybrid systems with piecewise affine, non-deterministic dynamics. An over-approximation of reachable states is computed with polyhedra and support functions. To achieve a good accuracy, variable step sizes for time between calculations are used in the computation. In [Bog+14], this approach is extended by an abstraction via location merging. Locations in the hybrid system representation are merged and replaced by an abstract location that represents the convex hull of the behavior of the original locations.

The authors of [Ben+14] present the tool Ariadne, which uses assume-guarantee reasoning for the verification of hybrid systems with nonlinear dynamics. The sys-

tem is composed into components and if these components fulfill given contracts, properties of the system can be proven. The reachability analysis of the contracts is done via an over-approximation and discretization of the reachable state space. The numerical approximation of the state space affects the performance and limits the number of continuous variables.

Logic-based approaches. Differential Dynamic Logic $\mathsf{d}\mathcal{L}$ [Pla08, Mül+16, Mül+17, Mül17] is a formal representation to model hybrid systems and enables their verification. The foundations of $\mathsf{d}\mathcal{L}$ are discussed in Section 2.5. In the following, we introduce research work about the verification of $\mathsf{d}\mathcal{L}$ models.

In [Mül+16, Mül+17, Mül17], the authors present a contract-based approach for the verification of hybrid systems. System components together with their contracts are specified in $\mathsf{d}\mathcal{L}$ and are then semi-automatically verified using the interactive theorem prover KeYmaera X. By providing deductive verification techniques, KeYmaera X provides a promising approach that scales better than model checking based approaches for many systems. In particular, by arguing with invariants as an abstraction of the complete system behavior, KeYmaera X has the potential to avoid an exhaustive exploration of the state space.

A system is modeled as a component with an interface and a contract. The authors split a component into a control part, a plant part and assignments that connect the ports of inner sub-components. The control part contains all possible operation modes of the system and the selection of one mode and the plant part contains all continuous evolutions according to the current operation mode.

However, the system design and the contracts must be provided as a formal model in $\mathsf{d}\mathcal{L}$, which typically requires a high expertise as well as a high manual effort. In practice, complex systems are often developed in languages like Simulink, where the semantics is informally defined. This impedes the verification process and requires either a manual development of a formal model or a transformation into a formally well-defined language.

3.2 Analysis and Verification of Simulink Models

Simulink [Matb] enables the modeling of hybrid control systems. The modeling language of Simulink does not provide formal semantics as is. Despite that shortcoming, the widespread use of Simulink makes its verification an interesting research topic [RS11, FP13, HRB13, AER14, RG14, MF16].

The MathWorks, Inc. provides the Simulink Design Verifier [Mat08], which provides model checking as well as abstract interpretation techniques. To enable verification with the Design Verifier, the specification has to be expressed as part of the model. However, the Design Verifier is only applicable for time-discrete Simulink models, and its scalability is limited [HRB13], as it is based on an exploration of exponential state spaces.

In the following, we discuss approaches for the analysis of the behavior of Simulink models. Each approach is tailored to capture specific behavior in the formal representation. First, we take a look at approaches that verify properties for Simulink models that only consider discrete behavior. Second, we introduce approaches for

the verification of Simulink models with hybrid behavior. Lastly, we look at approaches that provide contracts for Simulink components to enable compositional verification.

Formal Semantics. In [BC12], the authors define a formal Simulink semantics. However, their formalization does not provide an automated formalization of given models but formalizes the execution semantics of Simulink. Furthermore, it is not connected to any existing formal verification tool.

Simulation extensions. In [SCN13], a toolbox for hybrid equations in Simulink is presented. However, it only extends the simulation capabilities and does not enable comprehensive verification.

Verification of discrete behavior. The basic blocks of Simulink enable to model complex systems. The interaction between different blocks can be used to model different kinds of behavior, like hybrid behavior by using continuous blocks that interact with discrete blocks. Limiting the possible blocks during system design can simplify the analysis of the system behavior. A widely researched domain is the formalization and verification of the subset of Simulink blocks that contain discrete behavior.

In [AER14], the authors present a transformation of Simulink models into Why3 [FP13] to enable deductive verification. In [HRB13], Simulink models are transformed into the UCLID verification system [LS04], and thus the approach enables Satisfiability Modulo Theory (SMT) solving for the verification of safety properties. In [RG14], Boogie [Bar+05] is used as formal representation, and the SMT solver Z3 [DB08] for formal verification. However, all of these approaches only consider a discrete subset of Simulink and are not applicable for hybrid systems.

Verification of hybrid behavior. In [CK03], the tool CheckMate for modeling and verification of hybrid automata in Simulink is presented. The authors provide special blocks to model and verify polyhedral invariant hybrid automata (PIHA). However, this approach can only be applied for a special class of hybrid systems and requires the use of specialized blocks. All other blocks that model continuous behavior, like the *Integrator* block, are not considered in the verification. Thus, it is not applicable for most industrial Simulink models.

Related to our approach is the approach presented in [MF16], where a transformation from Simulink into a specific hybrid automata dialect (SpaceEx) is proposed. This enables the use of reachability algorithms for hybrid automata. However, concurrency is modeled using parallel composition of hybrid automata, so the state space is exponential in the number of concurrent blocks.

In [Zou+15, Che+17], the authors present the tool MARS for the verification of Simulink/Stateflow models. The tool transforms Simulink models with Stateflow parts into Hybrid CSP and enables the verification in the Hybrid Hoare Logic Prover. The use of Hybrid CSP enables high expressiveness and compositionality. However, the property specification and the verification in Hybrid CSP requires a very high level of expertise.

In the work of [Bou+17], a synchronous language that incorporates the use of ordinary differential equations is presented. The authors present how different Simulink blocks can be expressed in the Zelus language. However, the authors do not provide means to verify the modeled systems.

Statistical model checking. There exist approaches that use statistical model checking to analyze the behavior of stochastic Simulink models. These models contain transitions that contain probability distributions that determine whether the transition is taken. The approaches provide probabilities how likely the system fulfills a given property. In [Bar+18], the authors combine their formalization of Simulink models with Petri nets to use them as input for the statistical model checker Cosmos [Bal+15]. The authors of [Fil+16] propose a transformation of Simulink blocks into a network of stochastic timed automata. These are used as input for statistical model checking with UPPAAL SMC [Dav+15]. However, statistical model checking only provides probabilities that the system behaves correct and is not able to guarantee correct behavior.

Generated code. The tool CLawZ [OHa13] enables to automatically prove code that is provided by code generation of a Simulink model. The tool produces a formal model of a discrete Simulink model in Z notation and checks whether the code that is generated from the Simulink model is a refinement of the formal representation. However, this approach does not check the behavior of the initial Simulink model and only considers discrete system behavior.

In [Ber+18], the BTC EmbeddedValidator is used to perform bounded model checking on C-code that is generated from Simulink models. It can be used to find program errors that violate requirements and these errors are traced back to the original Simulink model. However, due to the nature of bounded model checking it is difficult to ensure correct behavior of the system outside of the chosen bounds.

Contract-based verification. The authors of [RS11] present a contract system for signal types in Simulink. By inferring the types of signal lines by its source ports, formulas are created that apply types to each signal line and input port. This approach is only able to check that the type of block outputs matches the connected blocks and no other checks are performed. In [Bos11], an approach for contracts for Simulink is presented. The authors transform Simulink models into Synchronous Data Flow graphs that are mapped to sequential program statements. This representation is then used to verify that a Simulink model fulfills a given contract. The use of contracts increases the scalability of the verification. However, only the discrete subset of Simulink blocks can be used in this approach.

3.3 Reuse and Variability in Simulink

In industrial applications, Simulink models can get large and can contain multiple interacting components. To increase the maintainability and reduce the development effort, approaches for clone detection and reuse have been developed [Pha+09, Dei+10, ASH11, Ala+12, Hab+13, Ala+14].

In the following, we introduce research work that introduces reuse and variability in data flow oriented models, namely models in Simulink. First, we introduce clone detection approaches. Second, we discuss approaches that introduce variability into the design of Simulink models.

Clone Detection in Simulink. The presence of clones in models or code increases the model and program size and reduces the maintainability. Clones are identical or similar sub-graphs in the same model. *Clone detection* aims at finding these duplicated sections in code or models to prevent inconsistent changes in the system and to increase the maintainability. In the following, we discuss clone detection in more detail for graph-based models in Simulink.

In graph-based modeling languages, the general idea of clone detection [Pha+09, Dei+10] is to find isomorphic parts in one graph or check whether two different graphs are isomorphic. To extend the applicability of clone detection, the authors in [Ala+12] extend the clone detection to not only find exact clones but also *renamed* and *near-miss* clones. Renamed clones are graphs that contain the same structure to each other but with nodes are named differently. Near-miss clones extend this by considering graphs that have a similar structure to one another, but some nodes and edges are removed or additional ones are added. These approaches all consider the syntax level of the model. The approach in [ASH11] considers semantic clones in Simulink. After a normalization of block structures in the system, parts that provide the same semantic behavior can be identified.

Clone detection approaches find clones in existing models and can provide a basic understanding, which model parts should be encapsulated into reusable components. However, they do not provide much information about the behavior that is given by the clones and how they can be reused in other systems.

Variability in Simulink. To facilitate the reuse of a model, variability can be used to allow for the customization of the model for a specific application context. Variability means that a given model can be changed according to defined rules to adapt it to a new context. The approach of software variability management for Simulink models [Ala+14] introduces variability operators for Simulink. With a combination of clone detection and these operators, an identification of variations in subsystems is provided. Different clones are represented as Simulink *Variant Subsystems* that contain the application of variability operators. The operators define the possible changes to a model and consider the following aspects of a model: the insertion or removal of blocks, the change of block inputs and outputs, block type changes, layout changes and renaming. While clones can be created automatically, the structural changes provided by Variant Subsystems is limited. All instances of a Variant Subsystem must have the same number inputs and outputs. That means, no changes on the interface of the component are possible.

Another variability approach [Hab+13] uses *deltas* to model variability. It provides general core systems and variability is provided by the application of delta operators. The possible deltas are the addition and removal of blocks, ports or signal lines, the modification of blocks, and the replacement of blocks.

While these approaches provide variability in Simulink, they are limited by the provided variability operators. Furthermore, they provide no means to describe how the changes influence the behavior of input and output signals.

3.4 Summary

The design and verification of hybrid systems have seen much research work. However, each approach comes with its own limitations. The verification of hybrid automata is mostly limited by its scalability. Hybrid automata are verified via model checking and over-approximations to determine the reachable state space. Furthermore, the verification is often closely connected to the system representation. Approaches for the formal verification of Simulink models are often restricted to the discrete block set and are not able to handle models with hybrid behavior. There are some approaches that support the verification of Simulink models that contain hybrid behavior. However, approaches that transform Simulink models into hybrid automata suffer from scalability issues and approaches that use specialized blocks to model hybrid behavior are limited in expressiveness compared to Simulink models used in industry. The approaches for integrating variability into Simulink do not consider the verification of the resulting components.

In this thesis we aim to provide a design and verification methodology for hybrid systems. With our approach, we enable deductive and thus potentially scalable verification of hybrid control systems modeled in Simulink with the hybrid theorem prover KeYmaera X.

4 Service-Oriented Design and Verification Approach

Cyber physical control systems are often employed in safety-critical areas and fulfill a wide range of tasks. They find application, for example, in the automotive and avionics industry and in the medical context. With better and faster processing, the desired functionality gets extended. To handle the increasing complexity of these systems, model-driven design processes are utilized. A major challenge in the design of these systems is to ensure their correct behavior, especially when they exhibit hybrid behavior.

To enable the formal verification of hybrid systems that combine continuous and discrete behavior and to facilitate their design, we propose a service-oriented design methodology. Our approach enables to model hybrid systems in a commonly used modeling language, enables the verification of these models and facilitates the modeling of further systems by reusing previous results in the design process of new models. We target the industrially widely used modeling language Simulink. Simulink is a data flow oriented language that supports modeling and simulation of hybrid systems.

In this thesis, we present an approach for the service-oriented design and verification of hybrid control systems. The key ideas of our approach are threefold: First, we provide a formalization of Simulink models into $\mathsf{d}\mathcal{L}$. This includes the automatic transformation of Simulink models into a formal representation and allows for the use of the interactive theorem prover for hybrid systems KeYmaera X. Second, we provide a formal description of the input-output relation of Simulink models in the form of *hybrid contracts*. Hybrid contracts provide a formal description of the interface behavior of a *Simulink* model. With our Simulink to $\mathsf{d}\mathcal{L}$ transformation and KeYmaera X, we can semi-automatically verify that a model fulfills its hybrid contract for all possible input scenarios that fulfill the assumptions of the contract. Furthermore, we propose a compositional verification approach for systems that contain components with hybrid contracts. Third, we provide *services* in Simulink to facilitate the reuse of verified components in further models. Services encapsulate a Simulink model, hybrid contracts and a feature model to adapt the behavior of the service. This increases the flexibility of our approach by enabling the customization of services for different environments and still enabling the reuse of the provided hybrid contracts. An implementation of major parts of our approach is available as open source project[2].

In this chapter, we firstly discuss the limitations and assumptions of our approach. Thereafter, we give an overview of our service-oriented design and verification approach.

4.1 Assumptions and Limitations

Our approach requires that the models under verification satisfy some assumptions. In the following, we first discuss assumptions that are due to our methodology. Thereafter, we discuss limitations that arise due to the current state of our automatic transformation.

[2]Project available at https://github.com/EmbSys-WWU/Simulink2dL

Conceptual Limitations. We require that a given Simulink model fulfills the following assumptions for our design and verification approach. Since we aim to provide a system designer with an approach to create verifiable services, these assumptions limit the Simulink block set that a designer can use during the creation of a Simulink service.

1. Limited use of algebraic loops.
2. No usage of S-Function blocks.
3. No usage of external scripts or libraries.

The first point considers loops in the design that only consist of *feed-through* blocks, which do not have an internal state. Note that feedback loops are generally allowed in the system design if they contain at least one stateful block. In Simulink, the use of algebraic loops is generally discouraged. Furthermore, they can only be simulated if the Simulink solver can find the solution of the underlying equations, therefore this is not a strong limitation. In our approach, we use a tool to solve the equations that are created by the algebraic loop. Therefore, we also require that the underlying equations of algebraic loops have a solution. This restriction is not severe, since only algebraic loops that have a solution can be simulated in Simulink. The second and third point consider Simulink blocks that can exhibit a very broad spectrum of behavior. S-Functions enable the use of system functions, which can be written in various programming languages. To include S-Functions and external scripts in our formalization of Simulink models, a transformation of every usable programming language for these blocks would be required. If they are used in a system, S-Functions can only be over-approximated in our transformation by assuming that the output could be every possible value at every time. This reduces the accuracy of the transformed system. The same considerations hold for external scripts and libraries. External libraries provide new blocks with custom behavior. It would be possible to define new individual transformation rules for each block of an external library.

Currently supported block set. Our transformation provides transformation rules for each block type individually. A major benefit of this approach is that the transformation can be extended for new block types without changing the rules for existing block types. The implementation supports representatives of the most relevant block classes, namely arithmetic, logic, discrete, continuous, and control flow blocks.

4.2 Formalization and Verification Framework

The design of hybrid systems is often a challenging and time-intensive task. This is especially the case if the system model must fulfill given specifications under all circumstances. The usual design process starts with the selection of a modeling language. On the one hand, there are languages that are designed to enable an easy creation of system models and provide a wide range of features that industrial designers typically use. On the other hand, there are languages that aim at

providing a formal foundation to easily verify system properties. The acceptance of formal languages in industrial design processes is often limited, mainly because the designers perceive them as unintuitive or hard to comprehend and the tool support for formal languages is often limited.

In our approach, a design can be created using the industrially widely used modeling language Simulink and then can be verified by using the formally well-founded interactive theorem prover for hybrid systems KeYmaera X. To enable this, we provide a transformation of informally defined Simulink systems into a behavioral equivalent representation in the formal language $\mathsf{d}\mathcal{L}$. The Simulink design may contain input ports and output ports that represent its connection points to its environment. During the verification, the inputs are assumed to be arbitrary, this means that they could change at any time to any value. To facilitate the verification and to define the system behavior more precisely, the designer can enrich the model by specifying assumptions for the input signals in $\mathsf{d}\mathcal{L}$. These assumptions represent known behavior of the incoming signals, e.g., bounds for their values or changes of their values, or that their trajectory is a continuous function. The resulting proof provides guarantees for the internal behavior of the model and its outgoing signals. With the given assumptions for the input signals and the guarantees, we create contracts that capture the externally visible behavior of the model. To enable reuse and compositional verification, we encapsulate the verified contracts and Simulink model in a Simulink service. A service can contain multiple hybrid contracts that formally capture its behavior depending on its input signals. The service can be used as a reusable component in larger system designs. This enables compositional verification by replacing the inner block structure by its contracts in the verification of the whole system. To increase the flexibility and modularization of our approach, we extend the services by variability. A designer can create variations in the inner structure of the service and capture the different variants in a *feature model*. The behavior of variants is captured in hybrid contracts that represent the special behavior of the individual internal customizations. During the later reuse of the service, a designer can choose an appropriate variant of the service to include in his design and still use the provided contracts for the overall system verification.

Transformation. Our verification process starts with a Simulink design that is provided by the designer. Figure 4.1 shows our transformation approach. The transformation provides rules for each block in Simulink. A transformation rule consists of behavior that is included in the target $\mathsf{d}\mathcal{L}$ representation and a replacement macro. The added behavior can include variable declarations, continuous evolutions or discrete assignments. For some blocks, the behavior is empty. The replacement macro allows us to consider each block individually. A macro defines which terms in the target $\mathsf{d}\mathcal{L}$ should be replaced by a given hybrid program. Additional to the individual block rules, we provide transformation rules that capture the overall behavior of the Simulink solver. With these rules, we are able to model the simulation loop, variable step behavior and zero crossing behavior in the target representation. Our transformation of Simulink to $\mathsf{d}\mathcal{L}$ is published in [LHG18].

Hybrid Contracts. The resulting $\mathsf{d}\mathcal{L}$ representation can be extended by assumptions for the input signals and the system properties that should be shown. Fur-

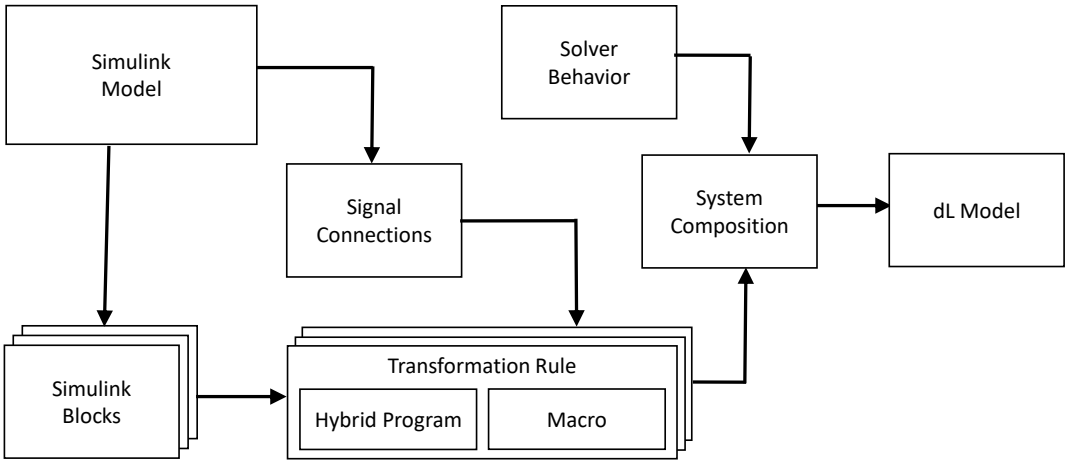


Figure 4.1: Transformation of a Simulink model

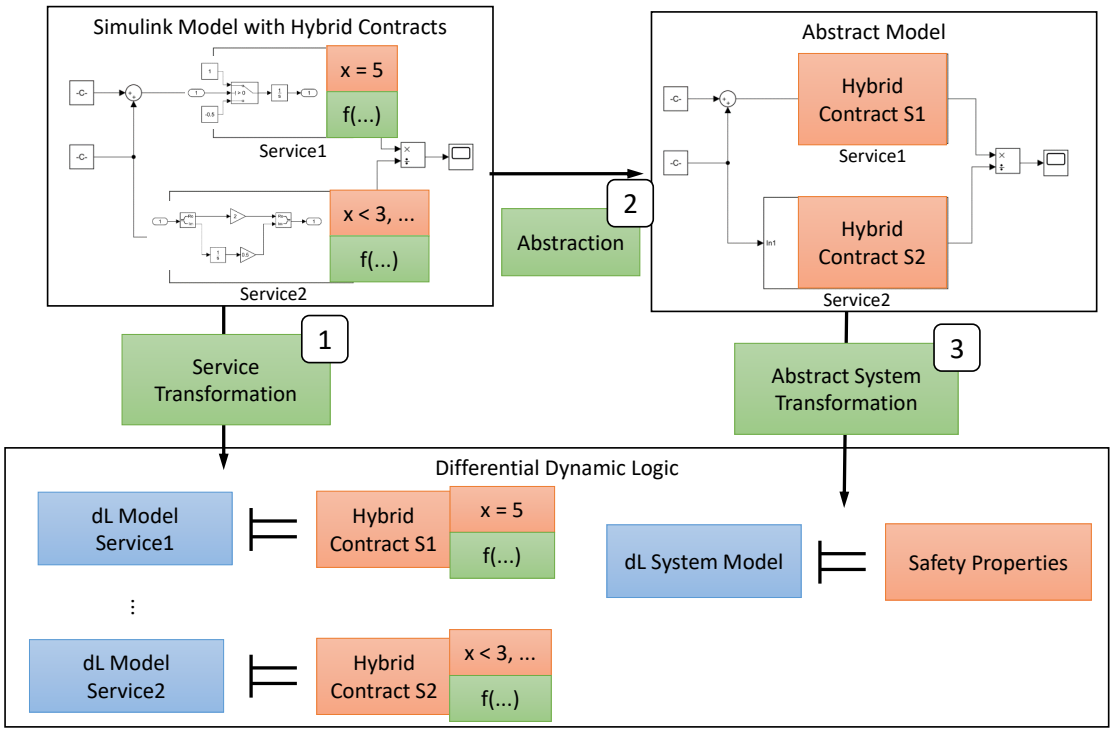


Figure 4.2: Compositional verification of a Simulink model

thermore, we provide extension functions for the resulting dℒ model that allow to insert observer variables and clocks in the system that can be used to model more complex system properties. We use the interactive theorem prover KeYmaera X to verify that the resulting dℒ model satisfies the contract. The verified properties are guaranteed under the condition that the assumptions for the inputs hold. We capture this assume-guarantee behavior in our hybrid contracts. The Simulink model and the hybrid contracts are encapsulated as Simulink services.

Compositional verification. The use of services in a larger system design allows us to abstract from its inner structure in the verification of the overall system. Figure 4.2 shows our compositional verification approach. First, a designer can

create a new Simulink design or apply our approach to an existing model. Second, the services are transformed individually into dℒ. Third, we can verify for each service that it fulfills its hybrid contracts. The second and third steps can be omitted if there are already verified contracts given for a service. Fourth, we replace the inner block structure of each service by its hybrid contracts in an abstract system transformation. The resulting system is an abstract dℒ version of the initial Simulink model. Fifth, we use the resulting dL model to prove system properties in the same way we have proven properties for individual services. We have published our compositional verification and hybrid contracts in [LHG19].

Variability. To increase the flexibility of our Simulink services, we extend our service representation by a feature model. The feature model captures different variants of the system and their dependencies. We adapt our hybrid contracts to consider the chosen features of a feature model. This enables us to create contracts that are specified more precisely by the features that are chosen. Our variability for Simulink services is published in [Lie+17] and [LHG21].

4.3 Summary

Overall, our framework facilitates the design process and enables the verification of hybrid control systems that are modeled in Simulink. We use hybrid programs in dℒ as formal representation and provide an automatic transformation from Simulink into dℒ. The interactive theorem prover KeYmaera X enables us to verify contracts for our systems, which we use in a compositional way to verify properties of larger systems. To increase the reuse of the verification results, we allow for variability of the underlying Simulink models. The different variants are also captured in the contracts. In the following, we introduce the different parts of our proposed approach in more detail. In Chapter 5, we introduce our transformation of Simulink models into dℒ. Chapter 6 presents our verification of transformed models and our hybrid contracts. In Chapter 7, we introduce our Simulink feature modeling. We evaluate our approach with four different case studies in Chapter 8, namely a temperature control system, a generic infusion pump, a model of a distance warner and a model of an autonomous robot in a factory.

5 Formalization of Simulink with dℒ

In this chapter, we present our formalization of Simulink. It is based on a transformation of informally defined Simulink models into a formally well-defined dℒ representation. Our transformation enables the formal verification of Simulink models with hybrid behavior and creates the foundation for our service-oriented design and verification approach. The semantics of Simulink is only informally defined. To enable the verification of Simulink models, we present a formalization by defining transformation rules that capture the behavior of Simulink models in the formal semantics of dℒ. This formalization considers the hybrid behavior of models.

The overall formalization approach is depicted in Figure 5.1. Our transformation automatically generates a dℒ model from a given Simulink model. The resulting dℒ model can be verified semi-automatically using the interactive theorem prover KeYmaera X. This spares the designer the tedious task of manually defining a formal model that captures the behavior described by the Simulink model and gives access to the powerful verification techniques that KeYmaera X provides.

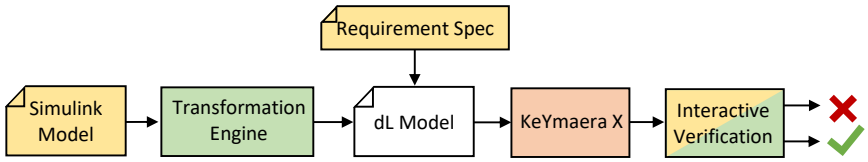


Figure 5.1: Simulink to dℒ approach

The key idea of our formalization of Simulink models is to model the behavior of a Simulink simulation loop step inside of a dℒ nondeterministic repetition. We define dℒ expressions that precisely capture the semantics of all Simulink blocks in a given model. We connect these expressions according to the signal lines and expand control conditions such that assignments and evaluations are only performed if the control conditions are satisfied. To achieve this, we define transformation rules that map the semantics of individual Simulink blocks to dℒ.

This chapter is based on [LHG18] where we have presented our formalization of Simulink models in dℒ. In Section 5.1, we introduce our simulation loop in dℒ and general transformation rules that consider the Simulink solver behavior. These rules are independent of the blocks in the system and are the same in the transformation of different systems. In Section 5.2, we provide individual transformation rules for different Simulink block types. In Section 5.3, we present rules to combine the transformation results of individual blocks to create the dℒ representation that contains the behavior of the interplay between all blocks in the system. Lastly in Section 5.4, we present optimizations that reduce the number of individual transformation rules in the system and that remove parts of the target dℒ representation that contain unreachable behavior.

5.1 Simulink Solver Behavior

In Simulink, blocks are executed in time steps and the signals are calculated for each step. The Simulink solver determines how the simulation is executed. It enables to choose between fixed step execution, where all time steps have the same

size, or variable time steps, where the size of time steps can change to obtain more precise calculations. In our approach, we aim at representing the variable time step behavior of Simulink into the dL model. This enables us to verify correct behavior for all possible step sizes that a variable time step solver can choose. To achieve this, we consider two parts. First, we capture the Simulink simulation loop behavior in our dL representation. Second, we model the Simulink zero-crossing behavior of continuous evolutions in our target representation.

General Simulation Loop

```
1  VariableDeclarations
2  Preconditions & Initializations ->
3  [ { ... }* ] Postcondition
```

Listing 5.1: Structure of a transformed model

To capture the combined behavior of a given hybrid control system modeled in Simulink, we introduce one global simulation loop, which is modeled as a *nondeterministic repetition* in dL, and contains both discrete assignments and continuous evolutions. An overview of the representation of the transformed system is given in Listing 5.1. The *Variable Declarations* contain all variables and constants that are used in the system. *Preconditions* are derived from the system requirements. In the *Initializations* section, initial values are assigned to variables and constants. The global simulation loop comprises the transformed system behavior, namely time-discrete behavior, additional assignments, e.g., assignments to variables that are used to influence the control flow and the time-continuous behavior, and continuous evolutions. Lastly, the *Postcondition* captures the properties that should be met according to the requirements specification. Note that in dL, whenever a loop is left, it may or may not be executed again. Whenever the loop may terminate, all verification goals must hold to ensure correct system behavior. Thus, we verify that the postconditions hold for all possible system runs.

