Technische Universität München
TUM School of Engineering and Design

smartIflow - A Model Based Approach for Safety Analysis in the Early Product Life Cycle Stages

Philipp Hönig

Vollständiger Abdruck der von der TUM School of Engineering and Design der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing. Johannes Fottner
Prüfer der Dissertation:	1. Prof. Dr. Florian Holzapfel
	2. Prof. Dr. Rüdiger Lunde

Die Dissertation wurde am 02.05.2024 bei der Technische Universität München eingereicht und durch die TUM School of Engineering and Design am 30.09.2024 angenommen.

Acknowledgments

Writing this thesis was a real challenge for me. At this point, I would like to thank all the people who supported me during this period. Without the help and support of these people, I would have never been able to complete this work.

First of all, I would like to thank my mentor Prof. Dr. Rüdiger Lunde for his enthusiasm for the project, for his great support, and his patience during the entire time. His expertise in the field of model-based diagnoses contributed greatly to the successful completion of this work. The meetings we had were always very inspiring and motivating. Many thanks for the great time!

I would also like to thank my supervisor Prof. Dr. Florian Holzapfel for giving me the opportunity to work on this topic. Despite his busy schedule, he always took the time to listen to my presentations about the current status of my work. The meetings and conversations I had with him were highly motivating and this feedback helped immensely. I am very grateful that he has initiated the project together with Prof. Dr. Wolfgang Berns. I also thank the team and partners of the SysRO project for the fruitful collaboration. I would also like to thank Prof. Johannes Fottner for chairing the examination committee.

Furthermore, I would like to thank my colleague Christian Müller for his support. I especially liked the conversations with him about new concepts, technologies, or exotic programming languages. I also thank the staff of the Institute for Computer Science at the Ulm University of Applied Sciences for their support. I also thank Julian Grotz and the other proofreaders for their helpful suggestions that significantly improved my thesis.

I would particularly like to thank my family for their patience and encouragement. Especially I want to thank my wife Sandra who always believed in me. This work is dedicated to you.

Abstract

Technical systems are getting more and more complex. One reason for this is the constantly growing proportion of software within systems. Consequently, safety analysis, which is mostly required for the approval process, is becoming increasingly complex and leads to higher development costs. Model-based safety analysis (MBSA) can significantly improve the development process. In model-based safety analysis, safety and design engineers work on a common model and thus always with the same state. Analyses can be automated and at the same time, the results obtained are objective and do not depend on the engineers' skills. The analyses require less time and the development costs decrease. Safety analyses should be performed in the early stages of the development process. The later errors are discovered, the more expensive it is to fix them. When a system is developed, usually only a rough idea of the system behavior is available. Therefore, special formalism is needed.

This thesis introduces the smartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs) formalism, which is particularly developed for the safety analysis of systems in the early development phase. smartIflow is a qualitative approach in which the components in the system are considered as finite-state machines. The communication between the components is extremely flexible. The components can publish so-called properties at the connection points which are propagated over bidirectional connections to all reachable components. With this mechanism, systems can be represented very well at the logical level. Depending on the focus of the analysis, systems can also be described at the physical level using the qualitative energy flow analysis. State changes of a component can be caused by internal or external influences. It will be addressed how temporal behavior can be described on an abstract level. Furthermore, the smartIflow language is going to be introduced, which can be used to describe the models in a computer-understandable language.

Unfolding is used to determine the transition system of the system models. It is discussed how the state space can be limited. An important task in the development of systems is the verification of safety requirements. In the thesis, a method for automated verification of a specification against a model is described. The algorithm delivers all relevant counterexamples in case of a violation of the specification. This is an essential difference to the existing model checkers. Moreover, an extension of the temporal logic CTL is going to be discussed, so that the temporal behavior of the smartIflow system models can be considered as well. From the counterexamples the minimal cutsets can be determined, which can be represented as a fault tree. In the thesis, an approach for efficient modeling is discussed, too. The practical usability of the approach is demonstrated by some case studies.

Zusammenfassung

Technische System werden zunehmend komplexer. Ein Grund dafür ist der stetig wachsende Softwareanteil. Folglich werden die Sicherheitsanalysen, die meist für den Zulassungsprozess erforderlich sind, zunehmend aufwändiger und treiben die Entwicklungskosten in die Höhe. Modellbasierte Sicherheitsanalyse (MBSA) kann den Entwicklungsprozess signifikant verbessern. Bei der modellbasierten Sicherheitsanalyse arbeiten die Sicherheits- und Konstruktionsingenieure an einem gemeinsamen Modell und somit immer mit dem gleichen Zustand. Analysen können automatisiert werden und gleichzeitig sind die erzielten Ergebnisse objektiv und hängen nicht von den Fähigkeiten der Ingenieure ab. Die Analysen benötigen weniger Zeit und die Entwicklungskosten sinken. Die Sicherheitsanalysen sollten bereits im frühen Entwicklungsstadium durchgeführt werden. Je später Fehler entdeckt werden, desto teurer ist es diese zu beheben. Bei der Entwicklung eines Systems ist zu Beginn meist nur eine grobe Idee vom Systemverhalten vorhanden. Dafür werden spezielle Formalismen benötigt.

Diese Thesis stellt den smartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs) Formalismus vor, der insbesondere für die Analyse von System im frühen Entwicklungsstadium geeignet ist. smartIflow ist ein qualitativer Ansatz, bei dem die Komponenten im System als endliche Zustandsautomaten betrachtet werden. Die Kommunikation zwischen den Komponenten ist extrem flexibel. Die Komponenten können sogenannte Properties an die Anschlusspunkte legen, welche über bidirektionale Verbindungen an alle erreichbaren Komponenten propagiert werden. Mit diesem Mechanismus können Systeme auf logischer Ebene beschrieben werden. Abhängig vom Analysefokus können die Systeme mit Hilfe der qualitativen Flussberechnung auch auf Ebene der physikalischen Wechselwirkungen beschrieben werden. Zustandsänderungen einer Komponente können durch interne oder externe Einflüsse erfolgen. Es wird diskutiert, wie zeitliches Verhalten auf einer abstrakten Ebene beschrieben werden kann. Weiterhin wird die smartIflow Sprache vorgestellt, mit der die Modelle in einer für den Computer verständlichen Sprache beschrieben werden können.