Variable Step Size Simulation Loop

In our approach, we consider variable time step behavior. This behavior also includes the behavior of fixed time steps. We represent the Simulink solver behavior in a *simulation loop* that contains the calculations that are performed in one Simulink solver step. Our resulting dL model has the following form:

$$\phi_1 \rightarrow [(\alpha)^*](\phi_2)$$

Where ϕ_1 are initial conditions, ϕ_2 are the safety guarantees and α is a hybrid program that captures the behavior of all Simulink blocks during one simulation step. Simulink blocks are executed concurrently. To capture this behavior, we use the continuous evolution of dL to model the progress of time. We add all evolutions of continuous blocks to the continuous evolutions in dL to execute them in parallel. In each simulation loop step in our dL model, we execute exactly one continuous evolution. To capture the variable step behavior, we do not restrict the continuous evolutions by fixed time bounds. This enables us to employ the exit

behavior of continuous evolutions in dℒ to capture the behavior of variable time. Additionally, if a model contains discrete blocks, we add the constant *STEPSIZE* and the variable *stepTime* to the system. *STEPSIZE* contains the value for the discrete time step. The variable *stepTime* evolves with a constant derivative of 1 and is reset to zero after a time amount of *STEPSIZE*.

```
ProgramVariables.
R simulationTime.
...
End.

Problem.
% Initializations
simulationTime = 0 &
...
-> {[
    % Simulation Loop
    {
     % Discrete Assignments
     ...
     % Discrete Step Behavior
     { ?(steptime >= STEPSIZE);
       steptime := 0;
       ...
     ++
       ?(steptime < STEPSIZE);
     }
     % Continuous Behavior
     (simulationTime' = 1, steptime' = 1, ...
       & ( % Domain Restriction
           ...
           & steptime <= STEPSIZE));
    }*
    % End of Simulation Loop
   ]
   % Safety Properties
   (...)
  }
End.
```

Listing 5.2: Transformed Simulation Loop

Listing 5.2 shows the structure of our into dℒ transformed Simulink model. We introduce a variable *simulationTime* that represents the simulation time. Note that time only elapses during the continuous evolutions that are executed at the end of the simulation loop. In the first part of the simulation loop, discrete assignments are performed. Afterwards, the continuous evolutions are performed. During the continuous evolution, the variable that represents the simulation time evolves with a constant derivative of 1. Note that we also add the condition $steptime \leq STEPSIZE$ to the evolution domain of all continuous evolutions to ensure that discrete assignments take place each time the steptime elapses. Also note that the verification considers all possible behaviors that are produced by the system in dℒ. Therefore, this variable time step also includes the behavior of fixed time steps. We can split

the behavior of the simulation loop into two parts:

$$\phi_1 \rightarrow [(\beta;\gamma)^*](\phi_2)$$

In the first part β, only discrete assignments are performed. In the second part γ, continuous evolutions are performed. Note that in general γ is not one continuous evolution but a nondeterministic choice with test formulas that selects the evolutions according to the control flow of the model.

Discrete Jumps and Zero-Crossing Semantics

During the simulation in Simulink with a variable step time solver, if the result of a calculation indicates a zero-crossing that causes a change in the system behavior, e.g. the condition at a *Switch* changes, smaller time steps are used to find the best approximation of the time where the *Switch* block switches. In dL, the default continuous evolutions can evolve an arbitrary amount of time. To enable the detection of switch points in continuous evolutions, we provide two additional rules for conditional macros: First, we create a new nondeterministic choice for each continuous evolution. Second, we add the corresponding conditions to the evolution domain. Together, this ensures that the continuous evolutions can only evolve as long as this control flow does not change. We extend the continuous evolutions in our transformation with $smallStep \leq EPS$ to ensure that control flow changes are evaluated with a delay of at most a given ϵ (EPS).

```
...
-> [{
      ...
      % Discrete Assignments
      smallStep := 0;
      ...
      % Continuous Behavior
      { % Control Flow Selection A
        ?(...);
        (simulationTime' = 1, steptime' = 1, smallStep' = 1, ...
        & (( % Domain Restriction A
             ...
             & steptime <= STEPSIZE)
           | smallStep <= EPS));
        ++
        % Additional Control Flow Selections
        ...
     }
  }*
  % End of Simulation Loop
  ]
  % Safety Properties
  (...)
End.
```

Listing 5.3: Zero-Crossing Evolution

Listing 5.3 shows the extensions to the dL model. The variable *smallStep* is reset at the start of every Simulation loop. During continuous evolutions, it evolves with a

constant factor of 1. All evolutions can either evolve according to the given domain restriction or for a time that is restricted by *EPS*. With this extensions to the continuous evolutions, we are able to capture the Simulink zero-crossing behavior in dℒ. The constant *EPS* represents a small delay between the change of a value of a switch condition and the evaluation of the condition. The value of *EPS* can be set by the designer and should be the same as the precision parameter for the variable step size solver in Simulink. This represents the behavior in Simulink, where the times of changes in the control flow block, i.e., zero-crossing, are approximated and a small delay can occur due to the numerical nature of the Simulink solvers. In our representation, *EPS* is an upper bound for the delay. A continuous evolution in dℒ models all possible durations and therefore captures all possible small delays up to *EPS* in our target model. This enables us to verify correct behavior for all possible delays in control flow changes up to a time amount of *EPS*.

5.2 Transformation Rules

We introduce individual transformation rules for different block types in Simulink. Our approach for the transformation from Simulink to dℒ is twofold: First, we define transformation rules that map the semantics of individual Simulink blocks to dℒ. By defining transformation rules for each block separately, we can consider blocks and their calculations individually. This enables us to extend the transformation by new block types without the need to change existing transformation rules. Second, we compose the individual blocks into a dℒ representation that precisely captures the semantics of the original model.

In the following, we introduce the transformation of individual blocks into dℒ. The resulting dℒ programs are used to represent the model behavior in the simulation loop, which is described in Section 5.1. The blocks in Simulink are processed concurrently and they use the values that are provided at their input ports to update their output ports. The evaluation of blocks in Simulink is executed according to a given block order. Our macros respect this block order such that the evaluation of feed-through blocks is grouped together. This enables us to omit the introduction of individual variables for the results of these blocks. Our key idea to faithfully model the exact data, control and timing dependencies of the original model is to introduce discrete state variables for time-discrete blocks that keep an inner state, continuous evolutions to model time-continuous blocks, and to use a sophisticated macro mechanism to represent stateless behavior, e.g., port connections, arithmetic calculations, and, in particular, control flow.

Definition 5.1 (Transformation rule)**.** A *transformation rule* is a tuple (V, M, δ, E), where V is a set of variables, M is a set of replacement *macros*, δ is a hybrid program that describes the behavior of the block, which also can be *skip*, and E is a set of continuous evolutions $e \in E$ A single continuous evolutions $e = (v, \theta)$ consists of a variable v and terms θ that describe the continuous evolution of the variable. The set of continuous evolutions E can also be empty.

When a transformation rule is applied to our target hybrid program, all variables in V are added to the variable definitions in the dℒ model. To obtain unique names for these variables, we use the name of the Simulink block as variable name or

as prefix for the variable names, if the rule produces multiple variables. For the general structure of the hybrid program in the dL model, we use the previously defined form of our simulation loop $(\alpha)^* = (\beta;\gamma)^*$. The hybrid program δ is added to the discrete part of the simulation loop and we obtain:

$$(\alpha_{new})^* = (\beta_{new};\gamma_{new})^* = (\beta;\delta;\gamma_{new})^*$$

We obtain γ_{new} by extending all continuous evolutions that are present in γ by the evolutions that are given in E.

$$\gamma = \{x_1' = \eta_1,\ \ldots,\ x_n' = \eta_n\ \&\ \psi\}$$
$$\gamma_{new} = \{x_1' = \eta_1,\ \ldots,\ x_n' = \eta_n, v_1' = \theta_1,\ \ldots,\ v_m' = \theta_m\ \&\ \psi\}$$

Definition 5.2 (Replacement identifier)**.** A *replacement identifier* term id is a term that extends the term definition of dL. A replacement term id has no interpretation in dL and is only used as placeholder. To obtain a valid dL model, all replacement identifiers id are replaced by other dL terms. The identifiers for id are provided by the outputs of blocks in Simulink, i.e., each Simulink individual output of each block produces a unique identifier id.

Definition 5.3 (Macro)**.** A macro $m \in M$ consists of a replacement identifier id that is replaced by an expression e during the transformation process. For each output of each block, we provide a separate transformation rule. To denote that such a macro is applied to a hybrid program α, we write $\alpha[id \leftarrow e]$. The application of macros to hybrid programs is defined as follows:

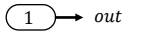
$$(\alpha;\beta)[id \leftarrow e] \overset{def}{=} \alpha[id \leftarrow e];\beta[id \leftarrow e]$$
$$(\alpha \cup \beta)[id \leftarrow e] \overset{def}{=} \alpha[id \leftarrow e] \cup \beta[id \leftarrow e]$$
$$\alpha^*[id \leftarrow e] \overset{def}{=} \alpha[id \leftarrow e]^*$$
$$(x := \theta)[id \leftarrow e] \overset{def}{=} x := (\theta[id \leftarrow e])$$
$$(x := *)[id \leftarrow e] \overset{def}{=} x := *$$
$$\{x_1' = \theta_1, ..., x_n' = \theta_n \& \phi\}[id \leftarrow e] \overset{def}{=} \{x_1' = (\theta_1[id \leftarrow e]), ..., x_n' = (\theta_n[id \leftarrow e]) \ \&(\phi[id \leftarrow e])\}$$
$$(?\phi)[id \leftarrow e] \overset{def}{=} (?\phi[id \leftarrow e])$$

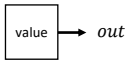
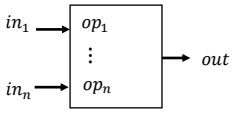
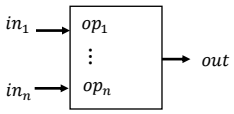
A macro is applied to a term θ or a formula ϕ by replacing the occurrences of the identifier id with the given expression e. The replacement for terms is defined as follows:

$$(x)[id \leftarrow e] \overset{def}{=} x$$
$$(f(\theta_1, ...\theta_k))[id \leftarrow e] \overset{def}{=} f((\theta_1)[id \leftarrow e], ...(\theta_k)[id \leftarrow e])$$
$$(\theta + \eta)[id \leftarrow e] \overset{def}{=} (\theta)[id \leftarrow e] + (\eta)[id \leftarrow e]$$
$$(\theta \cdot \eta)[id \leftarrow e] \overset{def}{=} (\theta)[id \leftarrow e] \cdot (\eta)[id \leftarrow e]$$
$$((\theta)')[id \leftarrow e] \overset{def}{=} (\theta)'$$
$$(id)[id \leftarrow e] \overset{def}{=} e$$

Table 5.1: Transformation Rules for Sources and Feed-Through Blocks

Simulink block			Macros and dℒ representation

sources	Inport		$V : \{input, IN\_MAX, IN\_MIN\}$
			$M : \{\alpha[out \leftarrow input]\}$
			δ : { `input := *;`
			`?(input <= IN_MAX & input >= IN_MIN);`
			}
			$E : \{\}$

	Constant		$V : \{\}$ $M : \{\alpha[out \leftarrow value]\}$ $\delta : \{\}$ $E : \{\}$
feed-through	Sum		$V : \{\}$ $M : \{\alpha[out \leftarrow 0\, op_1(in_1)op_2(in_2)...op_n(in_n)]\}$ $\delta : \{\}$ $E : \{\}$
	Product		$V : \{\}$ $M : \{\alpha[out \leftarrow 1\, op_1(in_1)op_2(in_2)...op_n(in_n)]\}$ $\delta : \{\}$ $E : \{\}$

The replacement for formulas is defined as follows:

$$
\begin{aligned}
(\theta \geq \eta)[id \leftarrow e] &\stackrel{def}{=} (\theta)[id \leftarrow e] \geq (\eta)[id \leftarrow e] \\
(p(\theta_1, ...\theta_k))[id \leftarrow e] &\stackrel{def}{=} p((\theta_1)[id \leftarrow e], ...(\theta_k)[id \leftarrow e]) \\
(\neg\phi)[id \leftarrow e] &\stackrel{def}{=} \neg(\phi)[id \leftarrow e] \\
(\phi \wedge \psi)[id \leftarrow e] &\stackrel{def}{=} (\phi)[id \leftarrow e] \wedge (\psi)[id \leftarrow e] \\
(\forall x\phi)[id \leftarrow e] &\stackrel{def}{=} \forall x(\phi)[id \leftarrow e] \\
(\exists x\phi)[id \leftarrow e] &\stackrel{def}{=} \exists x(\phi)[id \leftarrow e] \\
([\alpha]\phi)[id \leftarrow e] &\stackrel{def}{=} [(\alpha)[id \leftarrow e]](\phi)[id \leftarrow e] \\
(\langle\alpha\rangle\phi)[id \leftarrow e] &\stackrel{def}{=} \langle(\alpha)[id \leftarrow e]\rangle(\phi)[id \leftarrow e]
\end{aligned}
$$

With replacement macros M, a hybrid program δ and continuous evolutions E, we define transformation rules to capture the behavior of Simulink blocks in dℒ.

In the following, we define transformation rules for different block types. Note that *in* and *out* are identifier that are used in our macro replacement. The identifier *out* is unique for each block and in_i is the identifier of the output port of the block that is connected to the i-th input port. Furthermore, we use prefixing to obtain unique names for internal variables of the blocks, e.g., the *state* variables of time-discrete blocks. Each of these variables is also added to the dℒ model.

Sources

Sources provide values that can be used as inputs for other blocks. The transformation of a source assigns the provided value or variable to connected blocks.

Inports connect a system to an environment and provide incoming signals. Our transformation rule for Inports is shown in the first row of Table 5.1. There, we introduce a dL variable *input* for the provided signal, and define a macro that replaces all occurrences of the unique identifier assigned to the output port of a given Inport block (*out*) with *input*. The hybrid program δ that models an Inport block consists of two parts: First, to model arbitrary inputs, we use a *nondeterministic assignment*. Second, a *test formula* is added to the hybrid program, which defines the range of possible values. Initially, there are no values given for the upper and lower bound and they are set to the maximal and minimal value respectively. For the verification, these bounds can be used to define assumptions for the input values.

Constant blocks provide an assigned value to all connected blocks. This value does not change during the execution of the system. The transformation rule for this block generates a macro that replaces all occurrences of the connected port with the given constant value. The hybrid program of this block is empty.

Direct Feed-through Blocks

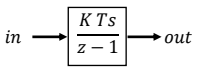
Direct feed-through blocks, e.g., arithmetic or logic blocks, do not have an inner state and write their results directly to their output ports. To model this, we create a macro that performs the operation defined by the semantics of a given block. As an example, the transformation rule of the *Sum* block is shown in the third row of Table 5.1. Note that all macros are fully expanded in the final dL model, that is, for the *Sum* block rule shown in the third row of Table 5.1, all occurrences of *out* will be replaced by the combination of all inputs in_i with the operators op_i defined by the parameters of the block (which might be '+' or '-' for a Sum block). The leading 0 is the neutral element of the addition and ensures that the replacement produces valid terms. The transformation of a product block is similar, but we use the neutral element of the multiplication (1) as first element in the resulting formula. Note that the input variables in_i are replaced by other expressions resulting from the transformation rules of the preceding blocks during the transformation process.

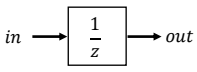
Time-Discrete Blocks

Blocks with time-discrete behavior, e.g. *Discrete Integrator* and *Unit Delay*, are blocks with an inner state that changes only at given time steps that are given by their sample time. The transformation rule for the *Discrete Integrator* block is shown in the first row of Table 5.2 and the transformation rule for the *Unit Delay* block is shown in the second row. Both blocks use an inner state to keep their values during the time step. At the start of a new time step, the *Unit Delay* takes the current input and stores it into its internal state and updates its output to the stored value of the previous time step. A Discrete Integrator takes the value

Table 5.2: Transformation Rules for Time-Discrete Blocks

Simulink block			**Macros and d$\mathcal{L}$ representation**

time-discrete	Discrete Integrator		V : {*state*}
			M : {α[*out* ← *state*]}
			δ : { `?(steptime >= STEPSIZE);`
			`state := state + in;`
			`++`
			`?(steptime < STEPSIZE);`
			}
			E : {}

	Unit Delay		V : {*output*, *state*0}
			M : {α[*out* ← *output*]}
			δ : { `?(steptime >= STEPSIZE);`
			`output := state;`
			`state := in;`
			`++`
			`?(steptime < STEPSIZE);`
			}
			E : {}

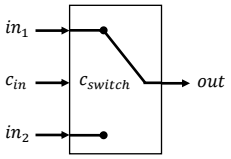
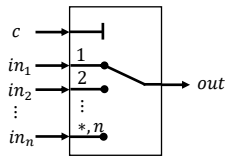
of the input signal at each new time step and adds it to its stored internal state. To capture this behavior, we introduce a variable `state` that represents the inner states of these blocks. Additionally, the *Unit Delay* block has an `output` variable that represents its output, while the *Discrete Integrator* directly outputs its inner `state`. To model discrete steps, we introduce a constant `STEPSIZE` that represents a given sample time and a continuous variable *steptime*. Discrete state variables are only updated if *steptime* is equal to `STEPSIZE`, otherwise no changes occur. To model time, we add *steptime* to the continuous evolution of the system with a derivative of 1. To consider each discrete step, we add `steptime <= STEPSIZE` to the evolution domain. We update all outputs of time-discrete blocks at the beginning of the evaluation of discrete steps. After all discrete assignments, we reset *steptime* to zero if `steptime >= STEPSIZE`.

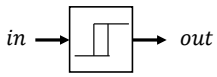
Control Flow Blocks

Control flow blocks, e.g. the *Switch* block and *Multiport Switch*, change the control flow of the system. This may create a discrete jump in time-continuous behavior. To transform control flow blocks, we introduce a new kind of macro, namely conditional macros. The idea of a conditional macro is that we make the macro mechanism dependent on control flow conditions. To this end, we first define an extended replacement function $\alpha[id \leftarrow e, c]$, which replaces id with e in a hybrid program α as above and additionally adds the condition c to all evolution domains in α. A *conditional macro* is given by $\alpha[id \Leftarrow CM]$, where id is the identifier that should be replaced and CM is a set of conditional replacements (e_i, c_i). The expansion of a conditional macro is defined as follows:

$$\begin{aligned}\alpha[id \Leftarrow CM] &= \alpha[id \Leftarrow \{(e_1, c_1), ..., (e_n, c_n)\}] \\ &= \texttt{\{ ?(}c_1\texttt{); } \alpha[id \leftarrow e_1, c_1]\texttt{; ++ ... ++ ?(}c_n\texttt{); } \alpha[id \leftarrow e_n, c_n]\texttt{; \}}\end{aligned}$$

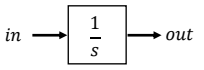
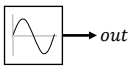
Table 5.3: Transformation Rules for Control Flow and Discontinuities

Simulink block			**Macros and dL representation**
control flow	Switch		V : {} M : {$\alpha[out \Leftarrow \{(in_1, c_{switch}), (in_2, \neg c_{switch})\}]$} = { { ?($c_{switch}$); $\alpha[out \leftarrow in_1, c_{switch}]$; ++ ?($\neg c_{switch}$); $\alpha[out \leftarrow in_2, \neg c_{switch}]$; } } δ : {} E : {}
	Multiport Switch		V : {} M : {$\alpha[out \Leftarrow \{(in_1, c == 1),$ $(in_2, c == 2), ..., (in_n, c == n)\}]$} = { { ?($c == 1$); $\alpha[out \leftarrow in_1, c == 1]$; ++ ?($c == 2$); $\alpha[out \leftarrow in_2, c == 2]$; ++ ... ++ ?($c == n$); $\alpha[out \leftarrow in_n, c == n]$ } } δ : {} E : {}

discontinuities	Relay		V : {$state$}
			M : {$\alpha[out \Leftarrow \{(state, in <= low),$
			$(state, in >= high),$
			$(state, in > low \ \& \ in < high)\}]$}
			δ : { ?(in <= low);
			state := off_output;
			++
			?(in >= high);
			state := on_output;
			++
			?(in > low & in < high);
			}
			E : {}

A conditional macro creates a nondeterministic choice where a hybrid program α that contains id is split into multiple cases (one case for each condition c_i). In each case with condition c_i, id is replaced by the corresponding e_i. We illustrate the use of conditional macros with the transformation rule for a *Switch* block in the first row of Table 5.3. The Switch has three input signals, namely a control input c_{in} and two data inputs in_1 and in_2, one output signal out, and an internal condition c_{switch}. If the control input c_{in} fulfills the condition c_{switch} the first data input in_1 is written to out, otherwise the second data input in_2 is written to out. Note that a Simulink Switch condition c_{switch} is of the form $c_{in} \sim C$ with $\sim \in \{>, \geq, \neq\}$ and C a constant Simulink expression. This concept to handle control flow can

Table 5.4: Transformation Rules for Time-Continuous Blocks

Simulink block			Macros and dℒ representation
time-continuous	Integrator		$V : \{s\}$ $M : \{\alpha[out \leftarrow s]\}$ $\delta : \{\}$ $E :$ { `s' = in` }
	Sine Wave		$V : \{x, y\}$ $M : \{\alpha[out \leftarrow x]\}$ $\delta : \{\}$ $E :$ { `x' = y, y' = -x` }

be easily adapted for other control flow blocks. Note that our conditional macro mechanism may introduce more cases than necessary. To increase the readability of the transformed program, our implementation of conditional macros only creates nondeterministic choices for assignments and evolutions where *id* actually occurs.

Discontinuities Blocks

Discontinuities blocks, e.g. the *Relay* block, can have sudden jumps in their output signals. To transform this behavior, we consider the special properties of these jumps. They are discrete changes of values and they occur whenever an input signal reaches a defined value. As an example, the *Relay* block has different parameters that define its behavior. When the input signal rises above a *high* value then the output value jumps to a specified *on_output* value. When the input signal falls below a *low* value then the output value jumps to a specified *off_output* value. Otherwise, the signal stays the same. Note that these values are real numbers and not variables. Therefore, only the state of the *Relay* is added as variable to the dℒ model. We use a nondeterministic choice with test formulas to model these discrete changes. The test formulas chose the value that is assigned to the internal state variable depending on the value of the input signal. The second part of the transformation of the *Relay* block behavior is the detection of changes of the behavior whenever the input signal reaches its bounds. To model the changes in the transformed dℒ, we use conditional macros. Note that all three cases use the internal *state* as replacement for the *out* identifier. The conditions cover the three states of the Relay block: First, the input is below the low value. Second, the input is above the high value. Third, the input is between the two values. Due to the conditional macro, the continuous evolutions of the dℒ model are split into these three areas depending on the value of the input signal. Since these areas are also part of the evolution domain of the dℒ model, we can detect when the signal crosses a border and a new simulation can be started to evaluate the discrete assignments.

Time-Continuous Blocks

To capture the concurrent execution of continuous Simulink blocks in our transformation, we combine the evolution of all state variables of all time-continuous

blocks into one continuous evolution. Note that this continuous evolution also contains a variable for the simulation time. Also note that the continuous evolutions may be split by conditional macros and each choice contains all continuous state variables. We illustrate our transformation rule for time-continuous blocks with the *Integrator* block and the *Sine Wave* block in Table 5.4 . The Integrator block takes the input signal *in* and integrates it over time. This means that it models the differential equation $s(t) = \int_0^t in(\tau)\, d\tau$, which is equivalent to $\frac{ds(t)}{dt} = in(t)$, where s is the inner state of the integrator. The *Sine Wave* is a source block that produces a signal that changes continuously over time. To model a sine wave signal, we use the representation of a sine wave as differential equations, whereas:

$$x' = y, y' = -x$$

The sine signal is given by x.

5.3 Model Transformation

In the previous section, we have defined transformation rules for individual Simulink blocks, including blocks with time-discrete and time-continuous behavior. In this section, we present our approach for the transformation of hybrid control systems that may consist of an arbitrary number of direct feed-through, time-discrete, control flow and time-continuous blocks into dL. The main challenge in combining the individual block transformation rules defined above is to precisely capture the interactions between blocks.

Transformation Algorithm

Our transformation algorithm analyzes a given Simulink model, applies the transformation rules defined above, and incrementally builds a hybrid program. During the transformation process, each block is translated into a set of macros, hybrid programs and continuous evolutions. Each transformation rule can add macros to the set of all macros used for the transformation, it can append its hybrid programs to the program β in the simulation loop and it can add evolutions to the continuous evolutions in γ. To ensure that all dependencies are correctly considered, we handle all time-discrete blocks in the correct order, i.e., we start with blocks that have no inputs and then successively handle all blocks where all input blocks have already been translated. Since direct feed-through and stateless blocks are transformed using our macro mechanism, the transformation is not dependent on the order of these blocks. When all blocks are translated, the macros are expanded, i.e., they are applied to the dL model according to the previously defined macro replacement rules. With our assumption that the system does not contain algebraic loops, each feedback loop in the original Simulink model contains at least one stateful block and this algorithm always terminates. Note that we flatten all subsystems at the beginning of the transformation and use prefixing to keep the structure of the original Simulink model transparent to the developer.

Transformation of a Temperature Control Model

The transformation of our example system (Figure 2.6) yields three simple macros and two conditional macros:

$$\begin{aligned} Integrator\_out \leftarrow & Integrator\_state \\ Sum\_out \leftarrow & Tdes\_out - Integrator\_out \\ Tdes\_out \leftarrow & Tdes \end{aligned}$$

$$\begin{aligned} Switch\_out \Leftarrow & \{(Heating\_out, Relay\_out > 0), (Cooling\_out, Relay\_out \leq 0)\} \\ Relay\_out \Leftarrow & \{(Relay\_state, Sum\_out \geq Relay\_max), \\ & (Relay\_state, Sum\_out \leq Relay\_min), \\ & (Relay\_state, Sum\_out > Relay\_min \wedge Sum\_out < Relay\_max)\} \end{aligned}$$

In this model, we have set the parameters for the Relay block as follows: The assignments for the output of the Relay block are 0 for the off state or 1 for the on state, depending on the input value. The jump to the on state occurs whenever the input value is larger than or equal to $0,5$ and the jump to the off state occurs whenever the input value is less than or equal to $-0,5$.

```
1  {smallStep:=0.0; OutPort1:=Integrator; Heating:=*; Cooling:=*;
2    {?(Tdes-Integrator>=0.5); Relay:=1.0;
3      ++
4      ?(Tdes-Integrator<=-0.5); Relay:=0.0;
5      ++
6      ?((Tdes-Integrator<0.5) & (Tdes-Integrator>-0.5));
7    }{?((Tdes-Integrator>=0.5) & (Relay>0.0));
8      {simTime' = 1.0, Integrator' = Heating, smallStep' = 1.0
9        & ((Tdes-Integrator>=0.5) & (Relay>0.0)) | (smallStep<=EPS)}
10     ++
11     ?((Tdes-Integrator>=0.5) & (Relay<=0.0));
12     {..., Integrator' = Cooling, ...
13       & ((Tdes-Integrator>=0.5) & (Relay<=0.0))|...}
14     ++
15     ?((Tdes-Integrator<=-0.5) & (Relay>0.0));
16     {..., Integrator' = Heating, ...
17       & ((Tdes-Integrator<=-0.5) & (Relay>0.0))|...}
18     ++
19     ?((Tdes-Integrator<=-0.5) & (Relay<=0.0));
20     {..., Integrator' = Cooling, ...
21       & ((Tdes-Integrator<=-0.5) & (Relay<=0.0))|...}
22     ++
23     ?((Tdes-Integrator<0.5) & (Tdes-Integrator>-0.5) & (Relay>0.0));
24     {..., Integrator' = Heating, ...
25       & ((Tdes-Integrator<0.5) & (Tdes-Integrator>-0.5) & (Relay>0.0))|...}
26     ++
27     ?((Tdes-Integrator<0.5) & (Tdes-Integrator>-0.5) & (Relay<=0.0));
28     {.., Integrator' = Cooling, ...
29       & ((Tdes-Integrator<0.5) & (Tdes-Integrator>-0.5) & (Relay<=0.0))|...}
30 }}*
```

Listing 5.4: Temperature Control System in dℒ

The hybrid program in dL (without initial conditions, variable declarations and safety guarantees) is shown in Listing 5.4. For brevity and simplicity of presentation, we omit prefixing and refer to the internal state of stateful blocks and to the output of stateless blocks with the block name (i.e., we use *Integrator* for *Integrator_s* and *Heating* for *Heating_out*).

The only discrete assignments (Lines 2 - 6) are generated for the *Relay* block, whose internal state is set to 1.0 or 0.0 depending on the deviation of the current temperature from the desired value of 19.0 degree celsius. For the continuous evolutions (Lines 7 - 29), we distinguish all cases where the switching behavior or relay behavior of the system changes. We use the corresponding conditions both as conditions in the evolution and as evolution domain. This ensures that whenever a switching condition or relay condition changes, the simulation loop is restarted and all conditions are newly evaluated. In each case, we have three continuous evolutions (e.g. Line 8): the simulation time *simTime* and the *smallStep* time evolve with a gradient of 1, and the *Integrator* evolves with *Heating* or *Cooling*, depending on the current control flow conditions.

5.4 Optimizations

To reduce the size of the resulting dL representation and to reduce the amount of macros that are necessary during the transformation, we have developed optimizations for the transformation. We have developed these optimizations for the transformation of groups of blocks with similar functionality and algebraic loops as part of a Bachelor's thesis [Won18].

Deletion of Unreachable Branches

The conditional macros can produce branches that are unreachable. During the evaluation of conditional macros, we check the resulting conditions. A conditional macro creates a branch in the dL behavior that contains a set of test formulas that define the prerequisites to execute this branch. We create a conjunction of all test formulas that guard a branch and also add the condition of the conditional macro. Note that this condition will be added as new test formula when the conditional macro is applied to the hybrid program. Whenever the resulting conjunction cannot be fulfilled, we omit the branch from the resulting dL model. The check for satisfiability can be automatically performed by an SMT solver, e.g., Z3 [DB08]. Note that we only omit the branch if its unsatisfiability is proven. If the result is not decidable, we add the branch to the dL model.

Furthermore, some conditional branches are directly connected with the assignment of specific values to variables. E.g., the Relay block has two cases, in which an assignment to its output takes place. For these conditional branches with assignments, we add the assignments as additional *invisible conditions* to the conditional macros. These *invisible conditions* are not added to the model, but used during the checks for valid branches.

Running example. In our transformed temperature control system in Listing 5.4, we have six different branches that are obtained by one Switch and one Relay. In

the cases that the Relay is turned on (`Tdes-Integrator >= 0.5`) and that it is turned off (`Tdes-Integrator <= -0.5`), its output is set to 1 or 0 respectively. We add these assignments to the Relay output as invisible conditions in its conditional macro. This enables us to remove two branches of the resulting system. It is not possible that the Relay is turned on (`RelayOutput = 1`) and the Switch is set to off (`RelayOutput<=0`). Analogous, when the Relay is turned off (`RelayOutput = 0`), the Switch cannot be set to on (`RelayOutput>0`).

Reduction of Algebraic Blocks

The individual transformation of blocks has some drawbacks. First, the transformation rules and the resulting macros do not contain information about the connected blocks and the connected macros. During the evaluation of the macros and the generation of the resulting $\mathtt{d}\mathcal{L}$ model, each macro needs to be applied to all other macros and the resulting model. This could be reduced, since most macros only need to be applied to macros that are generated by neighboring blocks. Second, the macro replacement mechanism is not able to handle algebraic loops. An algebraic loop is a group of Simulink blocks that construct a data dependency loop, where no block contains an inner state. That means that the output value of each block in the loop at each time is directly dependent on its current value. Algebraic loops can only be solved, if there is a stable assignment to the block outputs where they do not change. Our macro replacement mechanism for the model transformation, see Section 5.3, cannot handle algebraic loops, since these produce macros where the *id* that should be replaced is also part of the replacement term.

In the following, we introduce an extension of our block transformation to group the transformation of blocks. The aim is to reduce the macros that are produced during the transformation and to handle algebraic loops.

The key idea is to group blocks of similar type (e.g., arithmetic, logic) with a direct signal line connection into a block group. Then we can transform these groups independently from the rest of the system. The transformation of a block group generates macros similar to the transformation of individual blocks. For the transformation of a block group, we consider the blocks independent of the rest of the model. We determine macros that represent the behavior of the block group and that are used during the transformation of the whole model.

Transformation of Block Groups. In the following, we consider Simulink blocks that are connected and have the same functionality, e.g., arithmetic calculation or logic formula, as block group B. We define open inputs as all input ports of the blocks in B that are not connected to an output port of a block in B. Open outputs are all output ports that are connected to at least one block that is not in B. Note that it is possible that an open output is connected to an input port of a block in B.

Block groups can be transformed similar to Simulink models. We create macros for each block in B and get a resulting set of macros. We perform the macro replacement in the group until the resulting macros only contain identifiers that are produced by the open inputs. Note that this replacement terminates only if there is no algebraic loop present. The resulting macros represent the behavior of

the block in block in B. For the system transformation, we only need to consider macros that represent replacements for open outputs. During the transformation of the system, we do not consider the individual blocks in the group but use the resulting macros for the open outputs as transformation rule for the block group. A block group is interpreted as a single block during the system transformation. The rule by which this block is transformed is given by the macros that are generated by the evaluation of the block group. Furthermore, only the macros for the open outputs are relevant for the rest of the system.

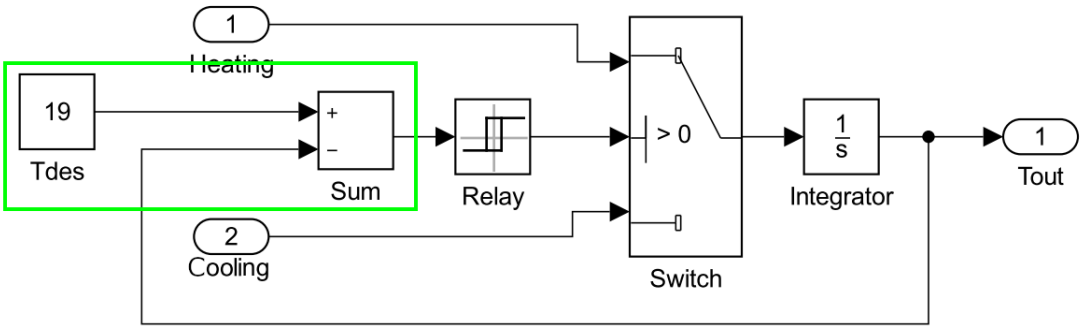


Figure 5.2: Block group in Temperature Control

Figure 5.2 shows the temperature control system with an arithmetic block group. This group consists of a *Constant* block and a *Sum* block. During the transformation as a first step, we only consider these two blocks. This group has one open input $out_{Integrator}$ and one open output out_{Sum}. We can apply the two macros of the two blocks within this group to obtain:

$$M : \alpha[out_{Sum} \leftarrow 19 - out_{Integrator}]$$

Only ports that are not part of the block group remain (i.e., $out_{Integrator}$). We can take the macro that describes the open output (out_{Sum}) as macro for this group. Therefore, the macro for the output port of the *Constant* block is not necessary during the transformation of the remaining system.

Transformation of Algebraic Loops. An algebraic loop is a loop that only consists of arithmetic blocks. Algebraic loops can be present in a block group. Note that we have defined a block group as connected blocks that have the same functionality. This means that there can be block groups that consist of blocks that perform an algebraic calculation. Furthermore, only feed-through blocks and source blocks can belong to this group, since blocks with internal states have a different functionality. To determine whether the system model contains an algebraic loop, we check whether there exists a block group of algebraic blocks that contains a loop. Since these blocks have no inner state, all loops in these groups are algebraic loops. To be able to create a transformation rule for an algebraic loop, we first encapsulate all blocks that are inside of an algebraic loop in its own block group. This results in a block group that contains another block group.

An algebraic loop can only be handled by the Simulink solver, if it has a solution. Therefore, we also only transform algebraic loops with a solution. To be able to solve the equation system that is represented by the Simulink blocks, we create the

equation system that is represented by the loop. Each block can represent one variable. To obtain the solutions that are relevant for the system, we use the blocks with open outputs as variables for the equation system. We use our macro replacement mechanism until the equation system only contains open outputs and open inputs as variables. With a computational algebra system, e.g., Wolfram|Alpha [Wol], we can automatically solve these equations. We obtain formulas that represent the values for open outputs. A formula for an open output only contains variables that refer to open inputs of the block group. We transform all formulas for open outputs into macros, where each variable in the equation system is either an *id* that is replaced by a macro that is generated by the block group or that describes the identifier of an open input. If the equation system does not have a solution, we do not obtain a formula and we cannot transform the system that contains this loop.

The resulting macros can be used in the system transformation similar to the macros of block groups.

5.5 Summary

In this chapter, we have introduced our transformation of Simulink models that capture the informal Simulink semantics into a formal $\mathsf{d}\mathcal{L}$ representation. We transform blocks with individual transformation rules. The overall $\mathsf{d}\mathcal{L}$ model is formed by hybrid programs that are provided by our transformation rules, evolutions of continuous variables and a sophisticated macro mechanism. The resulting $\mathsf{d}\mathcal{L}$ model represents a nondeterministic repetition of a Simulink solver step, where each step consists of the evaluation of discrete changes and the progress of time in one of different continuous evolutions. We presented optimizations to reduce the size of the resulting model and the macros that are used during the transformation. We transform blocks of the same type as groups, which encapsulate the behavior of similar blocks and can contain algebraic loops. We presented rules to connect the individual transformation rules and obtain a $\mathsf{d}\mathcal{L}$ model that we can use for the verification with the interactive theorem prover KeYmaera X.

6 Compositional Verification with Hybrid Contracts

With the formalization of Simulink models in dℒ, which we have introduced in the previous chapter, we enable the deductive verification of properties for Simulink models with the interactive theorem prover KeYmaera X, which we introduce in this chapter. We aim at integrating the verification results into the design process of hybrid control systems and reusing verification results for further formal proofs. We introduce a *service* concept for Simulink models that encapsulates a model and its formal interface description. Note that we use a simplified service definition in this chapter and extend our *service* concept by feature modeling in Chapter 7. First, we introduce our notion of *hybrid contracts* that provide guarantees for the behavior of a *service*. Second, we present extension functions that ease the creation of hybrid contracts. These functions insert observer variables in a dℒ representation of a Simulink model to enable the creation of more complex system properties. Furthermore, we present property templates that capture different important properties for Simulink models as hybrid contracts. Third, we introduce the verification of transformed models with the interactive theorem prover KeYmaera X. Fourth, we present our compositional verification to enable the verification of Simulink models that consist of interacting *services*. Lastly, we present a proof to show that properties that we verify for our abstracted compositional system also hold for the concrete system.

6.1 Hybrid Contracts

Complex models in Simulink are characterized by interacting components.

We introduce *services* in Simulink and *hybrid contracts* to capture the behavior of a Simulink component as hybrid program and to provide an abstract verifiable interface definition for services. Hybrid contracts formally describe the dynamic interface of a Simulink service and the interactions between different Services, which enables reasoning about the overall system. The dynamic interface captures the input and output behavior of a Simulink model, including time-discrete and time-continuous behavior. This section is based on [LHG19] where we have published our compositional verification of Simulink models.

Service-Oriented Design and Decomposition

In the following, we introduce a simplified definition of services in Simulink. We further extend this notion of a Simulink service in Chapter 7 by the addition of a feature model. The inclusion of the feature model is not necessary for the introduction of hybrid contracts and will be omitted in the following discussion. We define a service in Simulink as

$$s = \{sm, P_i, P_o, C\}$$

It consists of a Simulink model sm, a set of input Ports P_i, a set of output Ports P_o and a set of *hybrid contracts* C. The Simulink model defines the inner structure of a service, the input and output ports define its interface and the hybrid contracts

define its interface behavior. Whenever the Simulink model of a service is used in another Simulink model, we denote it as *Instance* of the service.

Hybrid Contracts

Hybrid contracts capture the hybrid behavior of a service, which means that they are restricted neither to purely discrete nor to purely continuous system behavior. To enable the integration of contracts in the system verification, we represent hybrid contracts in $\mathsf{d}\mathcal{L}$. A *hybrid contract* is a tuple $c = (\Phi_{in}, \Phi_{out})$, where Φ_{in} is a set of assumptions, and Φ_{out} is a set of guarantees. To verify that a service fulfills a hybrid contract, we verify that Φ_{out} holds after the behavior of the service, which is described in α, is executed an arbitrary number of times under the assumption that Φ_{in} holds:

$$\Phi_{in} \rightarrow [\{\alpha\}^*](\Phi_{out})$$

We show that the guarantees provided by Φ_{out} hold in all runs of the nondeterministic repetition of α, which represents the execution of the simulation loop. Therefore, our contracts are not restricted to the execution of one simulation loop and we can define hybrid contracts that contain information for multiple executions of the simulation loop. The assumptions describe behavior of input variables of a service. The guarantees provide information about the trajectories and values of output variables. Note that continuous evolutions run for a non-deterministic amount of time as long as their evolution domain holds. Our generated models are structured such that exactly one evolution is executed in each loop iteration. Each evolution can run an arbitrary amount of time as long as its evolution domain holds. Due to the [ ] operator in our proof of the system, each iteration of the simulation loop considers all possible runs for which the evolution domain holds. Therefore, all end states that are reachable by each evaluation of the simulation loop together provide the set of possible trajectories.

Assumptions and guarantees are modeled in $\mathsf{d}\mathcal{L}$. The use of $\mathsf{d}\mathcal{L}$ to define contracts enables us to describe a variety of possible behavior and we are not limited to classical discrete system properties, for example, range and timing properties. Inheriting the full expressiveness of $\mathsf{d}\mathcal{L}$ enables us to also express dynamic properties, i.e., differential equations together with discrete control signals, as well as continuous evolutions and assignments. However, to systematically make use of the high expressiveness of $\mathsf{d}\mathcal{L}$ requires a high expertise by the designer. To ease the contract definition process, we provide 1) *extension functions*, which enable the designer to systematically introduce observer variables into the design under verification and 2) *property templates*, i.e. templates for range properties, timing properties, and dynamic properties, which can be used as design patterns for typical requirements or contract definitions.

Contract for Running Example. In the following, we discuss how to create a simple contract for our temperature control service. The Simulink model of the service is shown in Figure 2.6. The following hybrid contracts serves as brief introduction to the creation of hybrid contracts. In the following sections, we extend the creation of hybrid contracts in more detail. A contract is created by the designer to formally describe the dynamic interface of a Simulink system.

Table 6.1: Example Contract for temperature bounds

ϕ_{in}	$\delta > 0 \land$ $Heating > 0 \land$ $Cooling < 0$
ϕ_{out}	$T_{des} - \delta < T_{out} < T_{des} + \delta$

The output temperature T_{out} is a crucial signal in this system. As example contract, which is depicted in Table 6.1, we provide the guarantee that the resulting temperature T_{out} is always close to the desired value T_{des}. It can deviate from T_{des} by a tolerance value of δ. The resulting guarantee ϕ_{out} is described as $T_{des} - \delta < T_{out} < T_{des} + \delta$. The temperature is time-continuous and this guarantee captures time-continuous behavior. To create the necessary assumptions, we consider the temperature control service presented in Figure 2.6. Note that the exact value of the tolerance δ is influenced by the *Relay* block, which can be determined during the interactive verification. As first assumption ϕ_{in1}, we assume that the tolerance δ is greater than zero. Furthermore, note that the temperature change is directly dependent on $Heating$ and $Cooling$. A temperature increase is necessary when the temperature falls below T_{des} and a decrease is necessary when it rises above T_{des}. Since the switch ensures that $Heating$ is used when the temperature is lower than $T_{des} - \delta$ and $Cooling$ is used when it is higher than $T_{des} + \delta$, we can create two separate assumptions ϕ_{in2} and ϕ_{in3} for these two input signals to ensure this behavior. Additionally, for all contracts it is necessary that the guarantee is initially fulfilled.

6.2 Requirement Definitions

Our formal interface description of Simulink models consists of formal behavior descriptions for the input signals and output signals. To provide a formal foundation for the verification of input and output properties, we present formal definitions for the creation of properties for Simulink systems. First, we introduce extension functions for generated $\mathsf{d}\mathcal{L}$ models that enable the inclusion of observer variables. These observer variables are used in the creation of properties for input and output signals to describe more sophisticated behavior. Afterwards, we introduce property templates that facilitate the creation of hybrid contracts by defining basic properties for Simulink models. These properties can be used in the verification of the behavior of a Simulink service.

Extension Functions and Observer Variables

To enable designers to systematically define requirements and contracts, we introduce functions that extend hybrid programs by observer variables. The introduced variables must not change the behavior of the underlying program. Note that it is possible that our extension functions change the values of newly added observer variables, only variables that are present in the original model must not be changed. Observer variables can be used to store the values of signals for use in contracts,

e.g., to compare the value of a signal at the start of a run of the simulation loop with its value at the end of each cycle.

Discrete observer variables. We introduce a function $u_D(\alpha, v := \theta, F)$ that prepends a conditional discrete assignment to a hybrid program. Under the condition F, the term θ is assigned to variable v. It is required that v is not bound in α. The expression θ may read variables from α.

$$u_D(\alpha, v := \theta, F) \equiv \{\text{if } (F) v := \theta; \cup ?(\neg F); \}\alpha$$

Example. In our temperature control system, we use u_D to keep track of how a variable changes during the execution of a simulation loop. To enable statements about the change of the temperature in our temperature control service, we introduce a variable $Integrator_{last}$ that stores the output value of the *Integrator* block (which represents the temperature) at the beginning of the simulation loop.

$$\begin{aligned}\alpha_{new} &= u_D(\alpha, Integrator_{last} := Integrator, true)\\ &= \{\text{if } (true) Integrator_{last} := Integrator; \}\alpha\\ &= \{?(true); Integrator_{last} := Integrator; \cup ?(\neg true); \}\alpha\end{aligned}$$

We set the condition to *true* to perform this assignment at every execution of the simulation loop.

Continuous observer variables. We introduce a function $u_C(\alpha, v' = \theta)$ that adds a continuously evolving variable $v' = \theta$ to the continuous evolutions of a given hybrid program. Note that this variable may not restrict the underlying differential equations. We define u_C for hybrid programs as follows:

$$\begin{aligned}u_C(\alpha; \beta, v' = \theta) &\equiv u_C(\alpha, v' = \theta); u_C(\beta, v' = \theta)\\ u_C(\alpha \cup \beta, v' = \theta) &\equiv u_C(\alpha, v' = \theta) \cup u_C(\beta, v' = \theta)\\ u_C(\alpha^*, v' = \theta) &\equiv u_C(\alpha, v' = \theta)^*\\ u_C(x := \mu, v' = \theta) &\equiv x := \mu\\ u_C(x := *, v' = \theta) &\equiv x := *\\ u_C(?(F), v' = \theta) &\equiv ?(F)\\ u_C((x_1' = \mu_1, ..., x_n' = \mu_n \& F), v' = \theta) &\equiv (x_1' = \mu_1, ..., x_n' = \mu_n, v' = \theta \& F)\end{aligned}$$

Other parts of a hybrid program are not changed by u_C. To ensure that u_C does not change the behavior of the underlying system α, we require that v is a fresh variable, i.e., it is not used anywhere in α. Note that the construction of our transformed model ensures that in each run of the simulation loop exactly one continuous evolution is executed. Therefore, each variable v only evolves once during the execution of the $\mathsf{d}\mathcal{L}$ model of the system.

Example. We use a continuous extension function in combination with a discrete extension function to add a timer that resets under the condition *flag*. The use of both extension functions is as follows:

$$\alpha_{new} = u_D(u_C(\alpha, timer' = 1), timer := 0, flag)$$

Property Templates

To ease the contract definition process, we provide three property templates, namely range properties, timing properties, and dynamic properties, which can be used as design patterns for the contract definition. Note that we leverage the basic structure of the simulation loop in our transformed system. The behavior of the transformed system is split in two parts, see Section 5.1. There are no continuous evolutions in β. The hybrid program γ contains the continuous parts of the system in a nondeterministic choice.

Range properties. Range properties define bounds on the values of given signals.

$$[\{\beta;\gamma\}^*]\ (x \sim r_1 \wedge y \sim r_2 \wedge ...)$$

where x, y are signals, $\sim \in \{=, \neq, <, >, \geq, \leq\}$, and r_1, r_2 are terms that evaluate to real numbers. It is possible to define more than one relation $\sim$ for one variable. Note that we typically assume that range properties should always hold for all possible runs of a given hybrid program, i.e., we use the [ ] operator.

Example. In the temperature control system, we can describe the requirement to keep the output temperature $Tout$ in desired bounds as a range property:

$$[\{\beta;\gamma\}^*](Tout \leq Tdes + \delta \wedge Tout \geq Tdes - \delta)$$

where $Tdes$ represents the desired temperature, $\delta > 0$ denotes the allowed deviation, $Tout$ is the controlled temperature.

Timing properties. To ensure that the system reacts to a special system state or signal before or after a given time, a timing property can be used. To refer to arbitrary starting points of the simulation time, we propose to use *clock* variables. A clock variable has a derivative of 1 and can be reset in the $\mathsf{d}\mathcal{L}$ representation of the system, for example, if a special system state is reached or if a given signal exceeds a given value. The general form of a timing property is:

$$[\{u_D(\beta, clock := 0, true); u_C(\gamma, clock' = 1)\}^*](stateCondition \rightarrow clock \sim r)$$

where *clock* is a clock variable, and *stateCondition* describes a special system state that triggers a timing condition $clock \sim r$. In the relation, r is a real number and $\sim$ is a relation with $\sim \in \{<, \leq, =, \geq, >, \neq\}$.

Example. In the temperature control system, we use a timing property to ensure that no rapid switching occurs. With additional clock variables *relayOnTime* and *relayOffTime* that keep track of the time elapsed between switching, we can formulate a property that states that no rapid switching occurs, i.e. at least a time

MIN elapses before the state of the switch changes.

$$
\begin{aligned}
&[\ \{u_D(u_D(\beta, relayOnTime := 0, 19.0 - Integrator \geq 0.5),\\
&\quad relayOffTime := 0, Tdes - Integrator \leq -0.5);\\
&\ u_C(u_C(\gamma,\ relayOnTime' = 1), relayOffTime' = 1)\\
&\ \}^*\\
&]((Relay = 0.0 \rightarrow relayOnTime \geq MIN)\\
&\ \wedge\ (Relay = 1.0 \rightarrow relayOffTime \geq MIN))\\
\equiv\ &[\ \{\{\text{if } (Tdes - Integrator \leq -0.5)\ relayOffTime := 0\};\\
&\quad \{\text{if } (19.0 - Integrator \geq 0.5)\ relayOnTime := 0\};\\
&\quad \beta;\\
&\quad u_C(u_C(\gamma,\ relayOnTime' = 1), relayOffTime' = 1)\\
&\ \}^*\\
&]((Relay = 0.0 \rightarrow relayOnTime \geq MIN)\\
&\ \wedge\ (Relay = 1.0 \rightarrow relayOffTime \geq MIN))
\end{aligned}
$$

This property describes that depending on the state of variable *Relay* the values of the clocks *relayOnTime* and *relayOffTime* have reached a value of at least *MIN*.

Dynamic properties. Dynamic properties define arbitrary relations of values before and after a run of the simulation loop. We use observer variables to keep track of the change of values and the elapsed time.

Example. To ensure that a dynamic change of the temperature is always kept in bounds that are given by δ, we define the following dynamic property:

$$
\begin{aligned}
&[\{u_D(u_D(\beta, clock := 0, true), Integrator_{last} := Integrator, true);\\
&\ u_C(\gamma, clock' = 1)\\
&\ \}^*]\\
&(Integrator_{last} - Integrator \leq \delta \cdot clock\ \&\\
&\ Integrator_{last} - Integrator \geq -\delta \cdot clock)
\end{aligned}
$$

where *clock* is a clock variable and $Integrator_{last}$ an observer variable, which stores the last value of the temperature ($Integrator$) at the start of each simulation loop. The property ensures that $Integrator_{last}$ can not deviate from $Integrator$ by more than $\delta \cdot clock$ in all runs of the hybrid program, for $\delta > 0$.

6.3 Service-Oriented Verification

To ensure that a given contract holds for a service, we transform the service into $\mathsf{d}\mathcal{L}$. If the service only contains atomic Simulink blocks, we can use our Simulink to $\mathsf{d}\mathcal{L}$ transformation (Chapter 5). If a service contains other subsystems, we can either flatten the system by replacing all subsystems by their inner blocks or, if an inner subsystem is a service with already verified contracts, we can use our abstracted transformation for this inner service.

The transformation provides us with a hybrid program α. To ensure that a contract (Φ_{in}, Φ_{out}) holds for this system, we create the following $\mathsf{d}\mathcal{L}$ formula.

$$(\Phi_{in}) \rightarrow [\alpha](\Phi_{out})$$

where Φ_{in} contains the preconditions, α is the transformed system as hybrid program and Φ_{out} contains the postconditions. We need to ensure that under the assumption that Φ_{in} holds, all possible runs of the hybrid program α fulfill the guarantee Φ_{out}. We can verify such contracts using KeYmaera X.

Compositional Verification of Hybrid Simulink Models

To increase the scalability, we present a service-oriented verification approach that uses hybrid contracts to abstract the behavior of services in system verification. The key idea is to decompose the system into services, define a hybrid contract for each service, and separately verify that 1) each service adheres to its hybrid contract, and that 2) the overall system with services replaced by their hybrid contracts satisfies the requirements. The results of the service verification are reusable in the system verification and the service proofs scale better than monolithic proofs for large systems. We use the following rules to transform a service by its contract instead of its inner structure:

1. We introduce fresh variables for all incoming and outgoing ports.
2. We connect its input ports, i.e., we assign the input variables of the service with the respective expressions from the surrounding Simulink model.
3. We nondeterministically choose the output signals of the service such that it satisfies the contracts, i.e., we use a nondeterministic assignment and restrict the resulting value with $\Phi_{out,n}$ whenever $\Phi_{in,n}$ holds for all hybrid contracts of the service.
4. If a contract refers to inner variables, these are added to the $\mathsf{d}\mathcal{L}$ model.

Note that it is not trivial to show that a given system provides the same guarantees as all of its internal services. The services can only provide their guarantees under the given assumptions for their input signals. In the system, these input signals are provided by the outputs of other services or blocks of the system. Therefore, to use the guarantees of a service in the compositional verification, we show that the assumptions are provided by the system or other services.

In the system verification, we obtain guarantees Φ_{out} for the output signals and inner signal of the system. We can also add assumptions Φ_{in} for the incoming signals. Therefore, a proof for a system provides a hybrid contract about this system. Note that it is also possible that safety properties of a system should always hold. In this case, the assumptions are set to *true*.

Example Contracts for the Temperature Control Service

We have verified that our temperature control system keeps the temperature, which is given by the output value of *Integrator*, in a certain range around the desired temperature, and that we avoid rapid switching.

Correct Temperature Range. To show that the temperature control system keeps the temperature in a certain range of $\delta \in \mathbb{R}$ around the desired value $Tdes \in \mathbb{R}$, we have defined the following property:

$$\Phi_{in} \rightarrow [\{\alpha\}^*]\, Tdes - \delta \leq Integrator \leq Tdes + \delta$$

We use the modal box operator [ ] to prove that this property holds after each simulation step. In the following, we set $Tdes = 19$ and $\delta = 1$. To verify that our desired property holds as a loop invariant, we use loop induction. This yields the following proof goals:

$$\begin{aligned} &Heating \cdot t + Integrator \geq 18 \land Heating \cdot t + Integrator \leq 20 \\ \land &Cooling \cdot t + Integrator \geq 18 \land Cooling \cdot t + Integrator \leq 20 \end{aligned}$$

where $t \in \mathbb{R}$ is a small step. With the interactive proof in KeYmaera X, we obtain the following open subgoals:

$$\begin{aligned} t \geq 0 \land \forall \tau (0 \leq \tau \leq t \rightarrow &(19.0 - (Heating \cdot \tau + Integrator) >= 0.5 \\ &\lor \tau + smallStep <= EPS)) \\ t \geq 0 \land \forall \tau (0 \leq \tau \leq t \rightarrow &(19.0 - (Cooling \cdot \tau + Integrator) <= -0.5 \\ &\lor \tau + smallStep <= EPS)) \end{aligned}$$

These proof goals show us that the input values *Heating* and *Cooling* need to provide a non-negative heating value and a negative cooling value, respectively. To resolve these goals and finally, to prove the desired property, we have manually added the following preconditions, which we use as assumption for the hybrid contract:

$$0 \leq Heating \leq 20 \land -20 \leq Cooling \leq 0$$

This means that we can verify that the system keeps the temperature in the desired range for all possible input scenarios where the values of *Heating* and *Cooling* are restricted by 20 and -20, respectively. The desired property can be shown automatically using the *Auto* tactic in KeYmaera X. Overall, the only manual interactions necessary are the introduction of the desired property as loop invariant and two additional preconditions.

Absence of Rapid Switching

To show that our temperature control system avoids rapid switching we have defined a constant *MIN*, which defines the required minimal time interval between two switching actions. For this example, we set *MIN* to 0.01. In addition, we have introduced two additional time variables *relayOnTime* and *relayOffTime*, which are reset whenever the *Relay* is set to 1.0 or 0.0, respectively. Then, we have defined the absence of rapid switching with the following property:

$$Relay = 0.0 \rightarrow relayOnTime \geq MIN \land\ Relay = 1.0 \rightarrow relayOffTime \geq MIN$$

Again, the modal box operator defines that the property should be true after all runs of a given hybrid program. We have again introduced the property as a loop

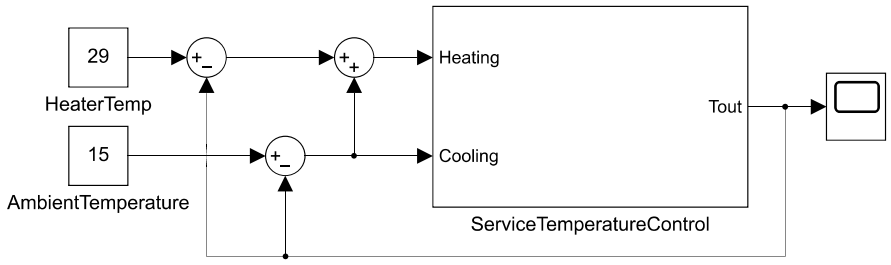


Figure 6.1: Temperature Control Service in Environment

invariant to ensure that it holds before and after each simulation step. In addition, we have manually inserted the following loop invariants:

$$Tdes - Integrator + Cooling \cdot relayOffTime \leq -0.5$$
$$Tdes - Integrator + Heating \cdot relayOnTime \geq 0.5$$
$$Relay = 1.0 \rightarrow Tdes - Integrator > -0.5$$
$$Relay = 0.0 \rightarrow Tdes - Integrator < 0.5$$

During the proof in KeYmaera X, we show that these invariants hold for the system and use them to prove the desired property. We have also included all loop invariants as preconditions, which also provide the guarantees of the hybrid contract. In addition, we have manually added the following preconditions:

$$MIN = 0.01$$
$$relayOnTime \geq MIN \wedge relayOffTime \leq MIN$$
$$(-1/MIN) \leq Cooling < 0.0 \wedge 0.0 < Heating \leq (1/MIN)$$

This means that we can verify the absence of rapid switching for all possible input scenarios where the values of *Heating* and *Cooling* are restricted to a certain range. For example, if the minimal distance between two switching actions is 0.01, *Heating* should be lower than 100 and *Cooling* greater than -100. Again, the desired property can be shown in KeYmaera X using the *Auto* tactic. Overall, the only manual interactions necessary are the introduction of the postcondition as loop invariant, four additional loop invariants, and four additional preconditions, which are the assumptions for the hybrid contract.

```
1  {
2  ...
3  S_Heating := 29 - S_Tout + 15 - S_Tout;
4  S_Cooling := 15 - S_Tout;
5  S_Tout := *;
6  relayOffTime := *;
7  relayOnTime := *;
8  Relay := *;
9  % Hybrid contract for correct temperature range
10 ?(((S_Heating ≥ 0) & (S_Heating ≤ 20) & (S_Cooling ≤ 0) & (S_Cooling ≥ -20))
11 ->
12 ((19.0 - 1.0 ≤ S_Tout) & (S_Tout ≤ 19.0 + 1.0)));
13 % Hybrid contract for absence of rapid switching
14 ?(((S_Heating ≥ 0) & (S_Heating ≤ 10) & (S_Cooling ≤ 0) & (S_Cooling ≥ -10))
15 ->
16 (((Relay=0)->(relayOnTime ≥ MIN)) & ((Relay=1)->(relayOffTime ≥ MIN))));
17 ...
18 }*
```

Listing 6.1: Compositional Verification Example: Abstract System

An illustrative embedding of our temperature control service into a simple environment is shown in Figure 6.1 and the transformed model in $\mathsf{d}\mathcal{L}$ is shown in Listing 6.1. Lines 3 – 4 show assignments to the input ports of the service given by the environment. They depend on the service output as it is embedded into a feedback loop in the overall system. Lines 5 – 8 show the nondeterministic assignments to the service output signal and its internal signals. In Lines 9 – 12, the output signal is assigned according to the first contract. In Lines 13 – 16, the internal signals are assigned according to the second contract. Note that in this example no assertions about the values of the internal *Relay* signal are present. We can simply add a third hybrid contract to the temperature control service that guarantees that the *Relay* output signal is always either 0 or 1. This can be always ensured and the assumption can be set to *true*.