Durch Entfaltung wird der Zustandsraum der Systemmodelle bestimmt. Es wird diskutiert, wie der Zustandsraum begrenzt werden kann. Eine wichtige Aufgabe bei der Entwicklung von Systemen ist die Überprüfung von Sicherheitseigenschaften. In der Thesis wird ein Verfahren zur automatischen Überprüfung einer Spezifikation gegen ein Modell beschrieben. Der Algorithmus liefert bei einer Verletzung der Spezifikation alle relevanten Gegenbeispiele. Dies ist ein wesentlicher Unterschied zu den existierenden Modellprüfern. Des Weiteren wird eine Erweiterung der Temporale Logik CTL diskutiert, dass auch das zeitliche Verhalten der smartIflow Systemmodelle berücksichtigt werden kann. Aus den Gegenbeispielen können die minimalen Schnittmengen bestimmt werden, welche als Fehlerbaum dargestellt werden können. In der Thesis wird auch ein Ansatz zur effizienten Modellierung diskutiert. Die praktische Nutzbarkeit des Ansatzes wird anhand einiger Fallstudien gezeigt.

Contents

1 Introduction

Safety analysis plays an important role in the development of safety-critical systems. It is essential to verify design concepts and requirements as soon as possible, preferably before they are implemented (see Figure 1.1).

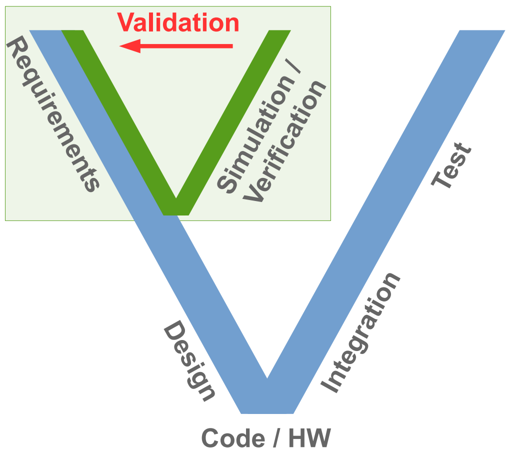


Figure 1.1: The V-Model with early verification

Discovering design flaws at a later stage makes it more expensive to fix them. Undetected design flaws can lead to failures or critical situations causing considerable material and environmental damage, or even to the loss of life. If a faulty system is already on the market, it can be extremely expensive and potentially dangerous. Expensive recall actions are necessary to fix the flaws and at its worst, people may have already been injured.

There are numerous examples from the past where undetected design flaws led to considerable material and environmental damage or even to the loss of life. The best-known example is probably the launch of Ariane 5 in 1996 [JM97] as the flight ended already a few seconds after take-off. The rocket drifted off course due to a malfunction in the control software and in consequence self-destructed. The error caused a loss of 290 million Euros. Apart from the total loss of the rocket, fortunately, no one was injured.

A software bug in the radiation therapy machine Therac-25 led to a massive overdose of radiation [Lev17]. As a consequence, three cancer patients died between 1985 and 1987 and others were seriously injured. Appropriate countermeasures were not taken until after these incidents.

A more recent example is the malfunction of the Teslas autopilot feature that caused a fatal crash in which the driver was killed [Tes]. The crash happened when a truck crossed the road in front of the autonomously driving Tesla Model S. The autopilot probably could not distinguish the white painted side of the truck from the sky behind it and therefore crashed into the truck.

Safety analysis techniques are applied during the development process of systems to prevent the above-mentioned situations or to minimize the probability of their occurrence. However, there are even more reasons why safety analysis is so crucial: Failures can have significant financial consequences, including repair costs and potential fines. Safety analysis helps in identifying and preventing situations that could lead to financial consequences. Moreover, unplanned downtime of systems due to accidents or failures can affect processes and therefore lead to additional costs. Many industries define strict regulations and standards to ensure the safety of their technical systems. Compliance with these regulations is often essential for the certification of the systems.

Of course, there is no guarantee for 100% safety, but it is possible to minimize failure probabilities. Developing safe systems has always been a challenge and will not get easier. One challenge in safe product design is the countless ways in which real-world systems can fail. Another challenge is the increasing complexity of the systems. Ubiquitous computing, autonomy, or the high connectivity of different subsystems enable innovative systems, but at the same time, they lead to enormous complexity. Imagine for example a modern car featuring an automated driving system (e.g., a traffic jam assistant, which automatically brakes, accelerates, and stays in lane at low speeds). Usually, such cars are equipped with more than 150 Electronic Control Units (ECUs) that communicate with each other over the CAN bus [Ecu]. Each subsystem considered individually is already complex, but the interconnections between them further increase the complexity. Additionally, the system must react to changes in environmental conditions in real-time. It is impossible to analyze such systems manually.

Safety analysis, no matter whether it is done manually or automated, is an iterative task. Each time the system design changes, the requirements must be verified all over. In manual safety analysis, this requires a lot of time, leading to high costs. Furthermore, the quality of the results of a manual analysis depends on the skill of the safety engineer. Even if two safety engineers come to the same result, it does not automatically mean that the result is flawless. To obtain objective results and reduce costs, safety analysis must be automated.

1.1 Model-Based Safety Assessment

Model-Based Safety Assessment (MBSA) as described in [Jos+05] is an approach that tries to address some of these problems. Nowadays, model-based techniques are well established in industry for the analysis and verification of the nominal behavior of a system. The central point is a formal and computer-understandable model of the nominal system behavior, which is used for various tasks such as code generation or testing. MBSA is inspired by these traditional model-based techniques. Figure 1.2 depicts the MBSA approach. To automate the safety analysis process, MBSA extends the system model by a fault model (=extended system model). The fault model contains information about possible failure modes and their effects. In addition to an extended system model, safety requirements must be specified in a formal and computer-understandable form. Analysis techniques are used to verify the extended system model against the safety requirements and to deliver safety artifacts such as Fault Trees.