6.4 Automated Invariant Generation

The formal verification of hybrid control systems is a time intensive task. The interaction between discrete and continuous components introduces an enormous complexity. With our transformation of systems into $\mathsf{d}\mathcal{L}$, we gain access to the powerful, interactive theorem prover for hybrid systems KeYmaera X [Ful+15]. KeYmaera X, however, does not provide these proofs fully automatically, but requires manual interactions from the user to provide proof ideas, invariants, and to guide the verification process.

The overall system behavior of transformed Simulink models arises from the composition of the inner services. A major challenge in the interactive verification process is to specify contracts such that all behaviors are captured that are necessary to prove the system requirements and at the same time to make sure that the provided abstractions are strong enough such that the verification can be completed in acceptable time. The verification of the system consists of two parts. First, the verification of the individual services, where we show that each service adheres to its contract. Second, the verification of the overall system, where we use

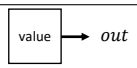
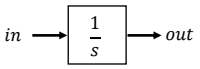
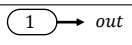
the service contracts to prove system properties. At first, this sounds straightforward. However, both parts of the verification process are intertwined. The service contracts are needed for the system verification, but at the same time only during the overall system verification, we notice whether the chosen service contracts are expressive enough to prove the system properties. If it is not possible to find a proof for a system property, this can have different causes. For example, if the model does not fulfill the system property, this means that either the model contains an error or the system property needs to be changed. If the contracts of the services are not expressive enough to infer the system property, more details need to be added. If a contract refers to values that are not considered by other services, additional contracts are needed.

Interactive verification. The overall structure of dℒ models that result from an automatic transformation of a Simulink model with our approach is a nondeterministic repetition as shown in Listing 5.2. This makes KeYmaera's *loop induction proof rule* the most important proof rule for transformed systems. The definition of loop invariants is generally the verification step where human insight is most needed [MP17]. The loop induction proof rule specifies that a nondeterministic repetition fulfills its guarantees if three conditions are fulfilled: First, the *loop invariant* must be fulfilled before the simulation loop is executed. Second, if the *loop invariant* holds before one execution of the loop, it must hold after one execution of the loop. Third, the guarantees can be provided if we can show that the *loop invariant* implies the desired guarantees. As dℒ models that result from our Simulink to dℒ transformation always have a nondeterministic repetition representing the global simulation loop as outermost statement, the guarantees that can be proven directly correspond to the invariants of this loop and vice versa. Overall, the interactive verification process has the following structure:

1. Define guarantees for any individual service. At first the assumptions can be set to *true*.
2. Perform the proof in KeYmaera X.
3. If the proof cannot be completed, use the open proof goals to infer necessary restrictions on the input signals. Add these as assumptions and go to 2.
4. Redo steps 1 to 3 for all services. It often makes sense to propagate assumptions of some services as guarantees for other services or vice versa.
5. Perform the compositional system verification.
6. If the proof cannot be completed, infer additional guarantees from the open proof goals that need to be provided by the service contracts and go to step 2 for all relevant services.

In our experiences, we found that it is often helpful to start with signal bounds on input or output signals: a signal is zero, a signal is not equal to zero, a signal has negative values, a signal has positive values, the value of a signal increases over time, the value of a signal decreases over time. Signal bounds can be used both as assumptions on input values and as guarantees on output values. Further

Table 6.2: Example Signal Bound Invariants

Block	Parameter	Signal bound
	value	$out \leq value \wedge\ out \geq value$
	upperBound(optional), lowerBound(optional)	$out \leq upperBound$ $\wedge out \geq lowerBound$
	User defined	$out \geq low_1 \wedge out \leq up_1$

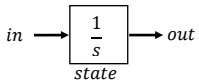
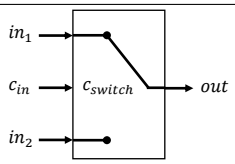
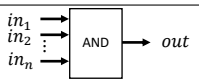
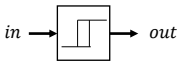
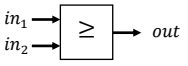
assumptions and guarantees can often be inferred from the system requirements. This requires some domain expertise since the effect of a service on the system behavior is often non-trivial. To ease the manual definition of hybrid contracts, we have presented property templates for range properties, timing properties, and dynamic properties in Section 6.2.

This section is based on [LHG20] where we have evaluated the manual effort during verification of services and have presented techniques to reduce the manual input during verification. In the following, we discuss the manual effort that is necessary to verify hybrid control systems that are modeled in Simulink with our transformation from Simulink to dℒ and the interactive theorem prover KeYmaera X. We present techniques to increase the automation in the creation of invariants for the verification. The key idea is to extract implicitly available information of the Simulink model, for example, data types and the semantics of blocks, to automatically generate invariants for the automatically generated dℒ model. The additional invariants simplify the interactive proof process and significantly reduce the necessary user interactions. Note that the automatically extracted information can, in some cases, also provide formal interface conditions that can be used as hybrid contracts and thus reduce the manual effort to define formal properties of a system or service. This further reduces the manual effort of compositional, service-oriented verification. We have developed the concepts of the automated invariant generation as part of a Bachelor's thesis [Sch19].

Invariant Types

In our dℒ models, invariants capture relations between state variables that hold before and after each execution of the global simulation loop. Such relations are generally hard to find. However, some invariants can directly be derived from data or signal types, or from the block semantics in a given Simulink model. For example, many blocks restrict the range of their output signals. Furthermore, the evolution of an *Integrator* block depends on its input value. If we assume that the input signal is in a given range then we can infer properties of the output of *Integrator* blocks, e.g., if the input signal is greater than zero then the value of the *Integrator* will increase. Such invariants can be viewed as contracts on a very fine granular level of individual Simulink blocks, and they are implicitly introduced into the model whenever the corresponding block type is used. In this section, we propose to exploit this implicitly available information and enrich it with knowledge about the block semantics that needs to be defined by the designer only once and then can be reused for every model that uses these blocks.

Table 6.3: Example Block Invariants

Block	Parameter	Invariant
	n/a	$(in > 0 \rightarrow state_{start} \leq state) \wedge$ $(in < 0 \rightarrow state_{start} \geq state) \wedge$ $(in = 0 \rightarrow state_{start} = state)$
	c_{switch}	$(c_{in} > c_{switch} \rightarrow out = in_1) \wedge$ $(\neg (c_{in} > c_{switch}) \rightarrow out = in_2)$
	c_{logic}	$(c_{logic}(in_1, in_2, ...in_n) \rightarrow out = 1) \wedge$ $(\neg c_{logic}(in_1, in_2, ...in_n) \rightarrow out = 0)$
	$up, low,$ out_{up}, out_{low}	$(in_1 > up \rightarrow out = out_{up}) \wedge$ $(in_1 < low \rightarrow out = out_{low})$
	$c_{relation}$	$c_{relation}(in_1, in_2) \rightarrow out = 1 \wedge$ $\neg c_{relation}(in_1, in_2) \rightarrow out = 0$

Signal bounds. Many blocks in Simulink restrict the range of their output signals. For example, for a unit delay block, optional block parameters *upperBound* and *lowerBound* may be defined by the designer. If they are set, the output signal is always kept in the given range. To automatically generate invariants that exploit and capture this information, we perform an analysis where we collect the range of each signal at each block input or output. An example for the invariants that we automatically generate as a result is shown in Table 6.2. Note that the bounds for the *Import* block are not part of the Simulink model but the user can define bounds for these signals that are used in the following invariant generations.

Block semantics. In some cases, the behavior of a block can be described as a fine granular contract individually. For example, an integrator block will never decrease its value if the input is positive (and vice versa), or a switch block will always put one of its inputs through to the output, depending on the control condition. Although this information is already (implicitly) encoded into the $\mathsf{d}\mathcal{L}$ model that is automatically generated from a given Simulink model with our transformation from Simulink to $\mathsf{d}\mathcal{L}$, it significantly eases the verification process if the underlying relation between state variables is explicitly captured as an invariant. Examples for such invariants, which can directly be derived from the block semantics, are shown in Table 6.3. Note that the derivation of such invariants is not trivial and depends on the specific semantics of the block. However, once such fine granular contracts of individual blocks are defined, they can be reused whenever the corresponding block type is used.

Delay propagation. Many discrete-time blocks in Simulink delay a given input signal according to their sample time. If the same input signal is fed into multiple discrete blocks that also have the same sample time, the internal state of these blocks will change to the same value exactly at the same simulation steps. To capture this, we perform an analysis where we track the number of delay samples

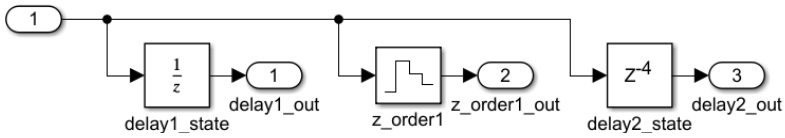


Figure 6.2: Example Delay Propagation

for each signal, and automatically generate invariants whenever state variables hold the same signals that are delayed for the same number of simulation steps. Figure 6.2 shows different variations of discrete delays in Simulink. The block *delay1_state* takes its input value at the start of a time-step and sets its output to this values at the start of the following time-step. The block *z_order1* sets its output value to the value that it has received at the start of the current time-step. The block *delay2_state* takes its input value at the start of a time-step and sets its output to this values with a delay of four time-steps. For this given model, we automatically infer the invariant $delay1\_state = z\_order1\_out = delay2\_state0$. In other words, the input signal delayed by one simulation step is exactly the same if it is propagated to the internal state variable of a unit block on one path (*delay1_state*), delayed by a zero order hold and written to its output on a second path (*z_order1_out*), or propagated to the first internal state variable in a delay block with four delay slots on a third path (*delay2_state0*). Note that *delay2_state* delays the incoming signal by four time steps before it is written to its output. Therefore, it produces the state variables *delay2_state0* to *delay2_state3*. For each block that introduces delays into the system, we can automatically generate the corresponding invariants to ease the verification process.

Error checking. There exist some general conditions that are often desirable to guarantee fault-free behavior, for example the absence of errors from well-known error-classes like *overflows* and *division-by-zero*. An overflow can be produced by arithmetic operations if the result does not fit into the underlying hardware data type. To account for this, we provide user-defined constants that may define lower and upper bounds on all signals. We automatically generate invariants that check for overflows whenever arithmetic operations are performed, e.g., at sum, product, and integrator blocks. Similarly, we automatically generate invariants that prohibit a division by zero for all blocks that perform a division.

6.5 Soundness of Compositional Verification

In this section, we discuss the soundness of our abstraction, which replaces services by their contracts to abstract from their inner block structure. Note that contracts provide an over-approximation of the service behavior, which is guaranteed by the verification of the services in KeYmaera X. Our approach is sound in the sense that properties that hold for the abstract system are guaranteed to be preserved in the concrete system. Our key idea to ensure soundness is twofold: First, we assume that variables written by a service are not written somewhere else in the system. This is ensured by the syntactical structure of a Simulink model, as all inner blocks of a service can only be accessed through input signals. Second, we have to ensure that our proposed abstraction of a service by its contract during system transformation

does not restrict the behavior that would be created by the system transformation when using the inner block structure of the service. To ensure this, we support only the modal operator [ ], i.e. we support only safety properties and no liveness properties. This enables us to use the contract as replacement for the service and still be sure that properties of the abstracted model also hold in the original model.

The basic idea of our soundness proof is that we show that the abstract $\mathsf{d}\mathcal{L}$ system model that contains the abstracted service is an over-approximation of the $\mathsf{d}\mathcal{L}$ system model that is obtained when the concrete inner block structure of the service is used in the transformation of the system. Since we obtain both models via our transformation (Section 5.1), the system model and the service model have the following layout:

$$\alpha = \{\beta; \gamma\}^*$$

where β contains the discrete behavior and γ contains the continuous behavior of the transformed model. Therefore, for the soundness proof, we first show that the discrete behavior of a service is always included in the abstraction provided by the hybrid contract. That is, when starting at the same states, all runs of the concrete hybrid program produce a set of states that is a subset of the states that are reached by the runs of the abstract hybrid program. Second, we show that the continuous behavior of the abstract system is not restricted by the abstraction.

To show soundness of our contract-based verification approach, we use the dynamic semantics of $\mathsf{d}\mathcal{L}$ as summarized in Section 2.5, namely the semantic interpretations of a hybrid program that defines transitions to reachable states. In addition, we use the bound variables of a hybrid program

$$BV_{HP}(\alpha) = \bigcup \{x \in V : \text{ there are } I \text{ and } (\nu, \omega) \in I[\![\alpha]\!] \text{ such that } \nu(x) \neq \omega(x)\}$$

that contain all variables that are changed by the hybrid program α.

We introduce two extension functions for hybrid programs: $store(BV_{HP}(\alpha))$ represents a sequential execution of all discrete assignments, where the current values of the bound variables of α are assigned to observer variables. Note that all observer variables are fresh variables and $store(BV_{HP}(\alpha))$ does not change variables that are in the original α:

$$BV_{HP}(store(BV_{HP}(\alpha))) \cap V(\alpha) = \emptyset$$

The second extension function $BV_{HP}(\alpha) := *$ assigns arbitrary values to all bound variables of α.

Let *sys* be a system that contains a service *ser*. Furthermore, the service *ser* fulfills the contract (Φ_{in}, Φ_{out}). The behavior of *sys* is given by the hybrid program ${\alpha_{sys}}^*$, where α_{sys} represents the behavior of *sys* for one execution of the simulation loop. Note that the hybrid program is created via the transformation of a Simulink model, i.e. the original Simulink model of *sys* contains a service *ser*. Therefore, the behavior that is represented in α_{sys} contains the behavior of the service *ser*. The behavior of the service *ser* is represented by the hybrid program ${\alpha_{ser}}^*$, where α_{ser} represents the transformed behavior of the Simulink service *ser*. Note that α_{ser} only contains the behavior of *ser*. To verify that the system *sys* fulfills the property Q under the assumptions P, we show the following:

$$\vdash P \rightarrow [(\alpha_{sys})^*]Q$$

The assumptions P are properties that hold for the whole system, like initial values and properties of the environment.

In the following, we consider hybrid programs that are created by our transformation, see Section 5.1. These programs have the form $\alpha = \beta; \gamma$, where β contains the discrete behavior and γ contains the continuous evolutions. We can split the concrete system behavior α_{sys} to obtain $\beta_0; \beta_{ser}; \beta_1; \gamma_{sys}$, where β_0 and β_1 are discrete parts that do not contain functionality of the service, β_{ser} denotes the discrete part of the service, and γ_{sys} represents the continuous part of the system. This separation is possible due to the construction of the $\mathsf{d}\mathcal{L}$ model according to the block transformation rules. For the abstracted system, we write:

$$\begin{aligned}\alpha_{sys/abstract} =& \beta_0; store(BV_{HP}(\beta_{ser})); BV_{HP}(\beta_{ser}) := *; ?(\Phi'_{in} \rightarrow \Phi_{out}); \beta_1; \\ & store(BV_{HP}(\gamma_{ser})); BV_{HP}(\gamma_{ser}) := *; \gamma'_{sys}; ?(\Phi'_{in} \rightarrow \Phi_{out})\end{aligned}$$

Where γ'_{sys} denotes the continuous evolutions of the system, that are not present in γ_{ser}. Φ'_{in} denotes the contract assumptions where all occurrences of bound variables of α_{ser} are replaced with their respective observer variables. We use this to show that the runs of the concrete model are a subset of the abstracted model. Since the system parts β_0 and β_1 provide identical transitions in the concrete and abstracted model, we only need to show that the two abstracted parts for the discrete and continuous parts provide at least the runs of their concrete representation. That is, all states that are reachable by β_{ser} are also reachable by the abstracted representation:

$$\begin{aligned}&(1)\ I[\![\beta_{ser}]\!] \subseteq I[\![store(BV_{HP}(\beta_{ser})); BV_{HP}(\beta_{ser}) := *; ?(\Phi'_{in} \rightarrow \Phi_{out})]\!] \\ &(2)\ I[\![\gamma_{sys}]\!] \subseteq I[\![store(BV_{HP}(\gamma_{ser})); BV_{HP}(\gamma_{ser}) := *; \gamma'_{sys}; ?(\Phi'_{in} \rightarrow \Phi_{out})]\!]\end{aligned}$$

The first statement describes that all runs of the discrete part of the service β_{ser} are contained in the runs of an abstracted execution. Note that β_{ser} is executed as part of the concrete system. In the abstracted execution, we first store the initial values of the bound variables of the service and assign arbitrary values to these variables. Afterwards, the abstracted execution only considers runs that do fulfill the hybrid contracts via a test formula. The second statement describes that the runs of the concrete continuous evolutions γ_{sys} are contained in the runs of an abstracted continuous evolution. In the abstracted continuous evolution, we store the initial values of the variables that are changed by the evolution of the service and assign arbitrary values to them. Then we evolve the remaining variables according to their evolutions γ'_{sys}. Afterwards, the abstracted version only considers runs that do fulfill the hybrid contracts.

Furthermore, we require that the variables in the continuous evolutions of γ_{ser} do not influence the evolutions of γ'_{sys}: $BV_{HP}(\gamma_{ser}) \cap FV_{HP}(\gamma'_{sys}) = \emptyset$. $FV_{HP}(\gamma'_{sys})$ denotes the variables that are read by the hybrid program γ'_{sys} and for which the evaluation depends on their initial value.

For (1), it suffices syntactically to consider β_{ser} because we have introduced fresh variables for all input and output ports. For (2), we need to consider γ_{sys}, since all continuous evolutions evolve concurrently.

After obtaining (2), (1) can be shown analogously, without continuous evolutions.

For the abstracted system, we obtain the following runs:

$$
\begin{aligned}
& I[\![store(BV_{HP}(\gamma_{ser})); BV_{HP}(\gamma_{ser}) := *; \gamma'_{sys}; ?(\Phi'_{in} \rightarrow \Phi_{out});]\!] \\
= & I[\![store(BV_{HP}(\gamma_{ser}))]\!] \circ I[\![BV_{HP}(\gamma_{ser}) := *]\!] \circ I[\![\gamma'_{sys}]\!] \circ I[\![?(\Phi'_{in} \rightarrow \Phi_{out})]\!] \\
= & \{(\nu, \omega) : \omega =_{exp} \nu, \text{ where } =_{exp} \text{ describes that the states are equal in all} \\
& \text{variables execept:} \\
& \text{i) all observer variables are set to the observed values} \\
& \text{ii) all variables } v \in BV_{HP}(\gamma_{ser}) \text{ have an arbitrary value that fulfills } (\Phi'_{in} \rightarrow \Phi_{out}) \\
& \text{iii) all variables in } \gamma'_{sys} \text{ evolve according to their evolution domains} \\
& \text{independent of the values of } v \in BV_{HP}(\gamma_{ser})\}
\end{aligned}
$$

For the concrete system the following states are reachable:

$$
\begin{aligned}
I[\![\gamma_{sys}]\!] = & \{(\nu, \omega) : \omega =_{exp2} \nu, \text{ where } =_{exp2} \text{ describes that the states are equal} \\
& \text{in all variables except for all bound variables in } \gamma_{sys}, \\
& \text{which evolve according to their evolution domains}\}
\end{aligned}
$$

As i) observer variables are guaranteed by definition to only read variables from the original system, and ii) our service verification has shown that the contract $(X'_A \rightarrow \Phi_{out})$ provides an over-approximation of the possible service behavior, i.e. allows more values for $v \in BV_{HP}(\gamma_{ser})$ than the concrete service, we can conclude that (2) is satisfied, and thus our embedding of contracts to abstract from services is sound.

6.6 Summary

In this chapter, we have presented our approach for the verification of Simulink models with the interactive theorem prover KeYmaera X. We used our formalization of Simulink models in $\mathsf{d}\mathcal{L}$ to obtain a formal representation and we have introduced hybrid contracts to provide a formal interface description of the behavior of Simulink systems. Furthermore, we have introduced extension functions for the $\mathsf{d}\mathcal{L}$ model and property templates for the hybrid contracts to facilitate the verification. Lastly, we have presented our compositional verification approach. This enables us to verify properties of systems that consist of interacting components.

7 Variability by Service-Oriented Design

In this chapter, we extend our service notion, to enable a wider reuse of services in Simulink models. In the previous chapter, we have introduced services in Simulink that consist of one Simulink model that defines their inner structure and a set of ports for their input and output interface. To facilitate the reuse of services, we present variability for our Simulink services. We create individual hybrid contracts that describe the behavior of different variants and also use extended hybrid contracts that can be applied to multiple variants of the same system.

To this end, we introduce a feature model that contains a core model (or *root feature*), which consists of Simulink blocks and signals. The core model, which is defined by the root feature, contains the Simulink blocks and signals that define the basic functionality of a modeled system. This core model can be adapted by further blocks and signals by its child features. It is also possible to change existing blocks and signals in the child features. We use feature dependencies to group different features together, which provide similar extensions to the core model.

With an adaption of feature modeling, we provide means to adjust the behavior of a service to adapt it to a required context of a system design in a well-structured and predefined way. In particular, feature modeling enables the addition and deletion of functionality and thus also structural changes. By defining the possible variations of a given component in a feature model, changes are applied such that no unexpected behavior may arise. This chapter is based on the concepts of [Lie+17], in which we introduced the idea of combining feature modeling with Simulink, and the concepts of [LHG21], where we presented the integration of feature modeling into Simulink.

In the following, we introduce a representation for our customizable services. Then, we explain our adaption of feature modeling in more detail.

7.1 Customizable Services in Simulink

In hybrid system design, functionalities often reappear in many practical applications, e.g. filters or control systems like the temperature control system as shown in Figure 2.6. Ideally, such components could be encapsulated in Simulink subsystems to enable their reuse in other designs. However, the simple structure of Simulink subsystems, which does not allow for structured internal changes, makes it difficult to insert them into different contexts. Furthermore, to make statements about the behavior of such subsystems, it is necessary to look at their concrete internal structure. This contradicts interoperability and impedes reuse.

To solve these problems and facilitate systematic reuse, we introduce services that represent customizable components with defined interfaces. In comparison with traditional subsystems, our services are extended with a *feature model* to describe possible variations, and with *hybrid contracts* that precisely and formally describe their dynamic interface. We extend our previously introduced service concept by variability concepts.

Definition 7.1 (Customizable Service)**.** A customizable service s_{cust} is a tuple

$$s_{cust} = (sm, P_i, P_o, F, C)$$

where *sm* denotes a Simulink model that consists of blocks and signal lines. P_i contains incoming signals and changeable parameters of the service. Service parameters are constant after a designer has chosen a value for them, and input signals can change over time. P_o represents the set of output variables. The key elements of a customizable service are given by F that denotes the *feature model* describing its possible variations, and by C that contains its *hybrid contracts*.

7.2 Feature Modeling for Simulink Services

In this section, we present our extension of services by feature modeling. Feature models in general capture the dependencies of variants. The key idea is to model the changes between service variants and their dependencies. Our feature model of a customizable service provides us with a core model that can be extended or changed by features that can be enabled or disabled. The root feature of the feature model contains the core model. This allows us to capture different variants of the same core service systematically in the same model. Note that we also support the semi-automated generation of feature models by automatically computing model differences from a given set of variants. A designer can manually adapt a given Simulink model and these changes are used to automatically create a new feature in a feature model for this customizable service. The resulting feature model can be used to select which features should be activated and deactivated to automatically create a Simulink model that represents the corresponding service variant. This facilitates the reuse of our services since they can easily be adapted for different environments or requirements. We have developed the concepts for the feature modeling and automatic service instantiation as part of a Bachelor's thesis [Umo20].

In the following, we first introduce feature modeling in general and our feature model representation for Simulink services. Afterwards, we present how we integrate feature modeling into the design process. Lastly, we present how the feature models are supported in our verification process.

Feature Modeling

A feature model [Kan+90] is a tree structure $fm = (F, D)$ that represents the dependencies between different design variants.

Figure 7.1 illustrates a feature model for the design of a temperature control system. The root feature *Temperature Control* describes the system that is designed in this model. The child features enable changes to the control system when creating a variant of the system. To create a variant, it is mandatory to choose a type of *Cooling* for this system. In this case the cooling can be either *Active*, e.g. the system can control an air conditioner unit, or *Passive*, e.g. cooling means that the heating is turned off and the environment cools the temperature. One of these two must be chosen, but it is not possible to choose both at the same time. Furthermore, it is possible to consider a *Disturbance* when creating a variant. This feature is optional and can be a disturbance at the *Gain*, e.g. variations in the temperature of the environment influence the current temperature, or a disturbance at the sensor, e.g. the measured temperature can deviate from the actual temperature. If

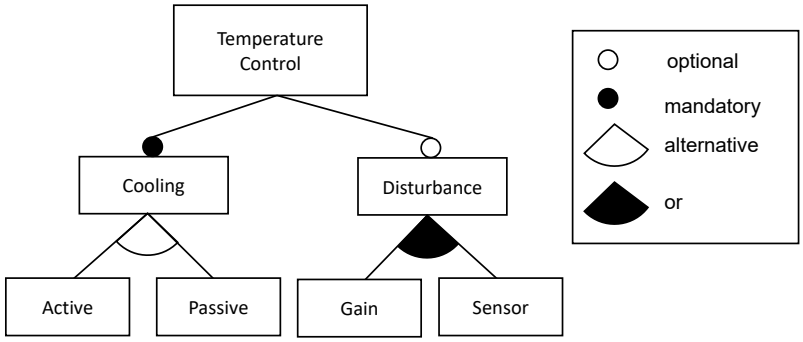


Figure 7.1: Example Feature Model for Temperature Control Variants

the *Disturbance* feature is active, at least one of these disturbances is present in the system.

Feature modeling in Simulink

To integrate feature modeling into Simulink, we introduce a feature model that contains variations of blocks in a Simulink system. The main idea of our feature model is that we define a core model (or *root feature*), which consists of Simulink blocks and signals. The core model, which is defined by the root feature, contains the Simulink blocks and signals that define the basic functionality of the system. This core model can be adapted by further blocks and signals by its child features. It is also possible to change existing blocks and signals in the child features. We use feature dependencies to group different features together, which provide similar extensions to the core model. To ensure that the executability requirements are met, we introduce the concept of a consistent Simulink model.

Definition 7.2 (Consistent Simulink model)**.** A Simulink model $S = (B, S)$ with blocks B and Signals S is *consistent* if the following holds:

$$\begin{aligned}
&\forall b_1, b_2 \in B, (b_1.id = b_2.id \rightarrow b_1 = b_2) \wedge \\
&\forall b \in B, (\forall p_{in} \in b.in, (\exists s \in S, (s.dst = p_{in}))) \wedge \\
&\forall s_1, s_2 \in S, (s_1.dst = s_2.dst \rightarrow s_1 = s_2) \wedge \\
&\forall b \in B, (\forall p_{out} \in b.out, (\exists s \in S, (s.src = p_{out})))
\end{aligned}$$

Whereas $b.id$ describes the unique name of a block, $b.in$ describes the set of input ports of a block, $b.out$ describes the set of output ports of a block, $s.src$ describes the source port of signal s and $s.dst$ describes the destination port of signal s. The first rule describes that each block name is unique. As second rule, we define that each input port of every block is the destination of at least one signal line. This is extended by the third rule, which states that each input port has exactly one signal as input. The fourth rule states that each output port is the source of a signal line. Note that it is possible that an output port is the source of multiple signals. This represents branching signal lines in Simulink. We only allow for signals to have one destination. To represent branching signals, we use multiple signals with the same source port.

Definition 7.3 (Simulink Feature Model)**.** Our feature model in Simulink consists of:

$$fm = (F, D)$$

Where F is a set of features and D is the set of dependencies between features. A feature $f \in F$ consists of the following

$$f = (B_a, B_r, B_c, S_a, S_r)$$

Where B_a are Simulink blocks that are added to the system, B_r are Simulink blocks that are removed from the system, B_c is a set of blocks that are changed with the respective changes. S_a are Simulink signals that are added and S_r are signals that are removed. When a block is added it is identified by a unique name, its type and additional parameters depending on the block type. To increase the readability, we omit the additional parameters in the following examples and depict new blocks as *BlockName* [*BlockType*]. The blocks to be removed can be identified by only the block names. Note that Simulink only allows unique block names. Changes in a block are given by the block name and a parameter list that contains the new block parameters. Note that we do not allow for changing the type of a block in B_c. However, changing the block type can be achieved by removing the existing block and adding a new one that is of the desired type. Signals model the connections between a source block and a destination block. Since blocks can contain multiple input and output ports, we also consider the port number the signal is connected to. The numbering of ports in a block starts at 1. In the following, we depict signals in the form *SourceBlockName.PortNumber* $\rightarrow$ *DestinationBlockName.PortNumber*. Note that the left-hand side always denotes block outputs and the right-hand side always denotes block inputs, see also *signal routing* in Section 2.4.