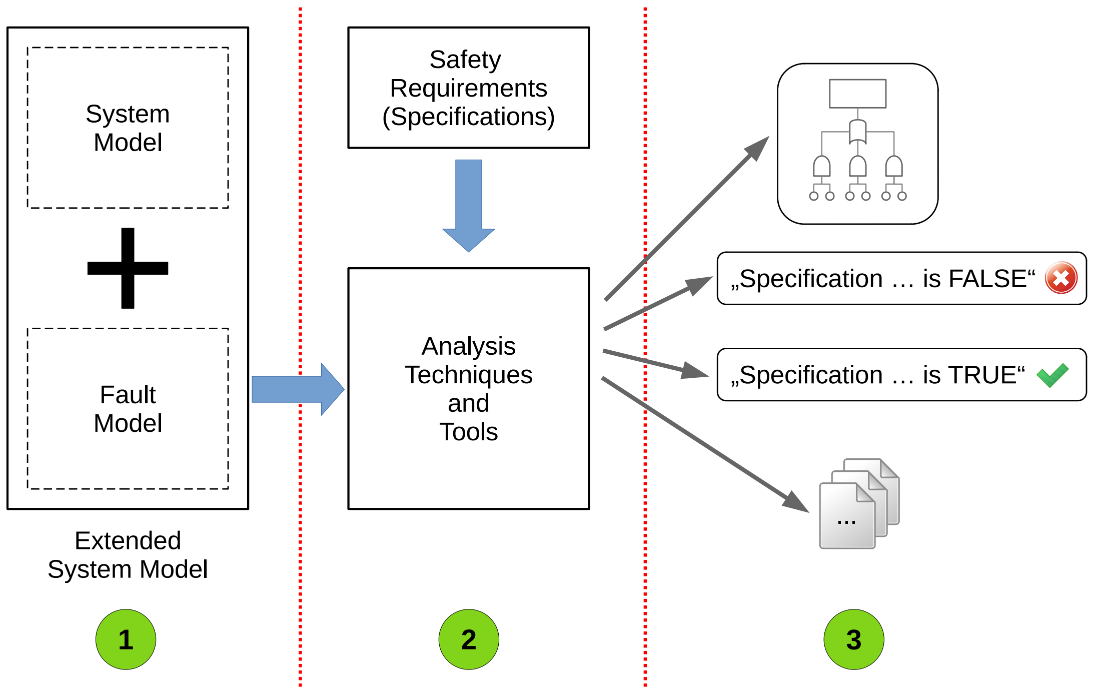

System Model
Fault Model
Extended System Model
Safety Requirements (Specifications)
Analysis Techniques and Tools
„Specification … is FALSE“
„Specification … is TRUE“
…
1
2
3

Figure 1.2: The model-based safety analysis process

MBSA enables safety and system engineers to work on the same model. On top of that, results are objective, reproducible, and the quality does not depend so much on the skills of the safety engineer. Different system designs can be compared efficiently, and the knowledge is reusable. Of course, the results are only as good as the models. An incorrect model will also produce incorrect results.

1.2 Motivation

The MBSA concept merely proposes the use of an extended system model but does not describe any modeling formalism.

A central challenge in the automation of safety-related tasks is the choice of an appropriate level of abstraction under which the systems are considered. Finding a good balance between abstraction and high fidelity constitutes a major challenge. Detailed system models deliver exact and comprehensive results, but at the same time lead to a high computation effort. Simulation tools like Simulink or Simscape are designed to analyze specific scenarios on a quite detailed level of abstraction. Simscape uses physical system models that deliver detailed values about timing or physical signals. To create such models, deep knowledge about the system is required.

This leads to the next problem: Ignorance. In the early phases of product development, the exact behavior of components in a system is often unknown. However, detailed system models do not necessarily lead to better predictions. In real systems, the behavior often depends on a large set of parameters (e.g., environmental temperature or system runtime) that may change over time. It is not possible to reflect all these external influences in system models. Therefore, even with a precisely defined behavior, the predictions only cover small parts of what actually happens.

Qualitative abstraction is the key to the problems mentioned above. The idea is to combine similar real states into one qualitative state. For example, the speed of a car can be reduced to the qualitative states `Stopped`, `Accelerating`, `Moving`, and `Braking`. This abstraction reduces the state space significantly. With a quantitative approach, there would be innumerable states. Qualitative models describe how the different qualitative states are reached. As long as the same state is reached, all extraneous implementation details in the model can be omitted.

A drawback of qualitative abstraction are ambiguities. For instance, if the amount of flow through pipes is modeled by signs (+,-,0), there are often situations where the exact behavior is unpredictable. An extension such as order-of-magnitude reasoning improves the reliability of the results.

Existing qualitative approaches to automated safety analysis use quite different levels of abstraction. HiP-HOPS [PM99] is an approach using an extremely high level of abstraction. System models are created in Simulink. Communication between components is unidirectional and limited to the propagation of deviations. Each Simulink component is assigned a failure logic that describes how deviations at the input or internal failures cause deviations at the output. HiP-HOPS supports automated generation of Fault Trees and FMEA tables. However, the directed connections, the limitation to failure behavior, and the lack of state variables significantly limit the applicability in practice. Some approaches to automated safety analysis utilize formal verification tools. Joshi et al. [Jos+05] propose an approach in which Simulink is used to create an extended system model. Once the system is modeled, the Simulink block diagram is translated into the input language of the NuSMV model checker. The model checker automatically verifies the system model against safety requirements specified in some formal language. If the model checker finds a specification violation, the algorithm stops and provides a counterexample. Another approach uses a formal modeling language and model transformations to support several verification tools (e.g., [GO11]). One problem with these approaches is that most model-checking tools only deliver one counterexample. Apart from that, these modeling formalisms are restricted to directed connections and timing aspects can only partly be covered. AltaRica 3.0 is the latest and most advanced version in the history of AltaRica. The level of abstraction of AltaRica 3.0 is somewhere between physical system models and HiP-HOPS. Components in systems are characterized by classes. AltaRica 3.0 is based on Guarded Transition Systems and is therefore able to handle looped systems and bidirectional information flow. However, communication between components is still limited and events can not be triggered by components within the system.

1.3 Mission Statement

This thesis aims to introduce a new modeling formalism for automated safety analysis of technical systems in the early stages of product development. To overcome the limitations of the existing approaches, the new modeling formalism must fulfill some requirements:

- Component behavior must be described on a qualitative level
- The modeling formalism should be component-oriented
- Component models should include both, nominal and failure behavior
- Connections between components must be bidirectional and can change depending on the state
- Components exchange quite different information, such as sensor values, failure modes, or abstracted physical quantities. Thus, communication between components should be as flexible as possible
- Timed aspects need to be described qualitatively
- Mechanisms to limit search spaces are required

The system models need to be created in a formal and computer-understandable form. Therefore, this work introduces an intuitive textual language. Since safety engineers usually do not have much experience with modeling languages, a graphical modeling approach is also discussed. Safety engineers need to know whether a system meets certain specifications. Formal verification methods such as model checking are promising candidates for this task since they are fully automated. However, existing tools are not appropriate for real-world applications (e.g., most model checkers deliver only one counterexample if a specification is not fulfilled). Therefore, a new verification algorithm must be developed. The verification results must be available in an appropriate form for safety analysts (e.g., as Fault Trees). To sum up, the objectives of this thesis are:

- The introduction of a new modeling formalism for safety analysis of technical systems in the early product life cycle states
- The definition of a textual modeling language and an approach for efficient modeling
- The development of mechanisms for automated verification of safety requirements and generation of safety artifacts
- The demonstration of the practicability of the approach using case studies

1.4 Contributions beyond the State of the Art

The following points summarize the contributions of this work that go beyond the state of the art:

- **Contribution C1:** New modeling formalism for automated safety analysis

  The proposed formalism (smartIflow) is a combination of features of existing approaches together with new concepts:

- smartIflow system models are created on a very high level of abstraction, but at the same time, the dynamic aspects of a system can be described on a very detailed level. In contrast to many existing approaches, component models are quite close to the specifications.
- Many approaches limit communication between components to the exchange of values from predefined domains (e.g., HiP-HOPS). In smartIflow, the property concept enables a very flexible message exchange between components.
- In [SL14], Snooke and Lee present an approach in which events in component models are assigned time delay periods of a certain order of magnitude. In a state, only the active events with the lowest delay value are triggered. In smartIflow, a differentiation is made between internally and externally triggered transition statements. Therefore, the mechanisms for specifying timed behavior in smartIflow are also more comprehensive. First, internally triggered transition statements can be assigned delay or stability values. Stability values limit the activation of internal events while delay values are used to model delayed reactions. Beyond that, externally triggered transition statements can be assigned stability conditions that limit the activation of external events to states that are stable at least with respect to the specified value. In this way, the frequency of external events can be controlled.
- In smartIflow, physical interactions between components can be modeled by means of qualitative energy flows. The qualitative energy-flow-based reasoning based on different orders of magnitude is nothing new. The concept is based on bond graphs first described by Paynter [Pay60]. In the meanwhile, different versions of bond graphs were developed and used by many approaches. An innovation is, that the qualitative energy flow analysis can be used together with a communication mechanism based on messages. On the one hand, systems, or parts of them, can be represented quite detailed on a physical level using the qualitative energy flow analysis as introduced in [SL14]. On the other hand, systems, or parts of them, can be described at the logical level using the message exchange mechanism.
- In contrast to other approaches, connections between components are in general bidirectional, and transferred information is not limited to one specific domain. Optionally, the direction of flow at ports can be restricted to inputs or outputs. While many approaches (e.g., AltaRica or HiP-HOPS) only allow one value per connection at a time, connections in smartIflow can simultaneously transport several messages. This enables a bus-like information exchange. Thus, artificial connections between the components can be avoided, and models are quite close to the specification.
- Compared to existing approaches, the mechanisms for modeling state transitions in smartIflow are much more comprehensive. For example, a distinction is made between internal and external events. This leads to smaller state spaces during unfolding since internal transitions are executed synchronously. Moreover, a differentiation can be made between necessary system reactions and possible

external stimuli.

- The smartIflow language is object-oriented and strongly inspired by Java, making it very easy to learn. Compared to other modeling languages, many concepts of object-oriented programming languages are offered (e.g., inheritance or type variables). Another innovation is the feature concept, which enables the extension of component classes (or parts of them) without having to extend the language itself.

- **Contribution C2:** Formal definition of smartIflow system model

  If parts of the safety analysis process are automated, it is very important to have a clear understanding of the modeling formalism. In contrast to some existing formalism, the semantics of smartIflow system model, the underlying transition systems, and the new timed variant of CTL are formally defined.

- **Contribution C3:** Flexible and effective mechanism for search space limitation

  Existing approaches to Model-Based Safety Assessment utilize quite simple constraints to keep the search space manageable (e.g., an upper limit for failure events). Event Trigger Specifications (ETS) also allow to limit the state space during unfolding. Compared to existing mechanisms, ETS are more powerful and flexible. For example, the state space can be restricted to nominal behavior or fault scenarios can be restricted to a maximum of two failure events. External events can be categorized into different groups which can be considered in Event Trigger Specifications. Beyond that, different filter conditions can also be chained together.

- **Contribution C4:** Model checking algorithm for generation of all relevant counterexamples.

  Most model checkers (e.g., NuSMV) stop the verification after the first counterexample is found. Only a few model checkers can generate multiple counterexamples, however, a lot of counterexamples are variations of each other and therefore do not lead to new knowledge. The model checker introduced in this thesis differs from existing tools in two aspects:

  1. To check whether a system fulfills a certain requirement, the model checker systematically explores the system behavior for all possible sequences of events (including failure events) by performing a simple depth-first search. Most existing model-checking tools utilize symbolic algorithms or are based on the calculation of satisfaction sets. Different from most other implementations the proposed model checker can produce all relevant counterexamples if a specification is not fulfilled. Thus, cutsets and Fault Trees can be created, which makes this approach so unique.
  2. In contrast to the existing model checkers, this approach enables to consider only paths that fulfill certain relevance criteria (e.g., paths with a maximum of two failure events) when traversing the transition system. This limits the search space significantly.

- **Contribution C5:** A new variant of CTL to express timed properties for smartIflow system models

  There already exists timed variants of CTL. TCTL, for example, can be used to express properties to timed automata. In smartIflow system model, timed behavior relies on different orders of magnitude of time. Therefore, existing timed variants of CTL are not compatible. Orders of Magnitude timed CTL (OMT-CTL) introduces a new timed variant of CTL that can consider the stability of states and the delays between state transitions based on different orders of magnitude in time.

- **Contribution C6:** Successful integration in an industrial project

  The presented modeling approach does not only exist in theory but has also been implemented in a prototypical tool (smartIflow Workbench) which has been successfully used in an industrial project. The integration in Simulink/Simscape enables efficient modeling in a familiar modeling environment as well as the conversion of the architecture of an existing model into the smartIflow language.

1.5 Structure of the Thesis

This dissertation introduces a new modeling formalism for the safety analysis of technical systems in early product life cycle stages. The structure of this thesis is as follows: Chapter 2 provides a brief description of the most important definitions. In addition to terminology, model-checking techniques and an algorithm for determining strongly connected components in directed graphs are explained. In Chapter 3 some safety-related model-based approaches are introduced and their differences are analyzed. Chapter 4 starts with the motivation for a new modeling formalism. It discusses how to overcome the weaknesses of existing approaches and introduces the new modeling formalism formally. This chapter also describes the modeling language and the unfolding algorithm that is used to create the transition system. A way is shown that reduces the search space and an approach for efficient modeling is introduced. Chapter 5 is dedicated to the formal verification of safety requirements. It is shown how an existing temporal logic is extended to describe safety requirements for the models. This chapter also focuses on the model checking algorithm, the calculation of minimal cutsets, and the generation of Fault Trees. In Chapter 6, the practicality of the new modeling formalism is shown with some example systems. Chapter 7 summarizes this thesis and discusses possible directions for future work. Last but not least, Appendix A contains the grammar of the proposed modeling language, the grammar of the logic to describe safety requirements, and the grammar to reduce search space.

2 Basic Definitions and Concepts

Before focusing on the basic definitions and concepts, it might be helpful to look at an example system. By using this system, we will explain the basic techniques and concepts. An Electric Park Brake (EPB) is available in many modern cars today [Bay+14]. A parking brake in a car is designed to block the wheels permanently and to prevent the vehicle from rolling away. In mechanical parking brake systems, the brakes are connected via cables to a pulling mechanism that can be operated by the driver. Modern cars are equipped with Electric Park Brake systems that have several advantages in comparison to mechanically operated parking brakes:

- No issues with freezing handbrake cables
- It takes much less force to activate or release the brake
- High braking power
- Childproof (in most cases the ignition must be switched on to release or activate the brake)

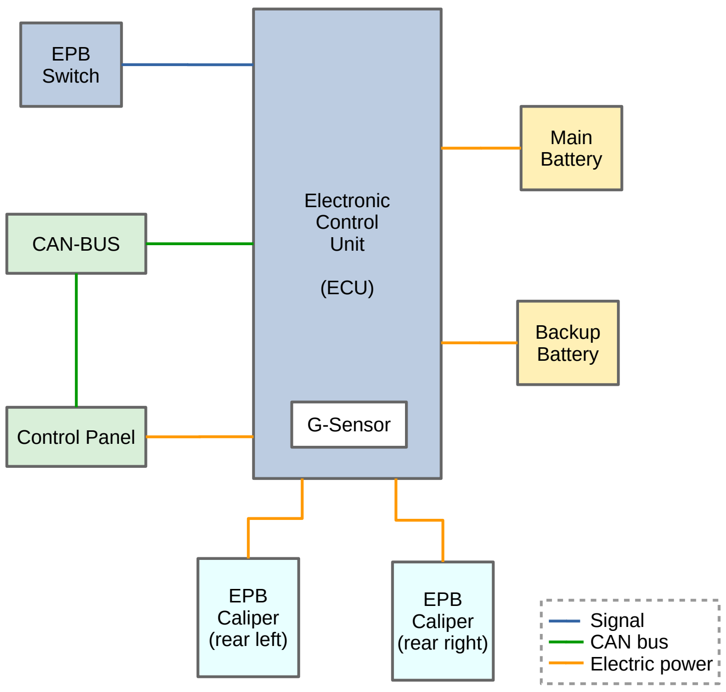


Figure 2.1: Electric Park Brake system

Figure 2.1 depicts the structure of an Electric Park Brake system. The central component is an ECU (Electronic Control Unit) that processes the input signals provided by various sensors and controls the motors at the brake calipers. Due to mechanical pressure emerged by the motors, the calipers are pressed onto the brake disks. The system automatically calculates the braking power that is necessary to prevent the vehicle from rolling away. The Electric Park Brake system is connected to the CAN (Controller Area Network) bus which enables access to sensor values of the car (seat belt sensor, tilt sensor, road speed sensors, ...). Therefore, even when the brake discs cool down whereby the material contracts, the system adapts the braking power to ensure a safe stand.

2.1 Safety Engineering and Terminology

This section introduces some important safety-related definitions used throughout this thesis. The following definitions are based on the work of Avizienis et al. [Avi+04].

A system, whether a hardware or software system, has the task of providing several services. Users (humans, other systems, or the environment) can interact with these services using the service interfaces. The situation in which the service result deviates from the intended behavior is called **service failure** (or just **failure**). The ways in which a system may fail are called **failure modes**. The situation when one or more external states of the system deviate from the correct behavior is referred to as **error**. An error is a deviation in a single state or in a sequence of system states. Errors are caused by **internal or external faults**. A fault is a flaw in software or hardware. A fault must not necessarily lead to a service failure, since the system could detect, react, and consequently avoid a failure. A **hazard** is the undesired situation that can cause harm or damage to humans or the environment.

In the case of the electric parking brake system, potential failure modes could be as follows:

- Total loss of brake power
- Insufficient braking power
- Unintended activation of the brake while driving

An error is defined as incorrect values in the system state as for example a valve that is closed but should be opened. Such an error can be caused by one or more of the following faults:

- Programming mistake, e.g., signals on the bus are interpreted incorrectly
- Defective switch that enables respectively disables the parking brake
- Material fatigue, e.g., a leakage in a hydraulic line

To avoid such hazards there exist a lot of standards and norms in industry that define a process with required activities and work products as well as methods for development and production. The regulation authorities require compliance with the standards. Well-known standards are for example ISO 26262 (“Road vehicles – Functional safety”), SAE ARP4761 (“Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment”), or IEC61508 (“Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”). Classical safety assessment methods, such as Fault Tree Analysis (FTA) or Failure Modes and Effects Analysis (FMEA) are often part of these standards.

2.2 Classical Safety Assessment Methods

In this section, some traditional, industry-relevant safety assessment methods are introduced.

2.2.1 Fault Tree Analysis

Fault tree analysis (FTA) is a widely-used technique for safety analysis in the industry in which the causes of an undesired system effect (hazard) are explained in terms of event combinations [Lee+85]. Such events can be either component failures or human operation errors. In addition, there might be also intermediate events. Events can be combined using *AND* and *OR* gates as shown in Figure 2.2. In this example hazard *H1* is caused by one of the following combinations:

- e_1 or
- e_2 and e_3 or
- e_4 and e_5

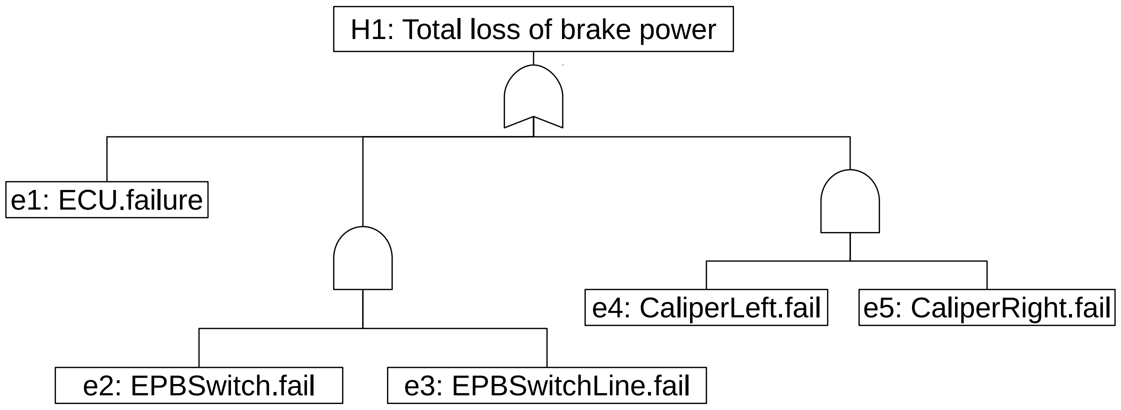


Figure 2.2: Fault tree for hazard “Total loss of brake power”

Fault tree analysis is a top-down, deductive failure analysis method that starts at the system hazard level and successively moves down to component failures (basic events) to find all possible causes. After the fault tree has been created, the evaluation can be started. In qualitative analysis, it is the goal to find groups of events that cause the top event. Such groups are called *minimal cut sets* if they do not contain any redundant element. The top event only occurs, if all events of a minimal cut set occur.