The root feature of a feature model can only add blocks. Therefore, the sets of removed blocks and signals and the set of changed blocks are empty. A child feature can remove or change blocks that are part of the added blocks of its ancestor features.

Instantiation of a variant. To obtain a Simulink model of a customizable service that we can insert into other systems, we introduce the concept of instantiating variants of services.

Definition 7.4 (Variant of a Simulink Service)**.** A *variant* of a Simulink service $i = (B, S)$ for a given set of active features F_a for a feature model fm is a set B of Simulink blocks and signals S that are part of the features in the feature model. All activated features provide the blocks and signal lines of the service variant and the following holds:

$$\forall b \in B, \exists f \in fm.F, ((f \in F_a) \wedge (b \in f.B_a \vee b \in f.B_c))$$
$$\forall s \in S, \exists f \in fm.F, ((f \in F_a) \wedge (s \in f.S_a))$$

The first line describes that all blocks in the variant are either added to the Simulink model by an active feature or are created by the change of a block. The second line describes that all signal lines in the variant are added by an active feature.

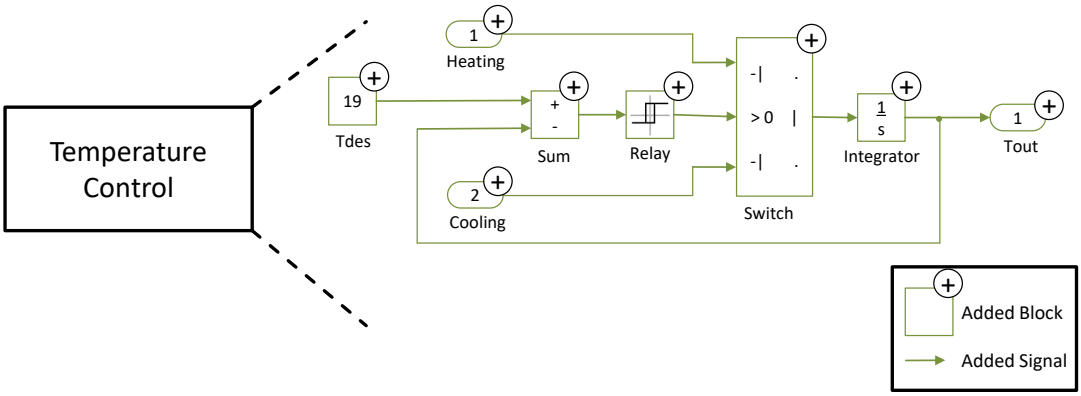


Figure 7.2: Root Feature for Temperature Control System

We define a variant construction algorithm, which is shown in Listing 7.1, to automatically generate variants from a given feature model. The input is a feature model fm. As intermediates, we have a list of features F and the resulting Simulink variant i. The function *fm.getRoot()* returns the single root feature of the feature model fm.

```
1   F ← fm.getRoot();
2   i.B = {};
3   i.S = {};
4   while (¬F.empty()) {
5       f = F.pop();
6       i.B = i.B \ f.Br;
7       i.S = i.S \ f.Sr;
8       adaptBlocks(i.B, f.Bc);
9       i.B ← f.Ba;
10      i.S ← f.Sa;
11      F = F ∪ getActivatedChildren(f);
12  }
```

Listing 7.1: Variant Construction Algorithm

First, we add the root feature to the set of features F to visit (Line 1). At the start, the sets of blocks and signals of the variant are empty (Lines 2 and 3). While there are still unvisited features in F, we perform the following: We take the next feature (Line 5) and remove all blocks and signals from the variant that are removed by this feature (Line 6 and 7). Afterwards, we change the blocks of the variant according to the block changes of the current feature (Line 8). Then, all blocks and signals are added to the instance according to the feature (Lines 9 and 10). Lastly, we add all activated child features to the set of current features F. When the set of features F is empty, we have visited all enabled features and the variant i contains all blocks and signals.

Temperature Control. In Figure 7.2, we depict the temperature control system as root feature in a feature model. The root node (which initializes the feature model with the root feature *Temperature Control*) is depicted as a box on the left, the underlying Simulink model is depicted on the right. So far, the feature model

contains only the root node and only one variant that can be instantiated. The added blocks consist of all depicted blocks:

$$B_a = \{Heating[Inport], Cooling[Inport], Tdes[Constant], Sum[Sum], Relay[Relay], Switch[Switch], Integrator[Integrator], Tout[Outport]\}$$

The added signals are:

$$S_a = \{Heating.1 \rightarrow Switch.1, Cooling.1 \rightarrow Switch.3, Tdes.1 \rightarrow Sum.1, Sum.1 \rightarrow Relay.1, Relay.1 \rightarrow Switch.2, Switch.1 \rightarrow Integrator.1, Integrator.1 \rightarrow Tout.1, Integrator.1 \rightarrow Sum.2\}$$

7.3 Variant Design in Simulink

In general, feature modeling allows for capturing the dependencies between variants of the same system. To integrate the idea of modeling of variants into Simulink designs, we enable the creation of Simulink models from a feature model. To ensure that the resulting models can be executed by the Simulink simulator, some properties are required by the Simulink model, for example, that identifiers are only used once.

During the design process of a customizable service, we aim at enabling the designer to create multiple variants of a service that can be encapsulated into the same Simulink service. This allows a designer to adapt a service for multiple execution contexts. If a set of active features in a feature model is valid according to its dependencies, it should always create a consistent Simulink model.

Definition 7.5 (Consistent Feature Model)**.** A feature model $fm = (F, D)$, where F is a set of features and D is a set of dependencies between features (optional, mandatory, alternative, or), is *consistent* if the following holds:

1. it contains at least one valid selection of active features
2. Simulink variants that are valid selections of active features are consistent

Note that inconsistent Simulink models can (syntactically) be defined but are rejected by our framework. To obtain consistent feature models, the designer should create a fresh feature model that only consists of a root feature and then add features step by step such that consistency is always ensured.

In the following, we illustrate this by extending the Simulink model of Figure 7.2 with the features that are depicted in Figure 7.1.

Optional Relation

An optional relation does not have to be activated if the parent feature is active. That means, if the feature is deactivated, the parent must create a consistent Simulink model. To add a new valid optional relation to an existing feature model, we consider the following: New blocks can be freely added, as long as they are connected via new signal lines. A new signal line can only have an existing block

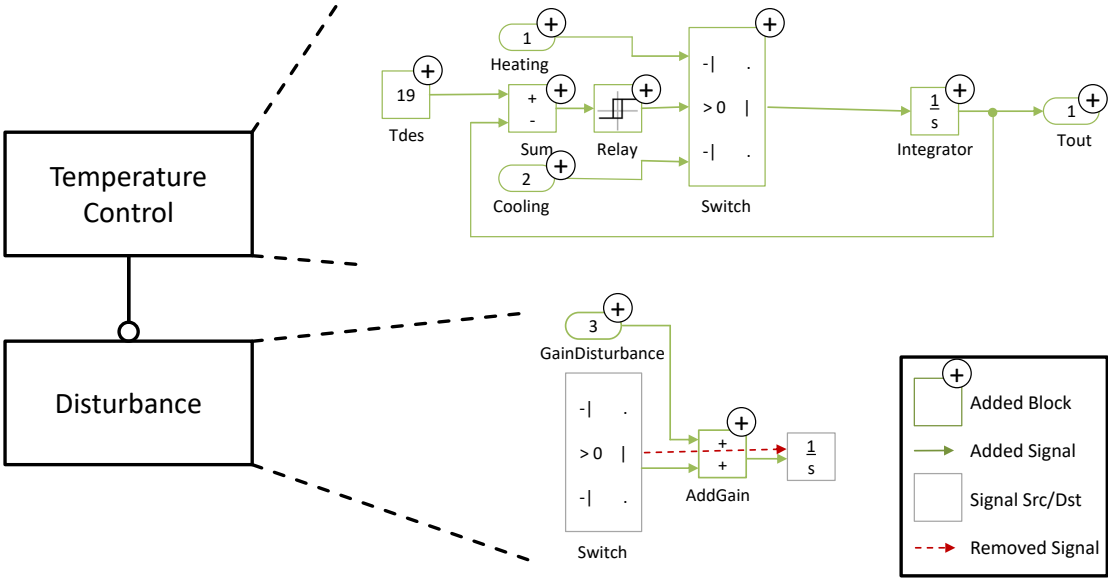


Figure 7.3: Optional Disturbance Feature

as destination if the existing line that has the block as target is removed by this feature. If the number of ports of a block is increased, the new ports are connected by new signal lines. If a block is removed or its number of ports is decreased, the connected signal lines are removed. Resulting open ports are connected by new signal lines.

Temperature Control. For the temperature example, we introduce the optional feature of an external *Disturbance*. As concrete disturbance, we model a disturbance at the *Gain* of the integrator. That represents influences of the environment to the current temperature. To achieve this, we first remove the signal that connects the switch to the integrator to free the input port of the integrator. Then we add a new input that represents the influence of the environment and a sum block to add this new input to the heating or cooling value at the output of the switch block. Lastly, we connect all blocks with signals and obtain:

$$
\begin{aligned}
B_a =& \{GainDisturbance[Inport], AddGain[Sum]\} \\
B_r =& \{\} \\
B_c =& \{\} \\
S_a =& \{GainDisturbance.1 \rightarrow AddGain.1, Switch.1 \rightarrow AddGain.2, \\
& \quad AddGain.1 \rightarrow Integrator.1\} \\
S_r =& \{Switch.1 \rightarrow Integrator.1\}
\end{aligned}
$$

The resulting feature model is depicted in Figure 7.3. Dashed arrows represent removed signal lines.

Mandatory Relation

A mandatory relation must be activated whenever the parent feature is active. It can be used to encapsulate blocks or signals of its parent features to achieve better

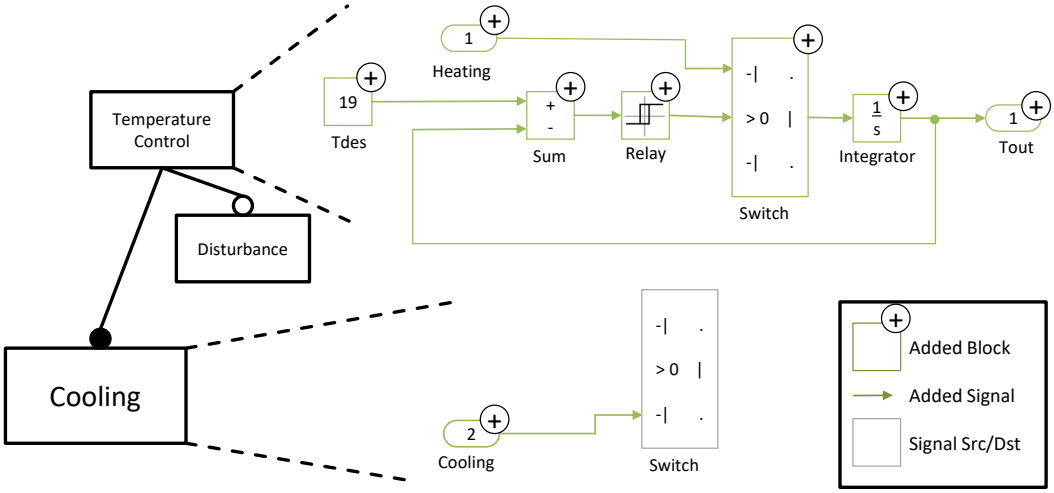


Figure 7.4: Mandatory Cooling Feature

comprehensibility. The following use of a mandatory feature is possible: During creation of the feature, we can transfer blocks and signals that are added by its ancestor features to the added blocks of this mandatory feature. That means that these blocks and signals are removed from the set of added blocks or signals of the ancestor feature. Note that this does not mean that the blocks and signals are added to the removed blocks or signals of the new feature, instead they are not present anymore in the added blocks and signals of the ancestor feature.

Temperature Control. For the temperature example, we encapsulate *Cooling* in its own mandatory feature. Note that this does not introduce new functionality into the model and only moves blocks and signals from the root feature into a new feature. We use the mandatory feature to encapsulate the blocks that influence the cooling in a new feature. This improves the comprehensibility of the feature model and simplifies changes to the cooling for other variants. The resulting feature is the following:

$$
\begin{aligned}
B_a &= \{Cooling[Inport\} \\
B_r &= \{\} \\
B_c &= \{\} \\
S_a &= \{Cooling.1 \rightarrow Switch.3\} \\
S_r &= \{\}
\end{aligned}
$$

Figure 7.4 shows the resulting model.

Alternative Relation

An alternative relation consists of multiple child features. If the parent feature is active, exactly one of the child features is activated. Therefore, it is possible to model different conflicting features with an alternative, e.g., the child features add signals that have the same destination port. The discussion of the mandatory feature can be applied in the following way: Blocks and signals that are added by

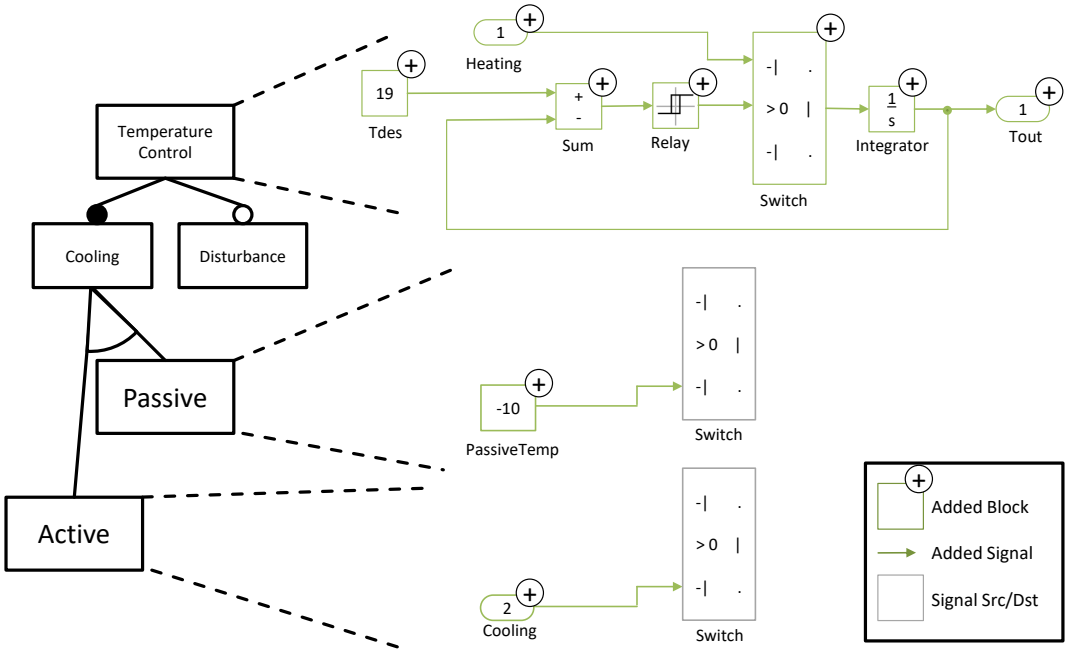


Figure 7.5: Alternative Cooling Feature

the parent features can be transferred into at least one of the child features in the alternative. The siblings in the alternative relation can also add the same blocks and signals or add new ones. It is required that the siblings add connections to all remaining ports for which the connected signals where transferred.

Temperature Control. In the temperature control system, the current cooling represents *Active* cooling. We add an alternative cooling that represents *Passive* cooling. To do so, we transfer the blocks of the previously added cooling feature into a new *Active* feature. This results in the *Cooling* feature to be empty and the *Active* feature to contain all previous contents of the *Cooling* feature. In the new *Passive* feature in the alternative, we are required to connect a signal to the empty input port of the switch. For the passive cooling, we use a constant block that we connect to the open switch port. The resulting *Passive* cooling feature is defined as follows:

$$
\begin{aligned}
B_a &= \{PassiveTemp[Constant]\} \\
B_r &= \{\} \\
B_m &= \{\} \\
S_a &= \{PassiveTemp.1 \rightarrow Switch.3\} \\
S_r &= \{\}
\end{aligned}
$$

Figure 7.5 shows the resulting feature model and the content of the cooling features.

Or Relation

Or relations are similar to alternative relations with the difference that multiple children in the relation can be activated if the parent is active. This means that

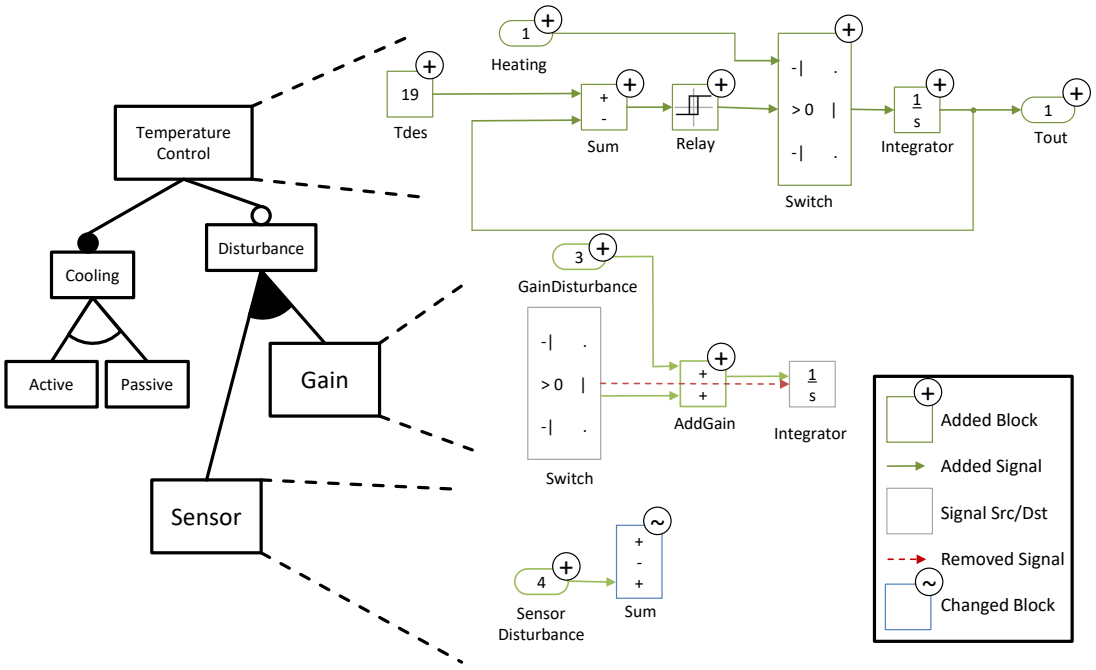


Figure 7.6: Or Disturbance Feature

it is not possible that the features in an *or* relation model conflicting features. Therefore, for each child of the *or* relation, the same rules as for the optional feature are required. Additionally, we can only transfer blocks of the parent features into one child of the *or* relation if the parent feature still creates a consistent Simulink model after the transfer.

Temperature Control. In the Temperature Control system, we add an *or* relation to the *Disturbance* feature. As first child, we transfer all content of the *Disturbance* feature into a new feature *Gain* to represent a possible disturbance of the change of the temperature. As second child, we introduce a disturbance at the *Sensor* that is represented by the feedback loop that compares the current temperature with the desired temperature. To achieve this, we add a new input for the disturbance at the measured temperature and add it to the *Sum* block that calculates the input of the *Relay*. Therefore, we alter the sum block to contain one additional input. Lastly, we connect the new input with the adapted sum block. The resulting sensor disturbance feature is described as follows:

$$
\begin{aligned}
B_a &= \{SensorDisturbance[Inport]\} \\
B_r &= \{\} \\
B_c &= \{Sum[Inputs = 3]\} \\
S_a &= \{SensorDisturbance.1 \rightarrow Sum.3\} \\
S_r &= \{\}
\end{aligned}
$$

The final feature model with the disturbance branch is depicted in Figure 7.6.

7.4 Hybrid Contracts with Features

The different feature variants described above refer to different behaviors. This also alters their interface behavior. For verification, the hybrid contracts need to reflect these different behaviors. One possible solution is to create new hybrid contracts for each possible variant of a given customizable service. However, this can significantly increase the verification effort for services with many variants. To reduce the verification effort, we (manually) extend our hybrid contracts by feature variables. Feature variables are boolean values that indicate whether a given feature is activated or deactivated in the resulting variant. This means that we can add a dedicated feature variable for each feature in the model. Note that we omit the feature variable of the root feature, since this feature must always be active and therefore the variable is always set to true.

The feature variables enable us to group our hybrid contracts. First, we have hybrid contracts that are independent of any feature variables. These hold for all variants of the customizable service. Second, we have feature-dependent contracts, which contain feature variables that influence parts of the hybrid contracts. These contracts also hold for all variants and contain variables that change according to the enabled features. Lastly, we have variant-dependent contracts, which only hold for variants that are created by choosing a defined set of features. These contracts only hold for single variants.

We extend our contracts by parameters that can be changed by features. These parameters are represented as variables in the contracts. During the instantiation of a variant, we choose a set of activated features. We extend our features by a set of contract parameter assignments that are performed if a feature is activated.

Only one assignment can be performed per parameter. This ensures that we do not obtain ambiguous contracts if more than one activated feature provides a value for the same contract parameter. Therefore, only features that are part of the same alternative can assign different values to the same parameter. If no value is assigned to a parameter, we define default assignments as part of the feature model that are assigned to all parameters that are left in the hybrid contract of a service variant. In general, we use this to define neutral values for the parameters that do not influence the behavior of the hybrid contract, e.g., the disturbance is set to zero if no disturbance is activated.

Temperature Control. For the temperature control system (Figure 2.6), we consider the previously introduced variants and the hybrid contract that ensures bounds for the temperature. In the feature model, we can choose whether we use active or passive cooling. Additionally, we can add disturbances to the gain or the sensor. In its basic variant with active cooling and no disturbances, the hybrid contract of the service is as follows (note that the exact values for the upper and lower bounds depend on the largest allowed stepsize):

$$\Phi_{in} = Heat_{On} \geq 0 \wedge Heat_{On} \leq 100 \wedge Heat_{Off} \geq -100 \wedge Heat_{Off} \leq 0$$
$$\Phi_{out} = Tout \leq Tdes + \delta \wedge Tout \geq Tdes - \delta$$

In the following, we change the contract to enable the use of one hybrid contract

for different variants. For the service with passive cooling, the input for $Heat_{Off}$ is replaced by a constant. Therefore, we replace $Heat_{Off}$ with a contract parameter *Cooling*. That means if the active cooling feature is selected, this variable is replaced by the input signal $Heat_{Off}$. If the passive cooling feature is selected, this variable is replaced by a constant cooling value that is determined by the cooling feature.

$$\Phi_{in} = Heat_{On} \geq 0 \wedge Heat_{On} \leq 100 \wedge Cooling \geq -100 \wedge Cooling \leq 0$$
$$\Phi_{out} = Tout \leq Tdes + \delta \wedge Tout \geq Tdes - \delta$$

For the basic variant, no disturbances are present. Disturbances can change the input gain at the $Integrator$ or the control input of the switch. For the disturbance at the $Integrator$, we introduce a new contract parameter $DisturbGain$. Since it directly influences the gain, which is already part of the assumptions Φ_{in}, we add this variable to the assumptions. If the GainDisturbance feature is disabled, this variable is set to 0. For the disturbance at the sensor, we need to extend the assumptions. The proof does not succeed with a disturbed sensor, since the sensor could measure a high temperature due to a large disturbance and would switch to $Cooling$, while the actual temperature is already at the lower bound. During the proof, we found that restricting the sensor disturbances enables a successful proof. The resulting contract encapsulates all possible behaviors of the variants and is modified according to the selected features:

$$\begin{aligned}\Phi_{in} =&(Heat_{On} + DisturbGain \geq 0 \wedge Heat_{On} + DisturbGain \leq 100 \wedge \\ & Cooling + DisturbGain \leq 0 \wedge Cooling + DisturbGain \geq -100 \wedge \\ & DisturbSensor \leq MAX_D \wedge DisturbSensor \geq MIN_D) \\ \Phi_{out} =&(Integrator \leq Tdes + \delta \wedge Integrator \geq Tdes - \delta)\end{aligned}$$

The aim of our customizable service approach is to ease the reuse of services in new systems and to facilitate compositional verification by providing hybrid contracts. The effort to (manually) create and verify hybrid contracts for all variants of a given system or service is only necessary once during the development of the customizable service and its variants. The hybrid contracts provide a precise reusable description for compositional verification without any additional effort.

7.5 Summary

In this chapter, we have presented our extension of services by feature modeling to enable variability for Simulink services. We encapsulate a Simulink model, a feature model and hybrid contracts into a customizable service. The feature model enables the customization of services and facilitates the reuse of services in other Simulink systems. The use of hybrid contracts in Simulink services enables us to provide a formal foundation for the use of services in larger systems. The feature model defines how the hybrid contracts of a customizable service are modified for each instance to enable the compositional verification of a larger system. Therefore, we combine the flexibility of customizable services with the verifiability that is provided by hybrid contracts.

8 Evaluation

In this chapter, we evaluate the performance of different aspects of our approach by using four different case studies. To enable the application of our service-oriented design and verification, we have implemented our approach and briefly present our implementation in Section 8.1. We evaluate the core concepts of our Simulink formalization with an extended temperature control system (Section 8.2). Note that this system consists of two interacting instances of the temperature control service that we use as running example. With this case study, we evaluate the application of our Simulink block transformation for Simulink models, consider the creation of hybrid contracts with the property templates and how the necessary variables can be added in the d$\mathcal{L}$ model with our extension functions. Furthermore, we evaluate the benefits of the use of hybrid contracts when considering multiple interacting services in a larger system. Additionally, we present how the use of services influences the feature model design of a system. We use a model of a Generic Infusion Pump (GIP) to evaluate our automatic generation of model invariants (Section 8.3). First, we manually define system properties and hybrid contracts for the components that are modeled as services of the GIP. Afterwards, we use our automatic invariant generation to evaluate how much manual effort can be saved for the verification of this case study. To demonstrate the applicability of our approach for larger case studies, we evaluate our approach with a distance warner provided by an industrial partner (Section 8.4). In recent work, we have extended our approach for intelligent components, which are components that use reinforcement learning to choose their behavior. One of the main strengths of our approach is that it provides the designer with systematic means for abstraction and thus has the potential to be applied to large and complex applications. We demonstrate this with a case study of an intelligent autonomous robot in a factory setting (Section 8.5). In Section 8.6, we summarize the results of our approach.

8.1 Simulink to d$\mathcal{L}$ Design and Verification Framework

We have fully implemented our framework in Java. An overview of our framework is given in Figure 8.1. To create a Simulink system model, a designer can either create a new model in Simulink that consists of the supported block set or use services from a *Service Library* and combine them with other Simulink blocks. We implemented a *Service Instantiator* that enables a user to automatically generate service instances. Service instances are added to a Simulink system as subsystems with a prefix in their name that marks them as services and defines the specific kind of service instance that they represent. The transformation tool *Simulink2dL* takes a Simulink model as input, uses the service library to find the contracts of used services, and automatically creates a d$\mathcal{L}$ model as file that can be loaded into the interactive theorem prover *KeYmaera X*. The d$\mathcal{L}$ model can be manually extended by properties of the system specifications. A valid proof in KeYmaera X for a system means that the proven property can be added as new contract for the given Simulink service instance in the *Service Library*. A large part of the implementation is online available as open source project[3].

[3]Project available at https://github.com/EmbSys-WWU/Simulink2dL

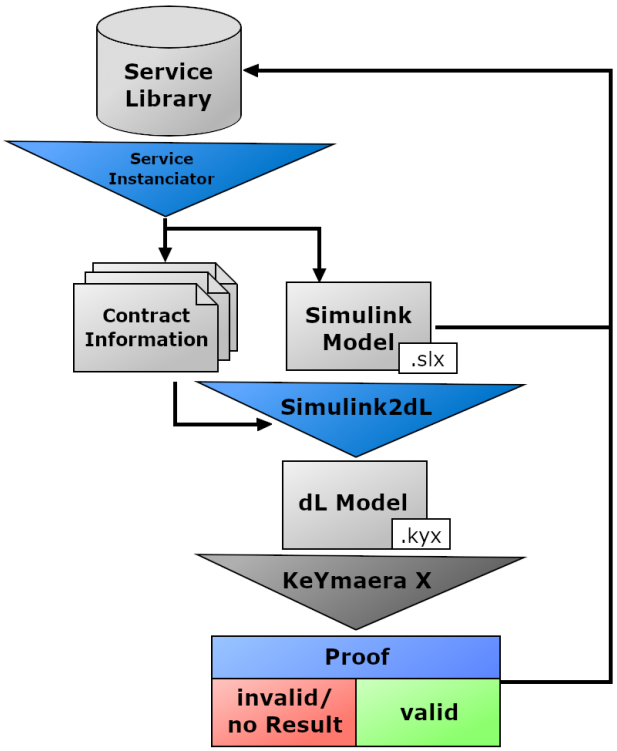


Figure 8.1: Workflow

Feature Models and Service Instantiator

To create a feature model, a designer manually creates a basis Simulink model. Thereafter, changes can be performed to the model to create a new variant, e.g., a disturbance is added. We can determine the changes automatically and generate a feature that represents these changes. The designer can then use either variant to integrate additional changes. These changes can also be independent of the previous changes and create new branches in the feature model. Therefore, it is not necessary to model all different variants and combinations of features explicitly. The resulting feature model can be added to our *Service Library* as new service.

To instantiate an instance of a customizable service, a user can chose a service from the *service library*. With the feature model of the chosen service, the user can enable or disable features for the desired context in the *Service Instantiator*. For a valid selection of features, the *Service Instantiator* can automatically generate a Simulink file that contains an instance of the service with the selected features, which can be used in the design of other systems.

Simulink2dL

The architecture of our *Simulink2dL* transformation tool is depicted in Figure 8.2. The implementation consists of 183 java classes and approximately 23k lines of code. We use an adapted version of the Simulink Parser *ConQAT* [CQS] to create an intermediate java representation of a given Simulink model. We first preprocess the model to find all service instances that are used in the model according to the prefixes in their names. If the user specifies that services should be replaced by their

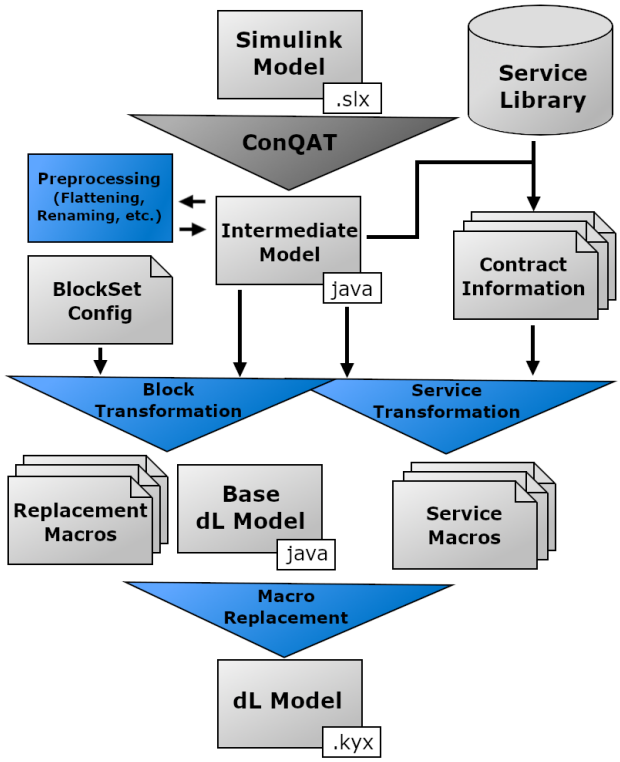


Figure 8.2: Simulink2dL

contracts, the services will be considered during the transformation. Otherwise, all subsystems are flattened to only consider atomic Simulink blocks. A *Block Transformation* creates a *Base dL Model* and *Replacement Macros* according to our transformation rules. For the transformation of Simulink blocks, we use the Factory Design Pattern to define the supported Simulink block set. All supported block types can be defined in a *BlockSet Config* file. A *Service Transformation* creates replacement macros that specify the *Contract Information* of the used services. The base model and all macros are used by our *Macro Replacement* to create the *dL Model* that represents the behavior of the given Simulink model.

8.2 Service-Oriented Design and Verification: Temperature Control System

In the following, we illustrate our transformation from Simulink to $\mathsf{d}\mathcal{L}$ using a larger temperature control system that consists of two instances of the temperature control service that we have previously introduced (Figure 2.6). We discuss the transformation results of the individual temperature control service. We verify properties for the larger system by using our compositional verification (Section 6.3) and compare it to a monolithic verification of the integrated system to show that the reuse of already obtained verification results decreases the verification effort. Lastly, we discuss present a feature model of the larger system that uses services.

8.2.1 Temperature Control Service

With our tool, we have fully automatically transformed the temperature control service (Figure 2.6) and have successfully verified that the temperature can be hold in a given range and that no rapid switching occurs. The transformation of the Simulink model into an equivalent dℒ representation took only a few seconds. The interactive verification within KeYmaera X took approximately five minutes for the correct temperature range and approximately one minute for the absence of rapid switching. The necessary manual interactions include the addition of preconditions, which restrict the possible values of the heating and cooling values, as well as the definition of loop invariants.

8.2.2 An integrated Environment-System Model

Cyber-physical systems consist of loosely coupled components that interact with each other and the physical environment. Our service-oriented approach allows us to model these components and the environment as individual instances of a customizable service. In the following, we use our service-oriented approach to model an integrated temperature control system where a heater with an individual temperature can be placed in a room to increase the temperature of the room. The model is depicted in Figure 8.3. The system contains 28 blocks and 34 signal lines. The temperature of the room can be changed by the ambient temperature or the current temperature of the heater. The temperature of the heater can rise by a given heating value of 37 or it can be cooled by the current room temperature.

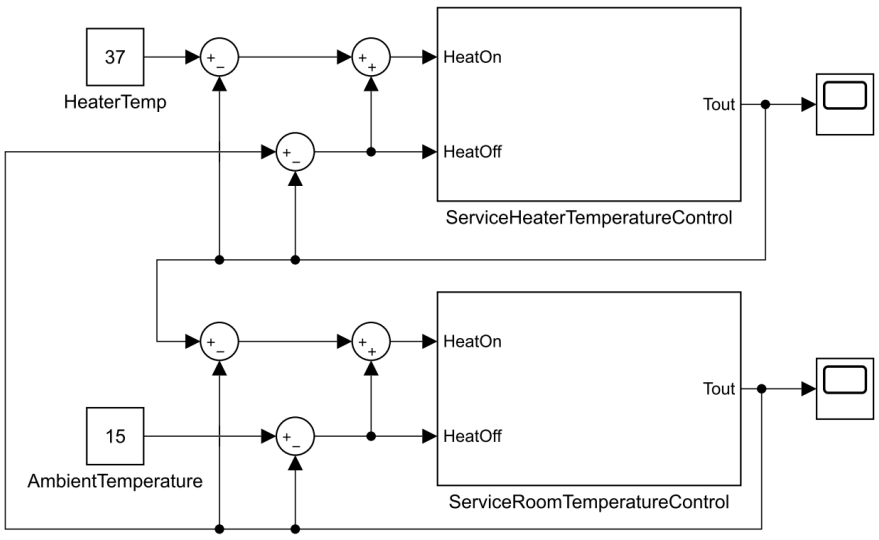


Figure 8.3: An integrated Environment-System Model

Feature Model

The individual temperature control services use the feature model that we have presented in Section 7.2. A feature model of the temperature control system does not consist of the feature models of the services, but it uses instances of these

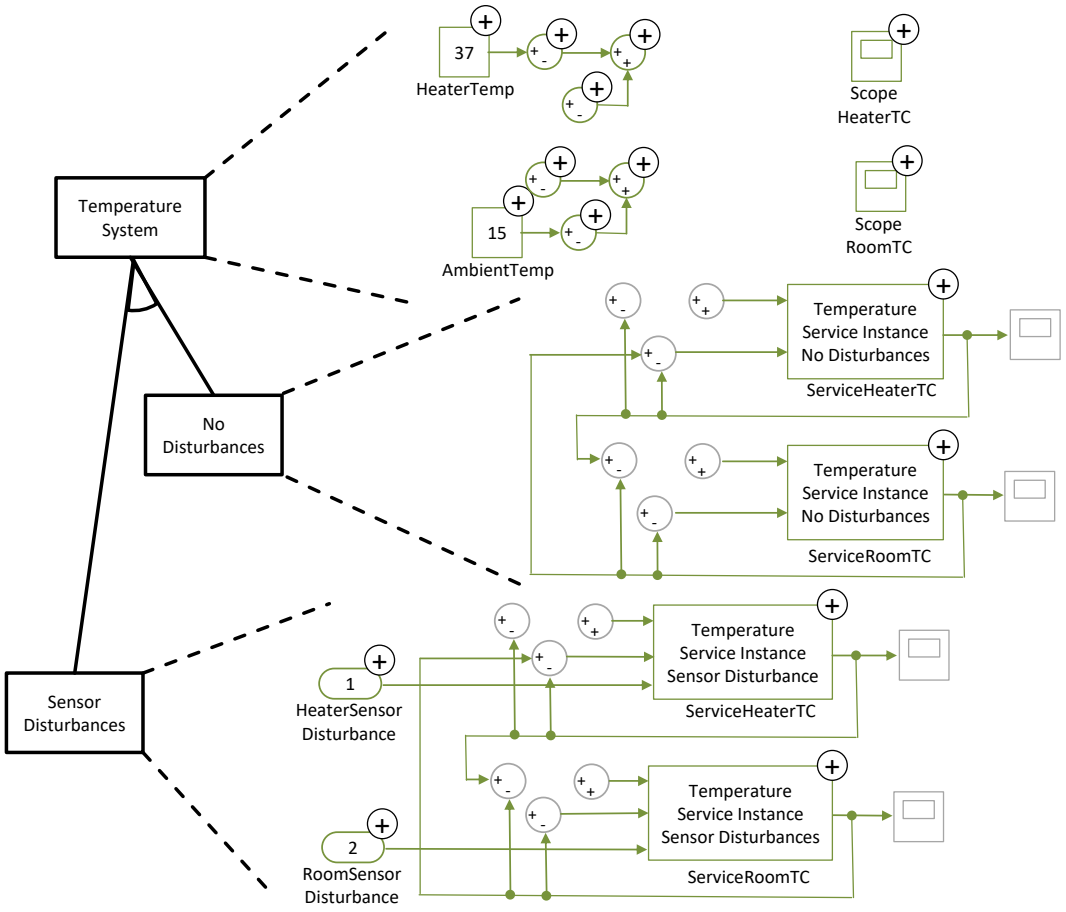


Figure 8.4: Optional Disturbance Feature

services as blocks. In the following, we create a feature model of the temperature control system that enables us to create two variants. The first variant consists of two services that do not contain any disturbances. The second variant takes two inputs that provide disturbances at the sensor of each service.

Figure 8.4 shows the resulting feature model. The root feature consists of all blocks without the temperature control services. Signals require a source and a destination, therefore all signals that are connected to one of the two services are not present in the root feature. As child features of the root, we create an alternative that enables us to choose between no disturbance or the aforementioned sensor disturbances. Both of these features add two instances of the temperature control service. These instances are created by explicitly choosing active features of in the feature model of the temperature control service. In the case of no disturbance, we select the *NoDisturbance* feature for the creation of the service instances. For the sensor disturbance, we select the *Sensor* feature for the disturbance. In both cases, we select *Active* as cooling feature. Additionally, we add two *Inport* blocks in the *SensorDisturbances* feature, which provide the values for the disturbance signals. Lastly, both features add the signals lines that connect the blocks of the root feature to the temperature control service instances.

Service-Oriented Design

We have created a library that implements the previously introduced temperature control service in Simulink. This service contains the presented blocks and is customizable via the described features. The hybrid contract of the service precisely describes its interface. Our customizable service enables us to reuse the same structure for all three components and customize their behavior and interface for their individual context. Therefore, we do not need to model the individual components.

Transformed Model

We have automatically transformed this system into dℒ. An excerpt of the simulation loop of the transformed model without disturbances is depicted in Listings 8.1.

```
1   {smallStep:=0.0; ScopeInput1:=HeaterTout;ScopeInput2:=RoomTout;
2     RoomHeating:=HeaterTout-RoomTout+15-RoomTout;
3     RoomCooling:=15-RoomTout;
4     HeaterHeating:=37-HeaterTout;
5     HeaterCooling:=RoomTout-HeaterTout;
6     RoomTout:=*;
7     HeaterTout:=*;
8     ?(((RoomHeating >= 0) & (RoomCooling <= 0)
9     & (RoomHeating <= 10) & (RoomCooling >= -10))
10    -> ((19.0 - 2.0 <= RoomTout) & (RoomTout <= 19.0 + 2.0)));
11    ?(((HeaterHeating >= 0) & (HeaterCooling <= 0)
12    & (HeaterHeating <= 10) & (HeaterCooling >= -10))
13    -> ((28.0 - 2.0 <= HeaterTout) & (HeaterTout <= 28.0 + 2.0)));
14    ...
15  }}*
```

Listing 8.1: Abstract Integrated Environment-System Model in dℒ

In Lines 2 to 5, the inputs for the services are assigned. In Lines 6 and 7, the outputs are set to arbitrary values. The contracts of the two services are in Lines 8 to 10 for the room and Lines 11 to 13 for the heater. Despite consisting of two instances of the service depicted in Listing 5.4, the abstracted system has far less lines to describe the behavior in dℒ.

Compositional Verification

We have verified in KeYmaera X that the temperature is kept in a given range and that no rapid switching occurs. To verify these properties, we have manually added two invariants to the simulation loop. For the temperature bounds, we use range properties as invariants to describe the upper and lower temperature bound. To show that no rapid switching occurs, the invariant contains two clock variables that keep track of the time since the relay switched its state. The minimum time between changes is described by a timing property. Furthermore, we use a second set of timing properties that capture that the heating or cooling amount since the last state switch is bounded. The verification times for the interactive verification in KeYmaera X are shown in Table 8.1. We can see that the verification time for the system with contracts is significantly reduced compared to the flattened concrete system. Even the verification time of system and service together is less than the

Table 8.1: Verification times in hh:mm

		Concrete System	Abstracted System	Service
Integrated Environment Model	temperature range	00:55	<0:01	0:05
	no rapid switching	-	0:06	0:01

time for the flattened system. Since we only need to verify the contract for both services once, we can reuse the verification result. For the rapid switching property of the concrete system, we could not even finish a proof, since KeYmaera X aborted due to an internal error, likely caused by the size of the model. With our service-oriented verification approach, we were able to verify both properties in less than half an hour.

8.3 Automated Invariant Generation: Generic Infusion Pump

In the following, we evaluate the verification process and the necessary manual interactions with a case study from the medical domain, namely a generic infusion pump (GIP) [GIP]. We present the Simulink model of a generic infusion pump (GIP), and the most crucial requirements this model has to satisfy. We discuss the manual interactions that are necessary in our service-oriented compositional verification process and discuss how the verification is supported by our automated invariant generation (Section 6.4). By comparing the automatically generated invariants with the invariants that we have manually defined during the proof of the system properties, we show that the automatic invariant generation can reduce the manual effort for the interactive proofs.

8.3.1 Generic Infusion Pump (GIP)

The generic infusion pump (GIP) project [GIP] is a research project to model and verify a wide range of infusion pump models. In our work, we use a model of the GIP that we have developed as part of a Bachelor's thesis [Che20]. To regulate the concentration of a drug in the blood of a patient, an infusion pump can perform different injections. A *bolus* is an injection of a large amount over a short period of time. A *basal* is an injection of low amounts of the drug over al long period of time.

Typical components of the infusion pump model are the infusion pump controller and the patient. The pump consists of sensors, input buttons, a tank, and the pump itself. The patient has a current concentration of the drug in her bloodstream and can generate input signals for the pump. An infusion pump has at least three different operation modes. First, the pump is turned off and no drug is injected. Second, a given basal rate is injected over a long time. Third, for a short time a high bolus rate is injected. The pump is controlled by the patient and can be programmed with different inputs. Note that a real pump could also be controlled by a doctor instead of the patient. The patient can set a basal and bolus rate for an infusion, change the batteries of the pump, change the tank for a new one or refill the tank.

The safe operation of a GIP is important since faulty behavior can harm the health of a patient. There are typically three major safety requirements: (1) No critical dosage should be administered. (2) To prevent overdose, if a critical concentration of the drug in the blood is measured, the administering of the drug should be stopped. (3) An alarm should be raised and the drug delivery should be stopped whenever the battery or the tank level is critically low.

GIP Model in Simulink

To develop a Simulink model of the GIP model in Simulink, we have followed a strictly service-oriented approach. We have decomposed the system into independent components that can be independently modeled, and, in particular, also later be independently verified. A service-oriented modeling approach increases the reusability of the components, enables distributed and joint development, and generally makes the structure of the model more transparent and the overall system easier to comprehend. In total, we have defined nine services:

1. *ServiceInputGenerator* simulates the input signals for the system.
2. *ServicePatientInput* prepares the input signal for the use by the controller by limiting the values to given bounds and ensuring that the signals are of the correct type.
3. *ServiceInputProcessing* processes the input signals.
4. *ServiceDrugConcentration* represents the concentration of the drug in the blood of the patient and its absorption over time.
5. *ServiceController* determines the operating mode of the pump.
6. *ServiceWarningGenerator* generates a warning if battery or tank level are critically low.
7. *ServicePump* represents the pump that performs the infusion.
8. *ServiceTank* models the internal tank of the pump that contains the drug.
9. *ServicePowerSource* represents the power supply of the pump.

The resulting structure of the overall GIP model in Simulink is shown in Figure 8.5. The system contains 133 blocks and 137 signal lines. Note that *ServiceInputGenerator*, *ServicePatientInput*, and *ServiceDrugConcentration* model the behavior and dynamics of the patient, while all other services model the infusion pump itself. Furthermore, note that the service structure is hierarchical, i.e., some services contain inner services, e.g. *ServiceController*. Overall, the system consists of nine different service types. The services *ServiceInputProcessing* and *ServiceWarningGenerator* are instantiated multiple times. The continuous dynamics are present by the services *ServiceDrugConcentration*, *ServiceTank* and *ServicePowerSource*.

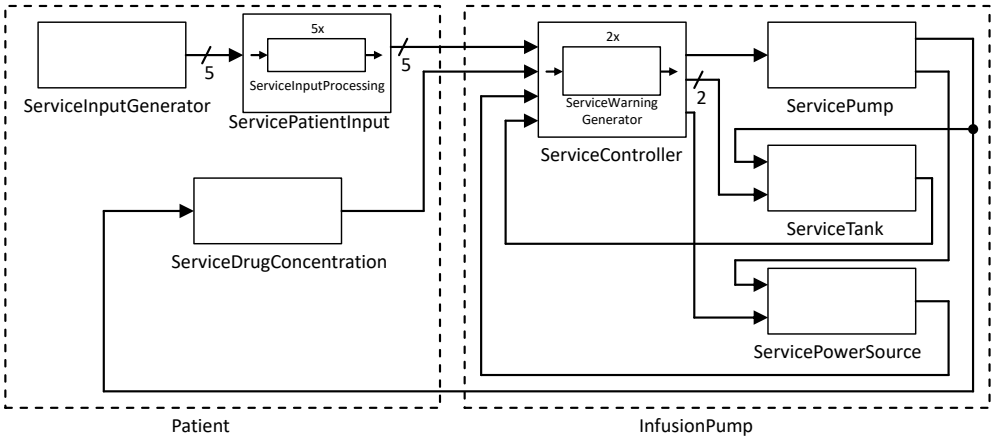


Figure 8.5: Structure of the GIP model in Simulink

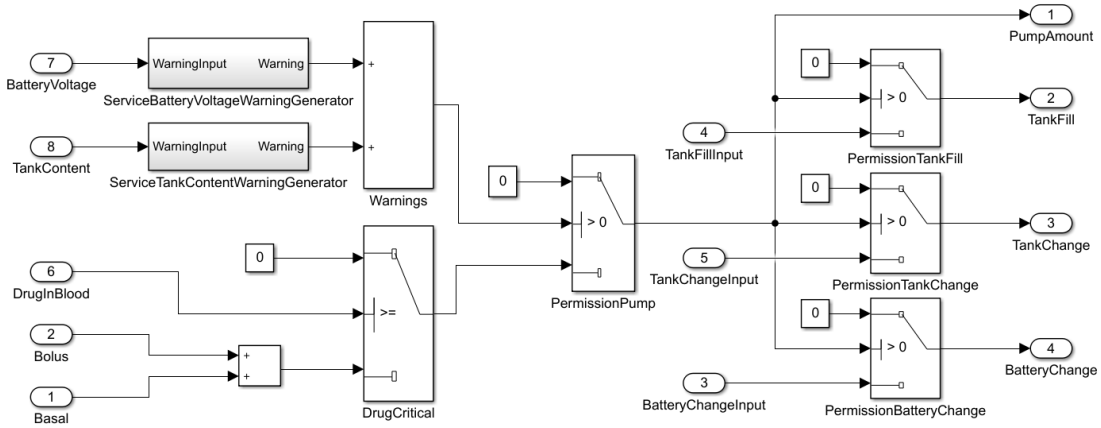
8.3.2 Automatic Generation of Hybrid Contracts

To evaluate the utility of an automatic generation of hybrid contracts, we first perform a verification of the system, where we manually define all hybrid contracts. Afterwards, we automatically generate hybrid contracts for the services in the system and compare them to the contracts that we used during the manual verification. This enables us to detect the guarantees and invariants that are missing in the automatic generation.

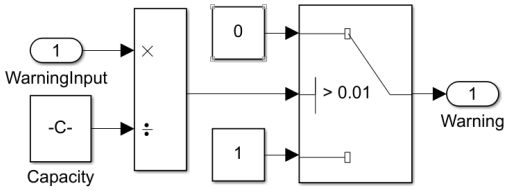
To compositionally verify the crucial safety properties of the Simulink model of the GIP described above, we have first manually defined hybrid contracts for all services that are present in the design. The hybrid contracts capture the interface behavior of each service, but abstract from inner details like internal computation steps. To illustrate our approach, we discuss the Simulink models and the corresponding hybrid contracts of three services in more detail, namely *ServiceWarningGenerator*, *ServiceController*, and *ServiceDrugConcentration*.

Warning Generator. The *ServiceWarningGenerator* is used twice, to give a warning whenever the energy level is critically low and whenever the drug tank is critically low. The Simulink model is depicted in Figure 8.6b. The value *Capacity* denotes the capacity of the tank or the battery. The value *CapacityWarning* denotes the percentage of the capacity at which the warning should be issued, i.e., at which the outgoing signal is set to 1. The contracts that describe the desired behavior of the warning generator are shown in the upper part of Table 8.2. *NoWarning* ensures that no warning is produced if the current value is above the critical value. *OutputWarning* guarantees that the warning output is set to 1 if the input value drops below the critical value. *BinaryOutput* guarantees that the output signal is binary and either has a value of 0 or 1.

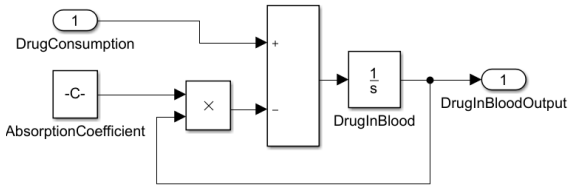
Controller. The *ServiceController* is the main controller of the infusion pump. Figure 8.6a shows the Simulink model. The controller has eight input signals and provides four output signals. *Basal* and *Bolus* represent the currently set basal rate and bolus. *DrugInBlood* represents the current concentration of the drug in the blood. *TankContent* represents the amount of drug in the tank, and *Battery-*



(a) ServiceController



(b) ServiceWarningGenerator



(c) ServiceDrugConcentration

Figure 8.6: Simulink models of Warning Generator, Controller and Drug Concentration

Voltage the voltage of the power supply. *BatteryChangeInput*, *TankChangeInput*, *TankFillInput* are user inputs that signal that a battery change, a change of the drug tank or a refill of the drug tank is requested. *PumpAmount* determines the current rate of infusion, *BatteryChange*, *TankChange* and *TankFill* signal that the battery or tank can be changed or refilled, respectively. The controller determines the current mode of operation and checks that no critical states are reached, like a low level of the drug tank. The change of the battery or tank, or refilling, is only possible if there is no ongoing infusion. The contracts that describe the desired behavior of the controller are shown in the middle part of Table 8.2. *CriticalConcentration* ensures that there is no pumping in progress when the drug concentration in the blood of the patient exceeds a critical value. *ValidPumping* states that the pump is only turned on if no critical states are reached, if there is actually a basal or bolus requested by the user, and if there is currently no battery change, tank change or tank refill in progress.

Drug Concentration. The *ServiceDrugConcentration* models the concentration of the drug in the bloodstream of the patient. Figure 8.6c shows the Simulink model. The current drug concentration in the blood of the patient is calculated by integrating the current drug flow rate (*DrugConsumption*) over time and the current concentration reduced by the absorption of the drug by body tissue. The signal *DrugInBloodOutput* provides the current concentration of the drug in the blood. Note that we use *DrugInBlood_Start* to denote the drug concentration at the beginning of each simulation loop, and thus can reason about changes in the drug concentration during one cycle. The contracts that describe the desired behavior

Table 8.2: Contracts of Warning Generator, Controller and Drug Concentration

Contract	Assumptions	Guarantees
Warning Generator		
NoWarning	WarningInput > CapacityWarning · Capacity	Warning = 0
OutputWarning	WarningInput ≤ CapacityWarning · Capacity	Warning = 1
BinaryOutput	true	Warning = 0 ∨ Warning = 1
Controller		
CriticalConcentration	DrugInBlood ≥ DrugCritical	PumpAmount = 0
ValidPumping	Basal > 0 ∧ Bolus ≥ 0 DrugInBlood < DrugCritical ∧ BatteryVoltage > VoltCritical ∧ TankContent > TankCritical	PumpAmount > 0 ∧ BatteryChange = 0 ∧ TankChange = 0 ∧ TankFill = 0
Drug Concentration		
NoInfusion	DrugConsumption = 0	DrugInBlood_Start ≥ DrugInBlood
ConcentrationIncrease	DrugConsumption > DrugInBlood * AbsorptionCoeff.	DrugInBlood_Start ≤ DrugInBlood
ConcentrationDecrease	DrugConsumption < DrugInBlood * AbsorptionCoeff.	DrugInBlood_Start ≥ DrugInBlood
ConcentrationPositive	DrugConsumption ≥ 0	DrugInBlood ≥ 0

of the drug concentration are shown in the lower part of Table 8.2. *NoInfusion* ensures that the concentration of the drug in the blood of the patient decreases if the pump is inactive. *ConcentrationIncrease* and *ConcentrationDecrease* ensure the concentration of the drug increases if the currently administered drug amount is larger than the absorption and vice versa. *ConcentrationPositive* ensures that the concentration of the drug in the blood is always larger than zero if some drug is administered.

8.3.3 Compositional Verification

We have successfully verified that the crucial safety requirements defined above are satisfied under all circumstances by our GIP model using our service-oriented design and verification. Namely, we have shown that no infusion occurs in the system if the drug concentration has reached a critical amount. Furthermore, we have shown that warnings are produced if the battery or the drug tank level fall below critical capacities. To this end, we have written down contracts for all services. Note that for our GIP, the previously defined system requirements directly correspond to the guarantees of the service contracts defined in Table 8.2. Thus, the contract of the overall system is a conjunction of these service guarantees. For compositional verification, we have used our Simulink to $\mathsf{d}\mathcal{L}$ transformation together with KeYmaera X to interactively verify that the connected services satisfy their contracts. Finally, we have built an abstract $\mathsf{d}\mathcal{L}$ model of the overall system

Table 8.3: Interactive Verification Results

Service	Total Invariants	Automatically Generated Invariants
Warning Generator	3	2
Input Processing	2	2
Controller	14	11
Pump	8	6
Tank	8	4
Power Source	5	2
Input Generator	0	0
Patient Input	10	10
Drug in Blood	9	7
Total	59	44

where all services are replaced by their contract to verify the system requirements. While the Simulink model uses fixed values for model parameters, e.g., the tank capacities and the possible pumping range, we have verified that the safety properties hold for arbitrary values for these parameters. To this end, we have replaced the corresponding fixed values by variables and added the assumption that describe these parameters, like that the capacities are greater than zero and that the pumping values are always in a certain range.

Manual Effort. With the interactive verification in KeYmaera X, we have proven that our GIP model satisfies its safety requirements. The second column of Table 8.3 shows the total amount of invariants that we have defined for the interactive verification of the GIP. In total, we have defined 59 invariants. The overall interactive verification of the GIP model took 25 person hours. Note that some of the invariants were only needed to generalize the proof for arbitrary parameter values of the pump. The input generator can produce arbitrary signals that are processed by the patient input service. Output signals without any guarantees represent arbitrary signals, therefore the input generator service has no invariants.

8.4 Industrial Application: Distance Warner

Our third case study is an industrial example of an automotive hybrid control system, namely a multi-object distance control system. With this larger case study, we show the practical applicability of our approach. While we could not perform a proof in KeYmaera X of the flattened system due to the size of the $\mathsf{d}\mathcal{L}$ model, we are able to prove safety properties of this case study with our compositional verification (Section 6.3).

Multi-object distance warner

The distance warner measures the relative speed of up to two leading vehicles and computes the relative distances. The distance is calculated in calculation cycles and each calculation cycle is performed over a given amount of time steps.

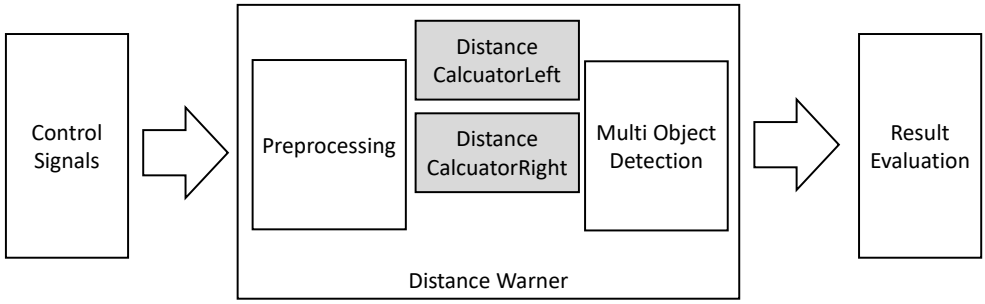


Figure 8.7: Architecture of the Distance Warner

The architecture of the model is depicted in Figure 8.7. Furthermore, continuous integrators are used to compute the relative distance as a dynamic function of the relative speed. The system contains 263 blocks and 364 signal lines. A time-discrete variant of this case study was also used in [HRB13], but the authors were not able to cope with the original hybrid version. In our evaluation, we have used the original hybrid version, and used our transformation for the core component of the system, namely a distance calculator, which comprises 18 blocks (including 5 time-discrete, 1 time-continuous, and 3 control flow blocks) and 23 signal lines. To enable contract-based verification, we have encapsulated a core component of the distance warner, namely the distance calculator, which is used twice in the overall system, as a service.