In quantitative fault tree analysis, each basic event is assigned a probability of occurrence. By iterative calculation of the probabilities at the AND- and OR-gates, the probability of the top-event can be calculated. Note that the basic events need to be statistically independent. The probability of an AND-gate with two inputs ($P(A)$ and $P(B)$) is just the probability that both systems fail:

$$P(A \,\text{and}\, B) = P(A \cap B) = P(A) \cdot P(B)$$

In the case of an OR-gate, the probability is calculated using the formula

$$P(A \,\text{or}\, B) = P(A \cup B) = P(A) + P(B) - P(A) \cdot P(B)$$

2.2.2 Failure Mode and Effects Analysis

Failure Mode and Effects Analysis (FMEA) is a bottom up, inductive analysis technique [MMB09]. The objective of FMEA is the identification of dependencies between causes and consequences, but also the prioritization of these cause-effect relations in terms of their risk. The analysis starts with an identification of all possible failures on the component level. For each failure mode, the effects and causes are determined. Frequently an evaluation of failure modes and consequences according to severity is performed. The result of an FMEA analysis is a table like shown in Table 2.1. There is no common standard format for a FMEA table, i.e., most industries define their own formats and processes.

Component	**Failure Mode**	**Potential Effects**	**Severity**	**Potential Causes**	**...**
Electronic Control Unit: Controls the motors placed at the calipers depending on the switch signal.	Output port	Loss of braking power. The car could roll away.	5	Fatigue	
	Failure in control logic			Excess pressure	
	Stuck at position intermediate			Loss of control signal	

Table 2.1: Example FMEA table

2.3 System Modeling Fundamentals

2.3.1 Transition Systems

Many approaches to model-based safety analysis describe the system under analysis as a transition system. As defined in [BK08], a transition system is a tuple

$$TS = (S, Act, \rightarrow, I, AP, L)$$

where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

Actions cause the model to change its current state as described by the transition relation. States are labeled with propositions that provide the base for the specification of safety properties. Finite-state machines are quite similar to transition systems. In the latter case, however, the set of states does not necessarily have to be finite.

For a state $s \in S$ of a transition system $TS = (S, Act, \rightarrow, I, AP, L)$ the set of direct successors of s is defined as:

$$Post(s) = \bigcup_{\alpha \in Act} \{s' \in S \mid s \xrightarrow{\alpha} s'\}$$

Accordingly, the predecessors of s are defined as:

$$Pre(s) = \bigcup_{\alpha \in Act} \{s' \in S \mid s' \xrightarrow{\alpha} s\}$$

2.3.2 Program Graphs

A program graph PG is a graph consisting of locations (=nodes) and transitions (=edges) that are labeled with conditions over variables [BK08]. This is an important distinction in comparison to transition systems that do not have conditional transitions. Formally a program graph PG over a set of typed variables Var is a tuple

$$PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$$

where

- Loc is a finite set of locations and Act is a set of actions,

- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$ is the effect function,
- $\hookrightarrow \subseteq \mathit{Loc} \times \mathit{Cond}(Var) \times \mathit{Act} \times \mathit{Loc}$ is the conditional transition relation,
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is a set of initial locations, and
- $g_0 \in \mathit{Cond}(Var)$ is the initial condition.

$\mathit{Eval}(\mathit{Var})$ denotes the set of variable evaluations over Var, i.e., all feasible variable-value combinations. The variable values determine not only the possible transitions but also the behavior at locations. Variable values are modified by effect functions. Effect functions are executed once the condition of a transition holds. Conditions of transitions are called guards. The transition system of each program graph represents the state space and can be gathered by unfolding [MP92].

2.3.3 Timed Automata

Timed automata are the key techniques for modeling time-critical systems. In principle, a timed automaton is a program graph extended with a finite set of clocks and clock constraints at transitions [BK08]. Formally, a timed automata is a tuple $TA = (\mathit{Loc}, \mathit{Act}, C, \hookrightarrow, \mathit{Loc}_0, Inv, AP, L)$ where

- Loc is a finite set of locations,
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is a set of initial locations,
- Act is a finite set of actions,
- C is a finite set of clocks,
- $\hookrightarrow \subseteq \mathit{Loc} \times CC(C) \times \mathit{Act} \times 2^C \times \mathit{Loc}$ is a transition relation,
- $Inv : \mathit{Loc} \to CC(C)$ is an invariant-assignment function,
- AP is a finite set of atomic propositions, and
- $L : \mathit{Loc} \to 2^{AP}$ is a labeling function for the locations.

Like stopwatches, clock values are increased continuously with the same speed. The access to the clocks is limited to read-only, but optionally clocks can be updated (e.g., reset to 0). $\mathit{Eval}(C)$ are evaluation functions that determine the current value of clocks C. The edges describe how the timed automaton can move between locations ($l \overset{g:\alpha,D}{\hookrightarrow} l'$). In the transition relation, 2^C denotes all possible combinations of clocks that can be reset. These edges are labeled with the tuple (g, α, D) where g is a clock constraint, α is an action, and $D \subseteq C$ is a set of clocks that will be reset to 0. $CC(C)$ denotes the clock constraints over clocks C, and $ACC(C)$ denotes the atomic clock constraints over clocks C. Clock constraints at transitions are logical expressions built upon comparisons between clock values and integers. Typically, a clock constraint specifies a time interval, for example, $cc_1(x) = x >= 2 \wedge x < 5$. Such time intervals are often abbreviated with an interval notation: $[c1, c2)$. These expressions can be seen as guards that initiate the transitions only under specific timing conditions. Inv is the invariant of a location that specifies the duration of time that can be stayed at the location. In summary, clock variables can be used in guards, invariants, and updates.

An example of a timed automaton is shown in Figure 2.3. The timed automaton depicts the behavior of a simple traffic light with a 3-stage sequence. The automaton consists of three locations (*green*, *yellow*, and *red*) and one clock variable (x). Initially, the automaton remains in location *green*. From there it is possible at any time, regardless of the value of the clock, to move to location *yellow*. The invariant in location *yellow* states: Once $x > 5$, the automaton must leave this location.

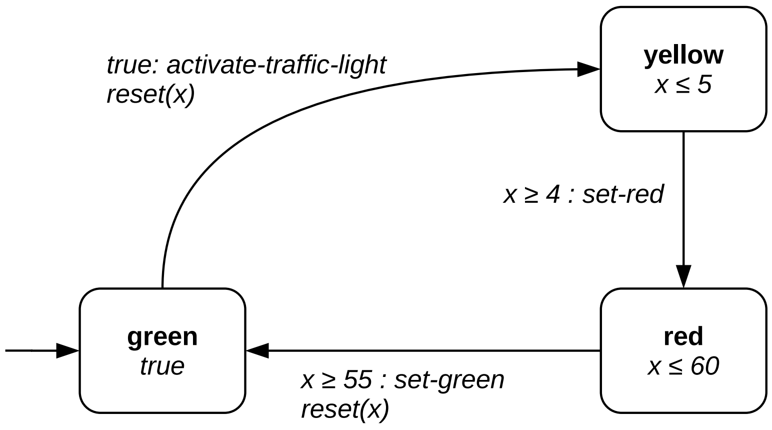


Figure 2.3: Traffic light represented as a timed automaton using clock variable x.