Verification

For the distance calculator, we verified the following safety properties: 1) the distance calculator does not produce overflows, and 2) the distance calculator performs a correct gain behavior, where the gain of the outgoing discrete signal is depending on the incoming continuous signal. To show the overflow property, we have added a range property as invariant. For the property that considers the gain behavior, we have added observer variables that keep track of the current sign of the input signal. The main property of the gain behavior is formulated as dynamic property that compares the last output state to the current one. The verification times are shown in the two bottom rows of Table 8.4. Note that we first detected a bug in the original system, where an overflow was possible at an integrator in the system. We detected this bug within 20 minutes in the interactive verification in KeYmaera X and subsequently used a patched version of the system where the integrator is saturated. For the corrected system, we were not able to verify any properties on the concrete system, since KeYmaera X crashed when starting a new proof. We assume that this is caused by the size of the model. For the abstracted system with the distance calculator as a service, we were able to show both desired properties for all possible input scenarios in less than 8 hours of interactive verification.

Table 8.4: Verification times in [hh:]mm:ss

		Concrete System	Abstracted System	Service
Distance Warner	no overflows	not loadable	30:00	19:00
	correct gain behavior	not loadable	24:00	7:00:00

Absence of Overflows

To verify the absence of overflows, we have introduced global constants *MINVAL* and *MAXVAL*. As shown in [HRB13], the original model actually produces an overflow. To produce a counter-example that demonstrates this faulty behavior, we have used the following requirements specification

$$< \cdot > Integrator < MINVAL \lor Integrator > MAXVAL$$

For the interactive verification with KeYmaera X, we were able to produce a counter-example in 20 minutes. To account for the over-approximations during the transformation, we have checked whether this was a spurious counter example, by using the information provided by the counter example as inputs for the Simulink simulation. To prevent the overflow, we have changed the integrator in the model to a bounded integrator, which holds its output value if it would rise above or fall below specified values. With the corrected model, we have then shown the absence of overflows using the following requirements specification:

$$[\cdot]\, Integrator \geq MINVAL \land Integrator \leq MAXVAL$$

We have verified this specification interactively in KeYmaera X in 21 minutes.

Increasing and decreasing distance

A major advantage of our approach is that we can not only verify static properties like the absence of overflows, but also dynamic properties, i.e., dynamic relations between inputs and outputs. To illustrate this, we have verified that, under the condition that there is no zero-crossing during the current measurement cycle time, a positive relative speed measurement causes an increase or no change in the calculated distance at the output. Since we use a bounded integrator to prevent overflows, a positive measurement causes no change at the output if the upper bound of the integrator value is reached.

$$[\cdot]\, gainPrevious > 0 \land noZeroCrossing \implies relativeDistance \geq 0$$

We have added the observer variable *noZeroCrossing* into the $\mathsf{d}\mathcal{L}$ model. This variable checks that the input relative speed measurement does not change its sign during one calculation cycle of the distance calculator. An analogous formula can be used for a negative relative speed and a decreasing distance. To enable the proof, we have added state variables, which store the current sign and detect whether a zero crossing occurred. To not change the system behavior, we only add hybrid programs that are assignments to these new variables or nondeterministic choices of the form $\{?(c); \alpha; \cup ?(\neg c); \beta;\}$, where c is a condition, and α and β are hybrid programs of the just defined form or empty. We were able to prove the desired properties with KeYmaera X interactively in 7 hours.

8.5 Formal Verification of Intelligent Hybrid Systems: Autonomous Robot

The key idea to enable compositional verification within our service-oriented design and verification approach are hybrid contracts. Hybrid contracts enable us to abstract from the inner details of (verified) services in system verification, which significantly reduces the verification effort. In our compositional verification approach, we formally verify that each service adheres to its hybrid contract and have formally shown that we can soundly replace services by their hybrid contract afterwards. The concept of hybrid contracts does, however, open additional opportunities for complex systems. While we can only guarantee for verified services that a system satisfies safety properties under all circumstances, safe system behavior can also be ensured at runtime using unverified services together with runtime monitoring, as shown in [FP18, ALH21]. The latter approach has the major advantage that we can ensure safety at runtime even for systems with components, whose behavior is hard or even impossible to verify. As example in the following, we use a reinforcement learning (RL) controller in an autonomous robot as a learning component in a complex control system. We can formally verify the correctness of such systems under the assumption that the RL component adheres to its contract and use runtime monitoring to make sure that this assumption holds at runtime. As our hybrid contracts precisely capture the safe behavior of the unverified components, we can also use it to generate runtime monitors automatically.

We have examined in the context of a Master's thesis [Ade20] and published in [ALH21, HAL21] how to ensure safe behavior of an autonomous factory robot that uses reinforcement learning to determine its behavior. In the following, we present the case study of an autonomous factory robot and define hybrid contracts that define safe behavior. We use our service-oriented design process to create the components in the system. For all known components, we verify that they provide guarantees and fulfill their hybrid contracts. For the overall system, we show that no collisions occur if all components fulfill their hybrid contracts. Therefore, we can ensure that no collisions occur if the unknown RL service also fulfills its contract. To evaluate the applicability of this assumptions, we simulate the whole system in Simulink and use a runtime monitor to ensure that the learning component always fulfills its hybrid contracts. We present that this application of our service-oriented design shows very promising results by simulating the robot in different factory environments. The basic system with the autonomous robot and two opponents contains more than 700 blocks and more than 850 signal lines.

8.5.1 Reinforcement Learning

During Reinforcement Learning (RL), an agent learns its behavior trough interaction with an environment and rewards that determine how beneficial its action was. This can be formalized as Markov decision processes (MDP) [SB18]. A basic visualization is depicted in Figure 8.8. The MDP consists of states S, actions A and rewards R. At discrete time steps, the agent observes the current state $s_t \in S$ of the environment and chooses an action $a_t \in A$. During the next observation step $t + 1$, the agent also receives a reward $r_{t+1} \in R$, which depends on the state and action that were chosen. According to the reward, the agent updates an internal

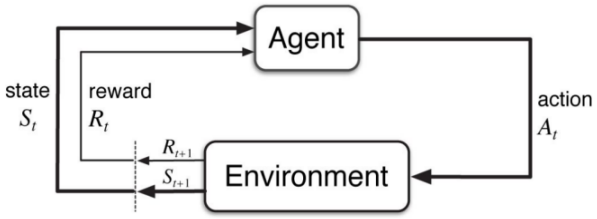


Figure 8.8: RL via MDP. Source: [SB18]

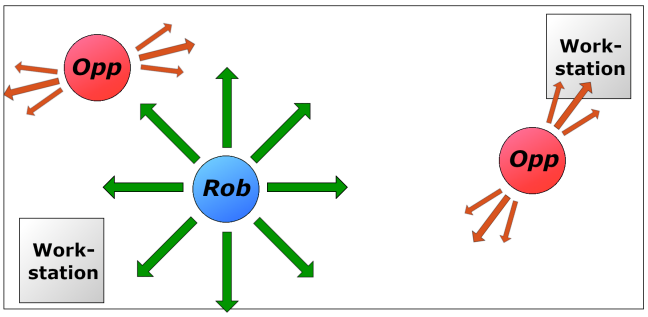


Figure 8.9: Autonomous factory robot

model that determines the actions that will be chosen for future timesteps.

The actions are chosen according to a probability distribution and therefore, it is difficult to formally predict the behavior of the agent and formalize the probabilistic system behavior.

8.5.2 Autonomous Robot in a Factory

The general concept of the system is shown in Figure 8.9. The autonomous robot *Rob* should move between different *Workstations*. There are also different opponents *Opp* that move inside the factory. The autonomous robot can choose different directions for its next movements. Its goal is to reach its next workstation while avoiding collisions with opponents.

The robot is rewarded for approaching its next workstation and penalized for interfering with opponents. If the robot enters a safety threshold of an opponent, the opponent will increase the distance between *Rob* and itself. The robot and its opponents have different clocks that determine their discrete time steps.

The Simulink model of the factory is depicted in Figure 8.10. The system consists of the RL robot, opponents, and job schedulers, which determine the next target workstation for each robot. The inputs of the opponents are their current goal and the current position of the RL robot. The inputs of the RL robot are its current goal and all positions of all opponents.

The Simulink model of the autonomous robot is depicted in Figure 8.11. The robot has sensors that measure the relative position of the other robots, a motor that determines its velocity from the control actions of the RL agent, integrators that continuously change its position, an *RL Agent* block that chooses the actions of the robot and an info service that determines current workstate and generates rewards accordingly.

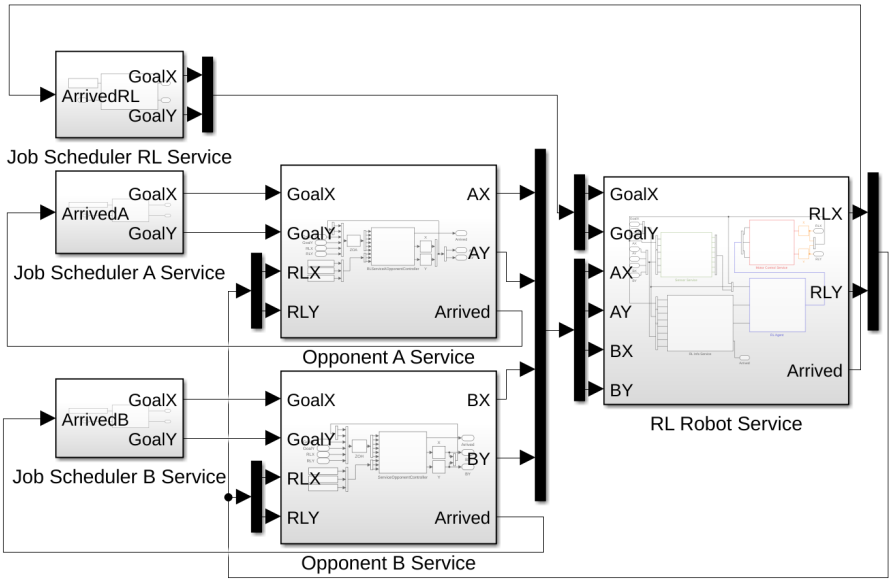


Figure 8.10: Simulink model of factory

8.5.3 Hybrid Contracts for Runtime-Monitoring

To ensure that the hybrid contracts that are used for runtime-monitoring only allow safe behavior, we first verify that these hybrid contracts ensure safe behavior. We verify the model under the assumption that the contract holds for the RL agent. Note that the Simulink model is parameterized in the sense that workspace variables are used for the maximum velocity of the opponents, minimal safety distances and thresholds, as well as sampling times, which can be set individually for each traffic participant. We can model different variants of these values as features for an opponent service. This allows us, to instantiate different opponents without creating entire new models. Furthermore, by updating the transformed model of the opponent to only use symbolic values for these parameters instead of concrete values, the resulting proof holds for all possible input scenarios and parameters.

Contract for the RL Robot. To find contracts for the RL robot that ensure safe system behavior, we have created a black box service inside the RL robot that represents its RL component. This component is assumed to always chose safe actions that are given by a hybrid contract. Later on, this behavior is enforced by a runtime monitor. Therefore, we can ignore the inner RL behavior of the RL robot and only consider its hybrid contracts in the system verification. In the system verification, we first define system properties that define safe behavior of the overall system. Afterwards, we transform the Simulink model into $\mathsf{d}\mathcal{L}$ and try to prove safe system behavior. To achieve this, we need to successively adapt the hybrid contracts of the RL robot, until they are strong enough to ensure safe system behavior.

We have created a contract for the RL robot that describes that it stops at a safe distance before any opponent's threshold.We have used our extension functions to create observer variables that store the positions of each traffic participant at the start of a loop iteration. This allows expressing the change in positions discretely. To ensure that the RL robot stops in time, the opponent must not exceed its

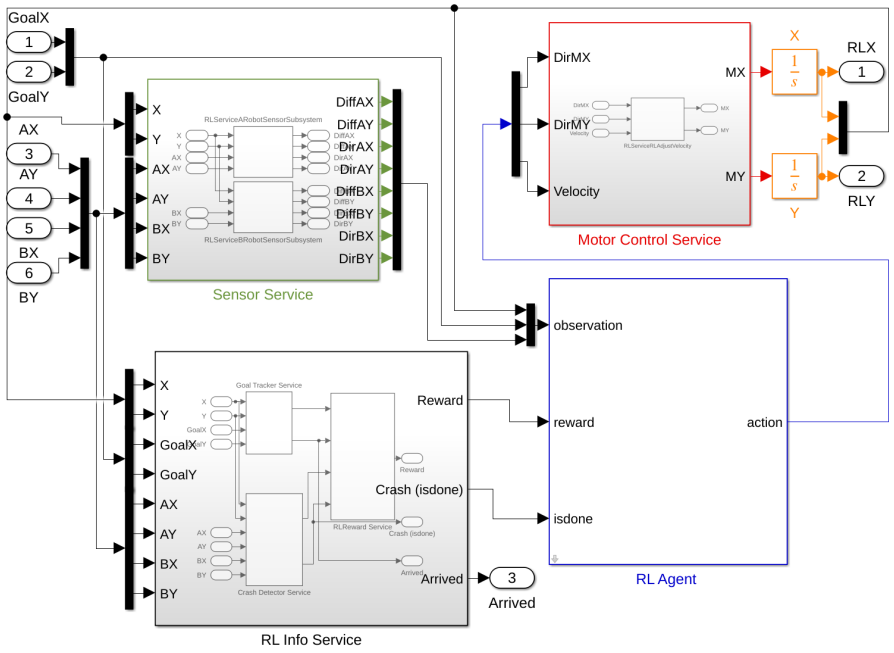


Figure 8.11: Simulink model of autonomous robot

Table 8.5: Contracts for the RL Robot and the Opponents

RL Robot	**Assumption**	$d(Pos_{old,Opp}, Pos_{Opp}) \leq v_{max,Opp} \cdot time_{\Delta}$
	Guarantee	$d(Pos_{old,Opp}, Pos_{RL}) > \theta_{Opp} \ \vee Pos_{RL} = Pos_{old,RL}$
Opponent Evasion	**Assumptions**	$(d(Pos_{old,Opp}, Pos_{RL}) > \theta_{Opp} \ \vee Pos_{RL} = Pos_{old,RL}) \wedge$ $\theta_{Opp} \geq d_{min,Opp} + v_{max,Opp} \cdot ts_{Opp} \wedge$ $d(Pos_{old,Opp}, Pos_{RL}) > d_{min,Opp}$
	Guarantee	$d(Pos_{Opp}, Pos_{RL}) > d_{min,Opp}$
Opponent Velocity	**Assumption**	$true$
	Guarantee	$d(Pos_{Opp}, Pos_{old,Opp}) \leq v_{max,Opp} \cdot time_{\Delta}$

maximum speed. To capture this, we introduce two additional observer variables: $time_{old}$, which stores the simulation time of the previous loop before the continuous evolution and $time_{\Delta}$, which stores the difference between the current and the old simulation time ($time - time_{old}$). The resulting contract is shown in the first row of Table 8.5.

Contracts for the Opponent. Similar to the RL robot, the opponent moves continuously, but takes discrete control decisions. As the opponent evades actively, we can provide a property that directly captures the collision freedom:

$$d(Pos_{old,Opp}, Pos_{RL}) > d_{min,Opp} \rightarrow d(Pos_{Opp}, Pos_{RL}) > d_{min,Opp}$$

The formula states that if the RL robot's new position is safe w.r.t. the opponent's old position, the independently chosen new position of the opponent will also be safe. However, we cannot verify successful evasion of the opponent for arbitrarily moving RL robots. To ensure that the opponent evades successfully, the RL robot has to stop before its safety threshold. To capture this, we add the guarantee of the RL robot contract to the assumptions of the opponent's contract. Furthermore, the opponent's threshold must leave enough room to detect an RL robot in time. To ensure that if a threshold is violated at least one control decision is made

Table 8.6: Interactive Verification Effort

Case Study	Interactive Verification Effort
Collision Freedom with two Opponents	approx. 20 hours
Collision Freedom with six Opponents	additional 10 minutes
Collision Freedom with Sensor Disturbances	additional 2 hours

by the opponent before it can collide, we add the assumption $\theta_{Opp} \geq d_{min,Opp} + v_{max,Opp} \cdot ts_{Opp}$. The resulting contract that describes successful evasion is shown in the middle row of Table 8.5. The RL robot contract assumes a maximum velocity for the opponent. For the RL robots contract to hold, we additionally define and verify a velocity contract for the opponent, as shown in the lower row of Table 8.5. Note that the actual contracts contain more assumptions than shown here. For example, most constants such as sampling times ts and velocities v are assumed to be non-negative.

Collision Freedom. By replacing the traffic participants with their contracts and transforming the overall model into $\mathsf{d}\mathcal{L}$ we were able to successfully verify the collision freedom property with KeYmaera X.

Modeling and verifying the case study in a service-oriented manner resulted in compact, abstract $\mathsf{d}\mathcal{L}$ models. The hybrid program of the RL robot consists of 128, the HP of the opponents consists of 79 and the top level HP of our case study consists of 114 lines of $\mathsf{d}\mathcal{L}$. As shown in the first row of Table 8.6, the time needed to verify the overall system was approximately 20 person hours, mainly allocated to verifying the hybrid contract of the opponent and finding the hybrid contract for the RL robot.

8.5.4 Collision Freedom with Additional Opponents

The required effort and expertise for interactive verification with KeYmaera X is considerable. However, one of the major advantages of our approach is that verified contracts can be reused in larger systems. Together with the deductive verification capabilities of KeYmaera X, this means that our approach scales well for a larger number of components.

To demonstrate this, we have extended the feature model of the factory to enable the selection of different numbers of opponents. Besides the initial number of two opponents, we enable the selection of six opponents. Each additional opponents adds an instance of the opponent service and its job scheduler service. Additionally, the RL robot service requires additional input signals to detect the position of these new opponents. Therefore, we create a new variant that has more input signals for positions of opponents. Additionally, the sensor service and RL info service are extended by similar variants. Figure 8.12 shows the feature model of the factory. The feature models of the new RL robot, sensor service and RL info service are analogous to the factory feature model. The motor control service, job scheduler service and opponent service only have one variant each and their feature models only consist of the root feature. Depending on the number of opponents, the according feature uses the appropriate instance of the RL robot, which in turn uses the appropriate instances of the sensor and RL info services.

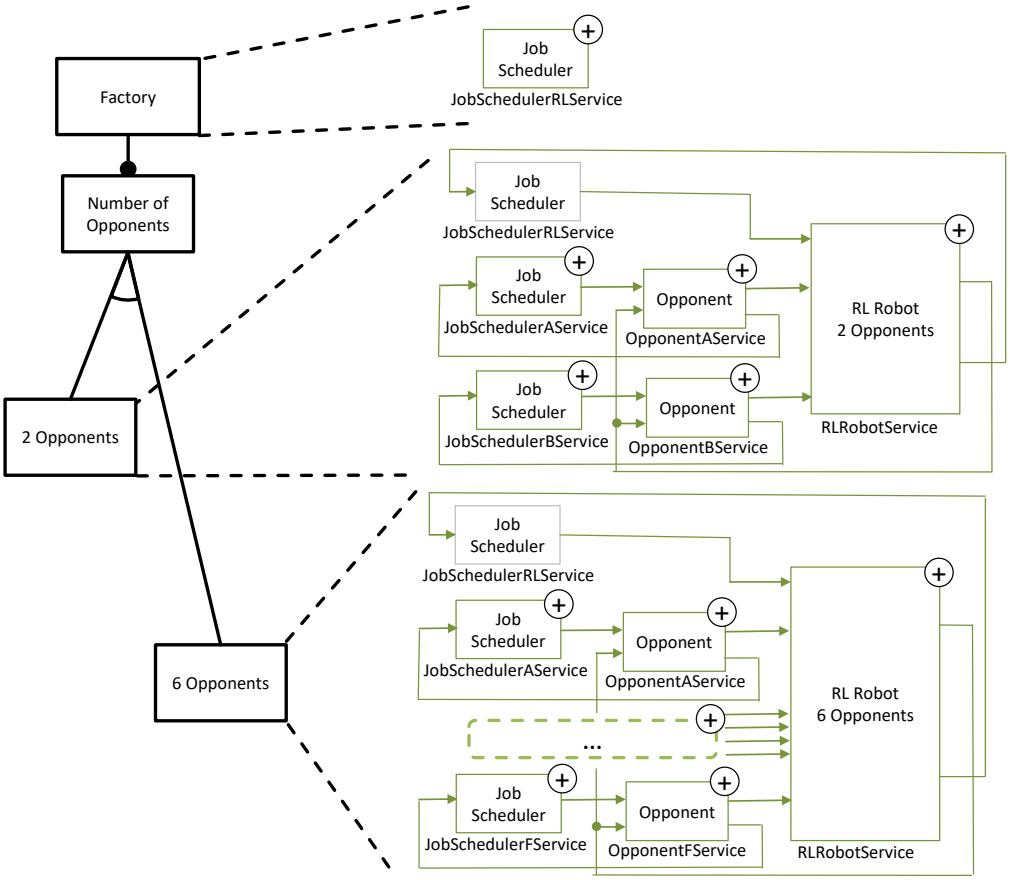


Figure 8.12: Feature Model of Factory with Different Number of Opponents

We have verified the collision freedom property for the abstract top level model with six opponents in several additional minutes, as shown in the second row of Table 8.6. Due to our service-oriented verification approach, the verified contract of the opponent can also replace all of its new occurrences, which results in no additional verification effort regarding the opponent service. Furthermore, we have verified the RL robot's contract for each opponent by applying tactical replacements to the proofs of the original case study.

8.5.5 Collision Freedom with Additional Sensor Disturbances

Capturing the safe behavior of an RL agent through a contract enables us to take changes in a model into account. Also, service-oriented modeling and verification enables us to make changes and to refine parts of a model with comparably low additional verification effort. Overall, this results in a flexible approach, which enables us to reuse contracts and verification results if the model is changed.

To demonstrate this flexibility, we have added another variant to RL robot service. In this variant, we add noise to the sensor and a maximum measurable distance. We have derived a new RL agent contract, which takes these changes into account. The resulting feature model of the factory is shown in Figure 8.13. In this feature model, the RL robot service can be replaced by one that contains a new sensor service variant that implements the noise. The new feature models for the RL robot

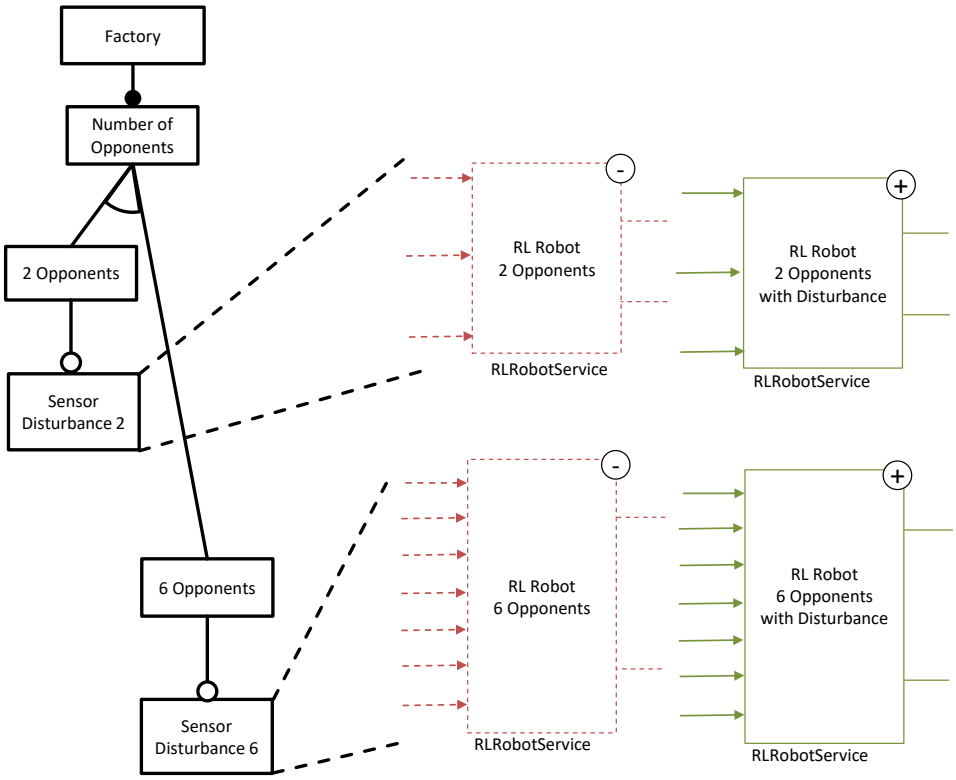


Figure 8.13: Feature Model of Factory with Optional Sensor Disturbance

service and the sensor service have the same structure as the factory feature model. The RL robots sensor service consists of multiple analogous subservices (*Sensor Subsystem*), which calculate the outputs for different opponents. To introduce disturbance, we multiply the relative positions on both axes ($DiffX, DiffY$) with a noise factor n_{Opp}, which is generated from a uniform distribution. For non-negative noise values, this stretches or shrinks the perceived distance to the opponent. We additionally limit the sensors range. If the disturbed distance exceeds a maximum distance, d_{Opp} is set to d_{max}.

We have defined a contract that captures the sensors modified behavior (see upper row of Table 8.7). However, for $n_{Opp} > 1$, the perceived distance can exceed the actual distance. This allows the RL agent to choose unsafe velocities. Given that the maximum possible noise factor is known, we have defined the modified RL agent contract shown in the lower row of Table 8.7, which computes the distance received from the sensors relative to the maximum noise factor n_{max}. This enables us to disregard the maximum distance d_{max}.

8.5.6 Simulation Experiments

Our approach enables us to model and verify safe intelligent hybrid systems. Furthermore, with the controller monitor, we can enforce safe behavior of an RL agent at simulation time. To illustrate that collisions are effectively prevented by

Table 8.7: Modified Contracts with Noise

Modified Sensor	**Assumption**	$n_{min} \geq 0 \wedge n_{max} \geq n_{min} \wedge d_{max} \geq 0$
	Guarantee	$n_{Opp} \geq n_{min} \wedge n_{max} \geq n_{Opp} \wedge$ $(d(Pos_{Opp}, Pos_{RL}) \cdot n_{Opp} \leq d_{max}$ $\rightarrow d_{Opp} = d(Pos_{Opp}, Pos_{RL}) \cdot n_{Opp}) \wedge$ $(d(Pos_{Opp}, Pos_{RL}) \cdot n_{Opp} > d_{max} \rightarrow d_{Opp} = d_{max})$
Modified RL Agent	**Assumption**	$n_{min} \geq 0 \wedge n_{max} \geq n_{min} \wedge n_{max} \geq 1 \wedge d_{max} \geq 0$
	Guarantee	$d_{Opp}/n_{max} - (v_{RL} + v_{max,Opp}) \cdot ts_{RL} > \theta_{Opp}$

our safety concept, we compare our monitored safe RL agent with a default RL agent. The default RL agent uses the same learning algorithm but does not implement a monitor to enforce safe behavior. In a second experiment, we demonstrate that our safety concept prevents collisions in dangerous scenarios even without a sophisticated RL algorithm. To this end, we compare a random agent that uses a safety monitor with a random agent that does not use such precautions in a totally chaotic environment.

Simulating Learning Agents in the Factory

To demonstrate that our safe RL agent effectively learns how to navigate the factory environment while acting safely, we have compared it with an unsafe default RL agent. Both agents use an approximate Q-learning algorithm [WD92] with a custom approximation function. The function combines manually defined features, which describe implications of actions in certain states, for example, whether the application of an action in the current state is likely to decrease the distance to the goal or whether it will likely decrease or increase the distance to the closest opponent. Approximate RL approaches generalize from observed behavior, leading to a faster and more adaptable learning behavior, which is useful for navigation and evasion.

Our test scenario consists of four workstation and a road leading vertically through the factory. Two opponents (Opp) move up and down separate lanes of the road. The RL robot (Rob) is tasked with reaching the different workstations (W). The order of the goals is chosen randomly by the RL robots job scheduler. Road lanes also feature intermediate goals, which can lead to the opponents changing direction unexpectedly. The RL robot starts at one of the four workstation and may choose from a range of velocities allowing moving slower and faster than the opponents. The RL agent receives positive rewards for decreasing the distance to a workstation and for reaching a workstation. It is punished for colliding with an opponent. To prevent that the RL robot constantly interferes with the opponents' paths, it is also punished for getting into an opponent's safety threshold, even if no collision occurs.

We have trained each agent for 250 episodes with a maximum of 120 steps each. A step represents one sampling time interval of the RL agent. If the RL robot crashes, the current episode ends early. After training, we have simulated both agents again for 100000 steps. Table 8.8 shows the results. The default agent crashed during the final simulation after 49689 of the targeted 100000 time steps. The fact that it

Table 8.8: Comparison of the Safe RL Agent and the Default RL Agent

		Default RL Agent	**Safe RL Agent**
Training	Episodes that ended in a crash	0/250=0%	0/250=0%
Final Simulation	Crash	after 49689 steps	no crash
	Times below threshold	8	25
	Goals reached	2945	4960

did not crash during training can be explained by the fact that it is punished for getting into an opponent's threshold. Because of this it learns that maintaining a certain distance to opponents is desirable, even without being supported by a monitor. The safe RL agent never crashed neither in training nor in simulation and thus acted safely. The fact that it reached 4960 goals in 100000 simulation steps while only getting below an opponent's threshold 25 times indicates that it learns to evade the opponents actively and avoids situations where the contract forces it to stop. We validated this by graphically visualizing the traffic participants movements. However, although the safe agent was simulated for more than twice as long, it only reached 68.42% more goals. This indicates that the contract impedes its performance.