2.4 Model Checking

Model checking [BK08][Cla08] is a formal method for verifying the correctness of safety-critical systems. This verification technique is also known as "push-button" method since no user interaction is needed. Figure 2.4 illustrates the principle idea of model checking. Once the system model M is created and a correctness property ϕ is specified, the model checking tool will automatically check whether the system satisfies these requirements. In other words: Does $M \models \phi$? The system model describes how the system behaves while the requirements specification describes what the system should do (e.g., reaction to a certain event). Such system models are mostly characterized by transition systems such as Kripke structures [CGP99]. A Kripke structure M over a set of atomic propositions AP is the tuple $M = (S, S_0, R, L)$ where

- S is a finite number of system states
- $S_0 \subseteq S$ is a set of initial system states
- $R \subseteq S \times S$ is transition relation. This relation must be total, i.e., each state $s \in S$ must have a subsequent state $s' \in S$ $(R(s, s'))$

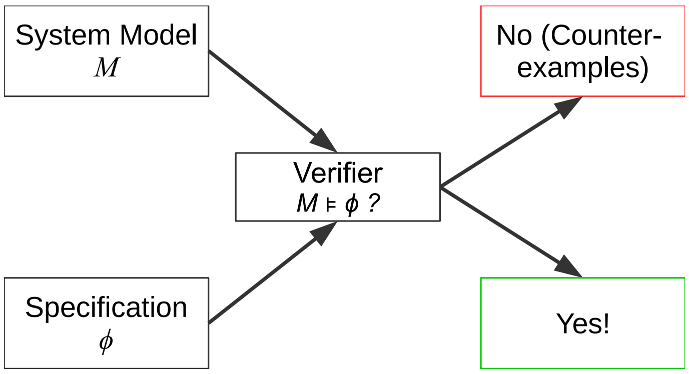
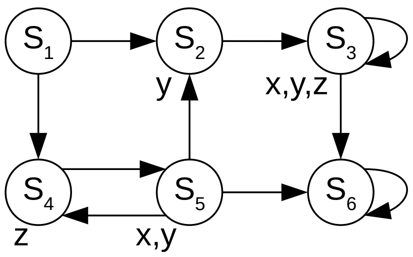


Figure 2.4: The principle of model checking

- $L : S \to 2^{AP}$ is a labeling function that determines the set of atomic propositions that are true in a state

Figure 2.5 shows an example of a Kripke structure.



$$
\begin{aligned}
M &= (S, S_0, R, L)\\
AP &= \{x, y, z\}\\
S &= \{S_1, S_2, S_3, S_4, S_5, S_6\}\\
S_0 &= \{S_1\}\\
R &= \{(S_1, S_2), (S_1, S_4), (S_2, S_3), (S_3, S_3), (S_3, S_6),\\
&\quad (S_4, S_5), (S_5, S_4), (S_5, S_2), (S_5, S_6), (S_6, S_6)\}
\end{aligned}
$$

$$
\begin{array}{lll}
L(S_1) = \{\emptyset\} & L(S_2) = \{y\} & L(S_3) = \{x, y, z\}\\
L(S_4) = \{z\} & L(S_5) = \{x, y\} & L(S_6) = \{\emptyset\}
\end{array}
$$

Figure 2.5: Example of a Kripke structure

The safety requirements need to be specified in some formal and computer-understandable language. Most model-checking tools accept requirements properties specified in some temporal logic, introduced by Pnueli [Pnu77]. In the meanwhile, different variants of temporal logic have been developed. *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL) are probably the best-known and most important temporal logic.

2.4.1 Linear Temporal Logic

LTL (Linear Temporal Logic) is based on a linear-time perspective [JK04]. In contrast to the propositional logic, where propositions about a single state can be expressed, LTL allows statements about the future of paths. These formulae are verified on each possible execution path of a system (linear time). A LTL formula is built up from a set of atomic propositions AP, logical operators $(\vee, \wedge, \neg, \rightarrow)$, and temporal operators (X, U, G, F, and R). The semantics of the temporal operators is defined as follows:

- $\mathsf{X}\,\phi$: ϕ must hold in the ne**x**t state.
- $\mathsf{G}\,\phi$: ϕ must be satisfied **g**lobally on the complete path.
- $\mathsf{F}\,\phi$: ϕ must hold in a **f**uture state, somewhere on the path.
- $\psi\,\mathsf{U}\,\phi$: ϕ is true in the next or some future state. **U**ntil then, ψ must be satisfied.
- $\psi\,\mathsf{R}\,\phi$: ϕ must be true up to the first state when ψ is valid. If ψ is never satisfied, ϕ must be true on the complete path.

Figure 2.6 shows some examples of LTL formulae that are valid on the path. LTL is often used to express safety properties to a system. Such a safety property describes an unwanted situation ϕ that should never occur ($G\neg\phi$).

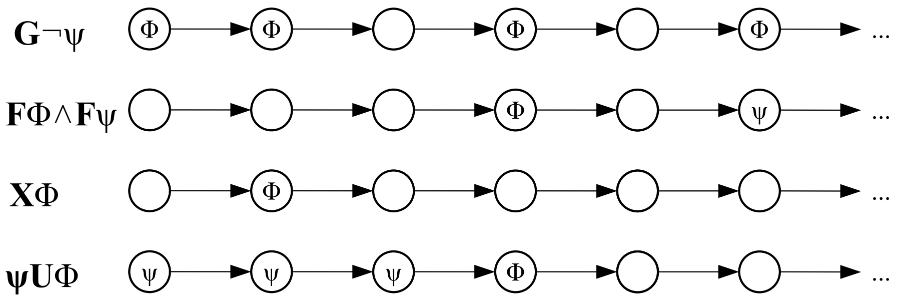


Figure 2.6: Examples of LTL formulae

2.4.2 Computation Tree Logic

The temporal logic CTL is a branching-time logic which means that different futures can be considered [CE82]. To achieve this, CTL provides not only the usual logical operators and temporal operators (X, G, F, U, and R like in LTL), but also two path quantifiers namely A ("along **A**ll paths") and E ("there **E**xists at least one path"). A path quantifier is always followed by a temporal operator, resulting in a total of 10 combinations (AX, AG, AF, AU, AR, EX, EG, EF, $\mathsf{E\,U}$, and ER). Figure 2.7 visualizes three examples of valid CTL expressions.

AG p:
On all paths proposition p is always true.

AF p:
On all paths p is somewhere in the future satisfied.