Simulating Random Agents in the Factory

To illustrate that our safety concept prevents collisions in dangerous scenarios even without a sophisticated RL algorithm, we consider a worst-case scenario in which an agent acts randomly in a chaotic environment. The default random agent can choose any action in any state. The safe random agent chooses randomly from the filtered set of actions allowed by its contract. In this way, agents cannot make use of obtained knowledge to avoid collisions.

To model a more chaotic environment, each opponent gets a random safe start position, and two random goals in a square of size 10×10 around the safe random robot's initial position. These change in each training episode. Other settings are equal to the previous experiment.

As an additional test, we also simulate the extended model from the scalability experiment (Subsection 8.5.4) with six opponents.

We have simulated the models for both the default random agent as well as the safe random agent for 1000 episodes with a maximum of 120 steps taken in each. The results are shown in Table 8.9. The safe random agent never crashed in any of the experiments neither with 2 nor 6 opponents. The default random agent crashed in a majority of episodes (in 96.5% for the upscaled version of the model). This demonstrates that our safety concept is a decisive factor in ensuring collision freedom for the overall system.

Table 8.9: Crashes of a Default and a Safe Random Agent

	Default Random Agent	**Safe Random Agent**
2 Opponents	753/1000=75.3%	0/1000=0%
6 Opponents	965/1000=96.5%	0/1000=0%

8.6 Summary

In this chapter, we have demonstrated the practical applicability of our approach by using four different hybrid system case studies to evaluate different aspects of our approach. We have shown the benefits of using hybrid contracts to define and verify crucial properties of services. For the distance warner, we have used our compositional verification to ensure safety of the system, which was not possible in a monolithic approach. While for some systems a monolithic verification is possible, we have reduced the verification time and effort by first creating and verifying hybrid contracts for the components in the systems that we designed as Simulink services. The verification times of the service and abstracted system were in general shorter than the verification time of the concrete system. A further major benefit of our approach is the reusability of our verification results. For all future systems that use services that are already verified, we can reuse the already verified hybrid contracts in the abstract system verification. Furthermore, we showed that our transformation and automatic invariant generation has a huge impact on the verification effort by reducing the manual effort. By defining hybrid contracts for an autonomous robot and using runtime monitors, we were able to enforce safe behavior of an intelligent robot that uses reinforcement learning at runtime. This can be extended to the design process of new systems. If the system verification requires that a service provides a given hybrid contract, a developer can create the service in such a manner that it fulfills the required hybrid contract. This still requires the transformation of the service and the verification that it fulfills the hybrid contract.

9 Conclusion

In this thesis, we have presented an approach for the service-oriented design and deductive formal verification of hybrid control systems modeled in Simulink. Our approach combines the strengths of Simulink for model-driven development of hybrid systems with the power of formally ensuring the correct behavior of modeled systems under all circumstances. In contrast to most existing work on the formal verification of Simulink models, we are not restricted to purely discrete models. Instead, we also support hybrid system models where discrete and continuous behavior are combined. This enables us to model and ensure the correct behavior of cyber-physical systems, which have a strong connection to the physical environment. Our approach enables the formal verification of hybrid systems modeled in an industrially widely used informal language. With our service-oriented approach together with deductive verification, we achieve a comparatively high level of scalability, enable the reusability of verification results, and provide an integration of our verification process with a model-driven design process.

9.1 Results

Our main contributions are:

1. We have defined an automatic transformation for hybrid control systems modeled in Simulink into the formal semantics of $\mathsf{d}\mathcal{L}$. This formalization is the foundation for our approach for the formal verification of hybrid Simulink models. Our automatic transformation enables a designer to use the well-developed industrial tool suite Simulink to create the model of a hybrid control system and without additional effort generate a formal model that can be used for verification. To ease the verification of Simulink models, we have presented *extension functions*, which enable the designer to systematically insert observer variables into a given hybrid system, and *property templates*, which can be used as design patterns to support the designer in defining properties based on commonly used properties, i.e., range, timing, and dynamic properties.

2. We have introduced *hybrid contracts*, which abstractly capture the dynamic behavior of Simulink subsystems that are encapsulated as services. Verified properties of Simulink services are described by hybrid contracts. These hybrid contracts formally capture the interface behavior of services and enable to abstract the concrete behavior of services in larger systems. The key idea of our compositional verification is that we use our hybrid contracts to replace services in a given hybrid system by their contracts, and thus enable the hierarchical abstraction from implementation details of services. To ensure the correct embedding of services into concrete systems, we have provided a proof sketch for the soundness of this abstraction technique.

3. We have defined customizable services that can be used in Simulink designs. Services facilitate the reuse of predefined systems and enable the reuse of existing verification results for compositional system verification. Our definition of services in Simulink has two major advantages: First, we provide

means to verify the correct behavior of the system. Second, we also provide means to explicitly and precisely capture the dynamic interface of a newly developed component. A Simulink service fulfills a concrete functionality in a system. The service comprises a Simulink model, a feature model that enables changes in the system, and hybrid contracts that formally capture its interface behavior.

The key idea of the transformation is threefold: First, we map the informally defined Simulink semantics to the formally well-defined semantics of differential dynamic logic (dℒ). Second, we use an expressive macro mechanism to efficiently capture stateless behavior and arithmetic or logic expressions. Third, we precisely capture discrete as well as continuous behavior in a nondeterministic repetitive simulation loop that combines discrete assignments and continuous evolutions. Our transformation approach supports a broad class of hybrid systems, i.e., it supports time-discrete, time-continuous and control flow blocks and takes their timing and interactions into account. To cope with discrete jumps in continuous behavior, we introduce a small time step behavior to model a maximum delay between the change of a value and the next step in which the control flow is updated. The dℒ representation of the model contains a nondeterministic repetition that contains a precise description of the behavior of the Simulink blocks in each simulation step. As a result, our dℒ model captures all possible behavior is captured and not only a single simulation run.

We have demonstrated the applicability of our approach with experimental results. We have presented a fully-automatic transformation from Simulink into dℒ, and we have shown how the resulting dℒ models can semi-automatically be verified with the interactive theorem prover KeYmaera X. With the use of KeYmaera X, we are able to prove safety and correctness properties of hybrid control systems deductively for all possible input scenarios. In particular, we have used an automotive industrial case study of a hybrid distance warner, which could neither be verified using the Simulink Design Verifier nor KeYmaera X on the concrete system. Our service-oriented design and compositional verification allows proving safety properties of systems for which a naive flattening approach does not scale well. We have also shown the applicability of our approach with an intelligent, autonomous robot in a factory setting. While the full verification of learning components is not possible in general, our hybrid contracts provide a suitable abstraction to capture their behavior formally. We have combined this with runtime monitoring to ensure the compliance of the learning component to its hybrid contract at runtime.

9.2 Discussion

In the introduction, we have required that our approach to formally verify hybrid control systems should fulfill the following criteria:

1. consider hybrid behavior
2. provide reuse capabilities
3. provide a formal foundation for hybrid Simulink models

4. enable compositional verification
5. reduce manual effort by providing automation
6. is evaluated with different case studies

In the following, we review our service-oriented design and verification approach using these criteria.

Hybrid behavior. Our service-oriented design and verification approach supports Simulink models that contain discrete as well as continuous blocks. Our approach currently supports Simulink models that contain a representative set of discrete and continuous blocks. Furthermore, we can describe the hybrid behavior of services with our hybrid contracts. $\mathsf{d}\mathcal{L}$ is excellently well suited to capture hybrid system behavior as well as safety properties of dynamic systems. With different case studies, we have shown that we can verify safety properties for hybrid Simulink models with acceptable effort.

Reuse capabilities and variability. Our service-oriented design allows reuse of customizable services and, furthermore, the variability by the feature model enables a wider application of services. Additionally, our hybrid contracts enable us to reuse verification results by replacing the inner structure of services by their abstract behavior defined by hybrid contracts.

The use of services in new systems can require the creation of new hybrid contracts that are then used in the verification of new system properties. The verification of new hybrid contracts for existing services can be facilitated by the reuse of invariants that were used in the creation of existing hybrid contracts.

Formal Foundation. With our transformation from Simulink to $\mathsf{d}\mathcal{L}$, we define a formal semantics for Simulink in differential dynamic logic. Note that there is a semantic gap between the simulation semantics of Simulink and the semantics of our $\mathsf{d}\mathcal{L}$ models. Thus, it is necessary to trust in the correctness of our transformation. However, as the Simulink semantics is not formally defined in [Mata], this semantic gap cannot be avoided.

By choosing $\mathsf{d}\mathcal{L}$ to formalize the Simulink semantics, we gain access to the mature and powerful verification tool KeYmaera X. At the same time, by precisely capturing the semantics of each block separately, we stay as close as possible at the original Simulink semantics, and thus keep the semantic gap small. We take the liberty to abstract from numerical errors quite roughly by allowing evaluations with an ϵ-delay, but we are confident that this is a good choice as it preserves the most important aspect of numerical errors (that a deviation may occur) while keeping the formal model manageable and comprehensible. A hybrid contract should capture enough behavior of a service to be useful in the verification of systems it is used in and at the same time should be verifiable in KeYmaera X. By defining more concrete behavior in a hybrid contract, it can be more useful for the verification of the overall system, but its own verification often gets more challenging.

The size of the target model grows linear in the number of data-flow blocks and exponentially with the amount of interacting control flow blocks. This is the case since the different control flows cause different behavior for the continuous evolutions in the system. However, we have introduced an optimization that removes control flow branches from the dℒ model that cannot be reached by the given Simulink model.

Compositionality. By using our compositional verification, we are able to verify safety properties for larger systems that consists of multiple interacting services. Our transformation can result in model sizes that impede or even prevent the interactive verification with KeYmaera X. This can be solved by encapsulating control flow blocks into different services and therefore enable the full capabilities of our service-oriented design and the resulting compositional verification. Services can also be verified compositionally. With our hybrid contracts, we can verify properties of larger systems by encapsulating system functionalities into services. During the transformation, each service can be replaced by its hybrid contracts. This enables us to verify systems that could not be verified in a monolithic transformation approach.

Automation. Our automatic transformation of Simulink into dℒ enables us to easily apply our approach to different models and spares the designer the tedious task of providing a formal model themself. Currently, our transformation framework only supports a representative set of Simulink blocks of different groups. The creation of hybrid contracts and their verification is done manually. One main challenge in the verification is to find appropriate invariants. With our automatic invariant generation, we are able to reduce the manual effort during the verification. However, the required expertise for contract design and interactive verification is still considerable. Good knowledge of the domain of the model and the expertise in the usage of the interactive theorem prover KeYmaera X can reduce the effort that is necessary to create useful hybrid contracts for services. The aim of our service-oriented modeling approach is that these verification experts create the hybrid contracts and enrich the services in the service library that can be used by designers in their system creation.

Practical Applicability. We have successfully applied our service-oriented design and verification approach to multiple case studies. With a temperature control system and integrated environment model, we have evaluated our transformation and the compositional verification. We used the model of a generic infusion pump to evaluate the automatic invariant generation. We also considered larger systems, with the industrial model of a multi-object distance warner, which was provided by an industrial partner. This model showed that we are able to efficiently verify with our compositional approach. Lastly, we used the model of an autonomous factory robot to show the powerfulness our hybrid contracts. We have verified that no collisions occur in the whole system and we were able to ensure safe behavior of a robot that uses reinforcement learning by enforcing its hybrid contracts via runtime monitoring.

9.3 Future Work

Our service-oriented design and verification approach enables the verification of hybrid control systems that are modeled in Simulink and it provides opportunities for further research.

Extended Formalization

We currently use small time steps to model zero crossing detection and small delays in the change of control states of a system. Due to the calculations of the Simulink solver, it is possible that numerical errors change the desired system behavior. Further research could extend the transformation more precisely capture these numerical errors. This could be used to perform error estimations during the proof and determine error margins for which the system is safe.

A rather basic extension is the addition of further blocks in the supported block set. With our macro replacement mechanism and our Factory Pattern Design, we provide the foundation for the extension of the block transformations. Furthermore, blocks with more sophisticated behavior, e.g., S-Function blocks, could also be considered as services. The complexity of these blocks makes them impractical for our transformation to $\mathsf{d}\mathcal{L}$ via transformation rules. However, if we can create and verify that these blocks fulfill hybrid contracts via other transformation and verification tools, we can reuse these results in the verification of systems that contain such blocks. The service library is currently limited to some services. More services with verified hybrid contracts would increase the overall applicability of our approach.

In this thesis, we have exemplified the application of service-oriented modeling with the modeling language Simulink. This enables us to show the applicability of our approach. However, this is only one modeling language for hybrid systems. Further research could extend our service-oriented design and verification for other modeling languages, e.g., SystemC-AMS [Acc]. The main challenge is to provide a formalization of the chosen modeling language into $\mathsf{d}\mathcal{L}$ and to provide an abstraction mechanism for services. This would not only enable service-oriented design in this new modeling language but also allow to combine models of the new modeling language and all modeling languages for that our approach is already implemented (at the moment Simulink). A service could be modeled in one language and a transformation into $\mathsf{d}\mathcal{L}$ (or even a different verification back end) could be used to proof that it fulfills some hybrid contracts. Similar to the *Reinforcement Agent* block in our factory robot case study, we can insert this service as black box into a new system. Their inner structure is unknown for the transformation and they still could be transformed, since we use their hybrid contracts to describe their behavior.

Increase Automation

Currently, we manually create the hybrid contracts for the variants of a customizable service. This also requires us to perform proofs for all individual services, whenever we change or add new hybrid contracts. Differential refinement logic [LP16] could provide a formal foundation to increase the automation of the verifi-

cation of variants. By defining abstract models for a feature model, we can reduce the effort that is necessary to perform proofs for variants of a customizable service.

While the Simulink Design Verifier only supports discrete systems, hybrid systems can contain components that only contain discrete behavior. Our approach could be extended to use different verification back ends for different components depending on their behavior. Additionally, our hybrid contracts can also be used to capture discrete behavior. Therefore, it could be possible to use the Simulink Design Verifier to ensure that discrete services fulfill their hybrid contracts. These results can still be used in the verification of the system that contains hybrid behavior. Even other verification back ends could be considered.

Support for Intelligent Systems

With a case study of an autonomous factory robot, we have presented first results for safe intelligent systems with hybrid contracts. This is an important current research topic since cyber-physical systems are often used in dynamic environments. Further investigation how we can use hybrid contracts to ensure that learning components always chose safe action is an interesting research topic.

List of Figures

Listings

List of Tables

Bibliography

[Acc] Accellera Systems Initiative. *SystemC AMS (https://accellera.org/community/systemc/about-systemc-ams)*.

[AD94] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[AER14] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. "Formal verification of control systems' properties with theorem proving". In: *UKACC International Conference on Control (CONTROL)*. IEEE. 2014, pp. 244–249.

[Ala+12] Manar H Alalfi, James R Cordy, Thomas R Dean, Matthew Stephan, and Andrew Stevenson. "Models are code too: Near-miss clone detection for Simulink models". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 295–304.

[Ala+14] Manar H Alalfi, Eric J Rapos, Andrew Stevenson, Matthew Stephan, Thomas R Dean, and James R Cordy. "Semi-automatic identification and representation of subsystem variability in simulink models". In: *Int. Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2014, pp. 486–490.

[Alu+93] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems". In: *Hybrid systems*. Springer, 1993, pp. 209–229.

[And19] Étienne André. "What's decidable about parametric timed automata?" In: *International Journal on Software Tools for Technology Transfer* 21.2 (2019), pp. 203–219.

[ASH11] Bakr Al-Batran, Bernhard Schätz, and Benjamin Hummel. "Semantic clone detection for model-based development of embedded systems". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 258–272.

[Bal+15] Paolo Ballarini, Benoit Barbot, Marie Duflot, Serge Haddad, and Nihal Pekergin. "HASL: A new approach for performance evaluation and model checking from concepts to experimentation". In: *Performance Evaluation* 90 (2015), pp. 53–77.

[Bar+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. "Boogie: A modular reusable verifier for object-oriented programs". In: *Int. Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[Bar+18] Benoit Barbot, Béatrice Bérard, Yann Duplouy, and Serge Haddad. "Integrating Simulink models into the model checker cosmos". In: *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer. 2018, pp. 363–373.

[BC12] Olivier Bouissou and Alexandre Chapoutot. "An operational semantics for Simulink's simulation engine". In: *ACM SIGPLAN Notices* 47.5 (2012), pp. 129–138.

[BCM16] Patricia Bouyer, Maximilien Colange, and Nicolas Markey. "Symbolic optimal reachability in weighted timed automata". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 513–530.

[Ben+14] Luca Benvenuti, Davide Bresolin, Pieter Collins, Alberto Ferrari, Luca Geretti, and Tiziano Villa. "Assume–guarantee verification of nonlinear hybrid systems with Ariadne". In: *Int. Journal of Robust and Nonlinear Control* 24.4 (2014), pp. 699–724.

[Ben+95] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL—a tool suite for automatic verification of real-time systems". In: *International hybrid systems workshop*. Springer. 1995, pp. 232–243.

[Ber+18] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. "Verifying auto-generated C code from Simulink". In: *Int. Symposium on Formal Methods*. Springer. 2018, pp. 312–328.

[BG11] Radhakisan Baheti and Helen Gill. "Cyber-physical systems". In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[Bog+14] Sergiy Bogomolov, Goran Frehse, Marius Greitschus, Radu Grosu, Corina Pasareanu, Andreas Podelski, and Thomas Strump. "Assume-guarantee abstraction refinement meets hybrid systems". In: *Haifa verification conference*. Springer. 2014, pp. 116–131.

[Bos11] Pontus Boström. "Contract-based Verification of Simulink Models". In: *Int. Conference on Formal Engineering Methods*. Springer. 2011, pp. 291–306.

[Bou+17] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "A Synchronous Look at the Simulink Standard Library". In: *ACM Transactions on Embedded Computing Systems (TECS)*. Vol. 16(5s). ACM. 2017, p. 176.

[Bul+12] Peter Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. "Checking and distributing statistical model checking". In: *NASA Formal Methods Symposium*. Springer. 2012, pp. 449–463.

[Che+17] Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. "MARS: A toolchain for modelling, analysis and verification of hybrid systems". In: *Provably Correct Systems*. Springer, 2017, pp. 39–58.

[CK03] Alongkrit Chutinan and Bruce H Krogh. "Computational techniques for hybrid system verification". In: *IEEE transactions on automatic control*. Vol. 48(1). IEEE. 2003, pp. 64–75.

[CK99] Alongkrit Chutinan and Bruce H Krogh. "Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations". In: *Int. workshop on hybrid systems: computation and control*. Springer. 1999, pp. 76–90.

[Cla+99] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 1999.

[CQS] CQSE. *ConQAT (https://www.cqse.eu/en/news/blog/conqat-end-of-life/)*.

[Dav+15] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. "Uppaal SMC tutorial". In: *International Journal on Software Tools for Technology Transfer* 17.4 (2015), pp. 397–415.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Dei+10] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. "Model clone detection in practice". In: *Proceedings of the 4th International Workshop on Software Clones*. ACM. 2010, pp. 57–64.

[Eri+17] Andreas Berre Eriksen, Chao Huang, Jan Kildebogaard, Harry Lahrmann, Kim G Larsen, Marco Muniz, and Jakob Haahr Taankvist. "Uppaal stratego for intelligent traffic lights". In: *12th ITS European Congress*. 2017.

[FHK04] Goran Frehse, Zhi Han, and Bruce Krogh. "Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction". In: *2004 43rd IEEE Conference on Decision and Control (CDC)(IEEE Cat. No. 04CH37601)*. Vol. 1. IEEE. 2004, pp. 479–484.

[Fil+16] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. "Simulink to UPPAAL statistical model checker: Analyzing automotive industrial systems". In: *Int. Symposium on Formal Methods*. Springer. 2016, pp. 748–756.

[FP13] Jean-Christophe Filliatre and Andrei Paskevich. "Why3 – where programs meet provers". In: *European Symposium on Programming*. Springer. 2013, pp. 125–128.

[FP18] Nathan Fulton and André Platzer. "Safe reinforcement learning via formal methods: Toward safe control through proof and learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[Fre+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. "SpaceEx: Scalable verification of hybrid systems". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 379–395.

[Fre05] Goran Frehse. "PHAVer: Algorithmic verification of hybrid systems past HyTech". In: *Int. workshop on hybrid systems: computation and control*. Springer. 2005, pp. 258–273.

[Ful+15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems". In: *Int. Conference on Automated Deduction*. Springer. 2015, pp. 527–538.

[GIP] GIP. *Generic Infusion Pump Research Project*. https://rtg.cis.upenn.edu/gip/. Accessed: 2020-05-18.

[Gro+93] Robert L Grossman, Anil Nerode, Anders P Ravn, and Hans Rischel. *Hybrid systems*. Vol. 736. Springer, 1993.

[Hab+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. "First-class variability modeling in Matlab/Simulink". In: *Int. Workshop on Variability Modelling of Software-intensive Systems*. ACM. 2013, p. 4.

[Hen+98] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. "What's decidable about hybrid automata?" In: *Journal of computer and system sciences* 57.1 (1998), pp. 94–124.

[Hen00] Thomas A Henzinger. "The theory of hybrid automata". In: *Verification of digital and hybrid systems*. Springer, 2000, pp. 265–292.

[HHW95] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. "HyTech: the next generation". In: *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*. IEEE. 1995, pp. 56–65.

[Hoa69] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[HPR94] Nicolas Halbwachs, Yann-Eric Proy, and Pascal Raymond. "Verification of linear hybrid systems by means of convex approximations". In: *International Static Analysis Symposium*. Springer. 1994, pp. 223–237.

[HRB13] Paula Herber, Robert Reicherdt, and Patrick Bittner. "Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving". In: *Int. Conference on Embedded Software (EMSOFT)*. IEEE. 2013, pp. 1–10.

[Kan+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[LP16] Sarah M Loos and André Platzer. "Differential refinement logic". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 2016, pp. 505–514.

[LS04] Shuvendu K Lahiri and Sanjit A Seshia. "The UCLID decision procedure". In: *Int. Conference on Computer Aided Verification*. Springer. 2004, pp. 475–478.

[Lyn+95] Nancy Lynch, Roberto Segala, Frits Vaandrager, and Henri B Weinberg. "Hybrid i/o automata". In: *International Hybrid Systems Workshop*. Springer. 1995, pp. 496–510.

[Mata] MathWorks. *MATLAB Simulink (www.mathworks.com/help/simulink/referencelist.html)*.

[Matb] MathWorks. *MATLAB Simulink (www.mathworks.com/products/simulink.html)*.

[Mat08] MathWorks. *White Paper: Code Verification and Run-Time Error Detection Through Abstract Interpretation*. Tech. rep. 2008.

[Mey92] Bertrand Meyer. "Applying'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[MF16] Stefano Minopoli and Goran Frehse. "SL2SX translator: from Simulink to SpaceEx models". In: *19th Int. Conf. on Hybrid Systems: Computation and Control*. ACM. 2016, pp. 93–98.

[MP17] Stefan Mitsch and André Platzer. "The KeYmaera X proof IDE: Concepts on usability in hybrid systems theorem proving". In: *3rd Workshop on Formal Integrated Development Environment*. Vol. 240. Theoretical Computer Science. Open Publishing Association. 2017, pp. 67–81.

[Mül+16] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. "A component-based approach to hybrid systems safety verification". In: *International Conference on Integrated Formal Methods*. Springer. 2016, pp. 441–456.

[Mül+17] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. "Change and delay contracts for hybrid system component verification". In: *Int. Conference on Fundamental Approaches to Software Engineering*. Springer. 2017, pp. 134–151.

[Mül17] Andreas Müller. "Component-based Deductive Verification of Cyber-Physical Systems/submitted by Andreas Müller". PhD thesis. Universität Linz, 2017.

[Mur+13] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. "Compositional verification of a medical device system". In: *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*. 2013, pp. 51–64.

[OAS] OASIS Committee Specification. *Reference Architecture Foundation for Service Oriented Architecture (http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html)*. Version 1.0.

[OHa13] Colin O'Halloran. "Automated verification of code automatically generated from Simulink®". In: *Automated Software Engineering* 20.2 (2013), pp. 237–264.

[Pha+09] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. "Complete and accurate clone detection in graph-based models". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 276–286.

[Pla08] André Platzer. "Differential dynamic logic for hybrid systems". In: *Journal of Automated Reasoning* 41.2 (2008), pp. 143–189.

[Pla17] André Platzer. "A complete uniform substitution calculus for differential dynamic logic". In: *Journal of Automated Reasoning* 59.2 (2017), pp. 219–265.

[PV06] Michael P Papazoglou and Willem-Jan Van Den Heuvel. "Service-oriented design and development methodology". In: *International Journal of Web Engineering and Technology* 2.4 (2006), pp. 412–442.

[RG14] Robert Reicherdt and Sabine Glesner. "Formal verification of discrete-time MATLAB/Simulink models using Boogie". In: *Int. Conference on Software Engineering and Formal Methods*. Springer. 2014, pp. 190–204.

[Roe+16] Hendrik Roehm, Jens Oehlerking, Matthias Woehrle, and Matthias Althoff. "Reachset conformance testing of hybrid automata". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. 2016, pp. 277–286.

[RPV16] Nima Roohi, Pavithra Prabhakar, and Mahesh Viswanathan. "Hybridization based CEGAR for hybrid automata with affine dynamics". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016, pp. 752–769.

[RPV17] Nima Roohi, Pavithra Prabhakar, and Mahesh Viswanathan. "HARE: A hybrid abstraction refinement engine for verifying non-linear hybrid automata". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 573–588.

[RS11] Pritam Roy and Natarajan Shankar. "SimCheck: a contract type system for Simulink". In: *Innovations in Systems and Software Engineering* 7.2 (2011), pp. 73–83.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SCN13] Ricardo Sanfelice, David Copp, and Pablo Nanez. "A toolbox for simulation of hybrid systems in Matlab/Simulink: Hybrid Equations (HyEQ) Toolbox". In: *Int. Conference on Hybrid Systems: Computation and Control*. ACM. 2013, pp. 101–106.

[Sel03] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25.

[WD92] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[Wol] Wolfram Alpha LLC. *Wolfram|Alpha (https://www.wolframalpha.com/)*.

[Zou+15] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. "Formal verification of simulink/stateflow diagrams". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2015, pp. 464–481.

Supervised Bachelor and Master Theses

[Won18] Philipp Wonschik. “Tranformation zusammenhängender Blockstrukturen aus Simulink in Differential Dynamic Logic”. Bachelor Thesis. Germany: Technische Universität Berlin, 2018.

[Sch19] Sascha Schweitzer. “Analyse von Simulink Modellen zur Reduktion des Verifikationsaufwands”. Bachelor Thesis. Germany: Technische Universität Berlin, 2019.

[Che20] Robin Fa-Yan Chen. “Service-orientierte Modellierung einer generischen Infusionspumpe”. Bachelor Thesis. Germany: Technische Universität Berlin, 2020.

[Umo20] Roland Umoru. “Extended Feature Modeling in a Service-Oriented Approach for Simulink”. Bachelor Thesis. Germany: Technische Universität Berlin, 2020.

[Ade20] Julius Adelt. “Verifying intelligent cyber-physical systems with the interactive theorem prover KeYmaera X”. Master Thesis. Germany: Westfälische Wilhelms-Universität Münster, 2020.

Publications by Timm Liebrenz

[Lie+17] **Timm Liebrenz**, Paula Herber, Thomas Göthel, and Sabine Glesner. "Towards Service-Oriented Design of Hybrid Systems Modeled in Simulink". In: *Computer Software and Applications Conf. (COMPSAC), 2017 IEEE 41st Annual*. Vol. 2. IEEE, 2017, pp. 469–474.

[Lie18] **Timm Liebrenz**. "Service-Oriented Design and Verification of Hybrid Control Systems". In: *Int. Conference on Formal Engineering Methods*. Springer. 2018, pp. 427–431.

[LHG18] **Timm Liebrenz**, Paula Herber, and Sabine Glesner. "Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X". In: *Int. Conference on Formal Engineering Methods*. Springer. 2018, pp. 89–105.

[LHG19] **Timm Liebrenz**, Paula Herber, and Sabine Glesner. "A Service-oriented Approach for Decomposing and Verifying Hybrid System Models". In: *Int. Conference on Formal Aspects of Component Software*. Springer. 2019.

[LHG20] **Timm Liebrenz**, Paula Herber, and Sabine Glesner. "Towards automated service-oriented verification of embedded control software modeled in Simulink". In: *Int. Symposium on Leveraging Applications of Formal Methods*. Springer. 2020, pp. 307–325.

[LHG21] **Timm Liebrenz**, Paula Herber, and Sabine Glesner. "Service-oriented decomposition and verification of hybrid system models using feature models and contracts". In: *Science of Computer Programming* 211 (2021), p. 102694. ISSN: 0167-6423.

[HAL21] Paula Herber, Julius Adelt, and **Timm Liebrenz**. "Formal Verification of Intelligent Cyber-Physical Systems with the Interactive Theorem Prover KeYmaera X." In: *Software Engineering (Satellite Events)*. 2021.

[HLA21] Paula Herber, **Timm Liebrenz**, and Julius Adelt. "Combine Forces: How to Formally Verify Informally Defined Embedded Systems". In: *Formal Methods*. Springer. 2021.

[ALH21] Julius Adelt, **Timm Liebrenz**, and Paula Herber. "Formal Verification of Intelligent Hybrid Systems that are modeled with Simulink and the Reinforcement Learning Toolbox (keynote paper)". In: *Formal Methods*. Springer. 2021.

,