A$[p\,\mathbf{U}\,q]$:
On all paths p is true until the first occurrence of q

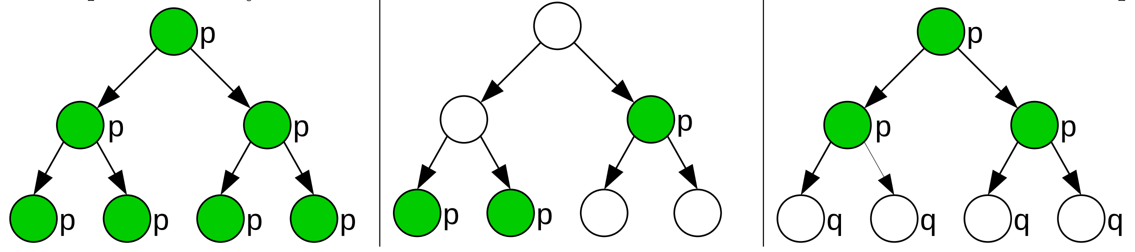


Figure 2.7: Examples of valid CTL formulae

Note that any CTL expression can be converted to the existential normal form (ENF). A CTL formula in ENF consists only of the operators $\wedge$, $\neg$, $\mathsf{EX}, \mathsf{E}\,\mathsf{U}$, and EG. Each CTL formula can be converted into an equivalent CTL formula in ENF using the following conversion rules:

- $\mathsf{AX}(\phi) = \neg\,\mathsf{EX}(\neg\phi)$
- $\mathsf{EF}(\phi) = \mathsf{E}((\mathit{true})\,\mathsf{U}\,\phi)$
- $\mathsf{AG}(\phi) = \neg\,\mathsf{EF}(\neg\phi)$
- $\mathsf{AF}(\phi) = \neg\,\mathsf{EG}(\neg\phi)$
- $\mathsf{A}(\phi\,\mathsf{U}\,\psi) = \neg\,\mathsf{E}(\neg\psi\,\mathsf{U}(\neg\phi \wedge \neg\psi)) \wedge \neg\,\mathsf{EG}(\neg\psi)$
- $\mathsf{A}(\phi\,\mathsf{R}\,\psi) = \neg\,\mathsf{E}(\neg\phi U \neg\psi)$
- $\mathsf{E}(\phi\,\mathsf{R}\,\psi) = \neg\,\mathsf{A}(\neg\phi\,\mathsf{U}\,\neg\psi)$

Comparison of LTL and CTL

One might think that CTL has a higher expressiveness than LTL, but this is actually not true. LTL is neither a super- nor subset of CTL. Both logics offer different views of time [EH86]. There exist formulae of each language that cannot be expressed in the other language. For instance, the property "There is always the possibility that a state *reset* can be reached" can only be expressed with CTL using the formula $\mathsf{AG}(\mathsf{EF}(\mathit{reset}))$. LTL can only express that the reset state is always reached and not that it can be reached. $FG(p)$, however, is not expressible in CTL, but in LTL.

2.4.3 Counterexamples and Witnesses

One big advantage of model checking is the generation of counterexamples in case a formula is not valid for a given system model ($M \not\models \phi$). In the case of LTL, a counterexample is characterized by a finite trace of system states that indicates why $M \not\models \phi$. For example, a counterexample of the expression $\mathsf{G}\,\phi$ is a path $(s_0 s_1 ... s_n)$ where $s_i \models \phi$ for $0 \leq i < n$ and $s_n \not\models \phi$.

In the case of an CTL expression with a universal path quantification (e.g., AG, AF, or AX) the situation is quite similar. For example, the counterexample of $\mathsf{AX}\,\phi$ is characterized by a pair of states (s, s') where $s' \in \mathit{Post}(s)$ and $s' \not\models \phi$. However, for an CTL expression with an existentially quantified path formula such as $\mathsf{EF}\,\phi$ the situation is different because it makes no sense to create a counterexample if $M \not\models EF\phi$. $M \not\models \mathsf{EF}\,\phi$ means that no path satisfies ϕ and subsequently all the paths of the transition system would represent a counterexample. If $M \not\models \mathsf{EF}\,\phi$ the answer *No* is sufficient. For the answer *Yes* ($M \models \mathsf{EF}\,\phi$), a witness is created. A witness is a path that indicates the satisfaction of an existentially quantified path formula (e.g., EG, EF, or EX). That being said, depending on the operator, CTL model checking either delivers counterexamples or witnesses.

Most model checkers (e.g., NuSMV) stop the verification after the first counterexample is found even though there could exist more.

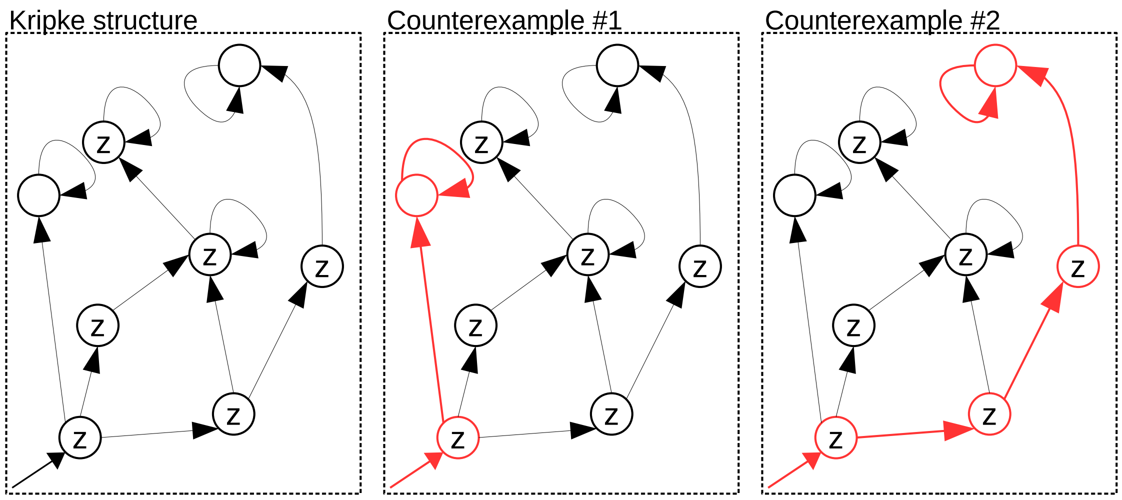


Figure 2.8: Counterexamples for $\mathsf{AG}(Z)$

Figure 2.8 shows a Kripke structure and all corresponding counterexamples for the CTL expression $\mathsf{AG}(Z)$. $AG(Z)$ denotes "On all paths, Z has to hold globally (in each state).". This specification is violated if there exists a path where somewhere Z does not hold. The Kripke structure contains two paths that violate the specification.

2.4.4 Model Checking Algorithm

After creating a model of the system M and specifying a safety property ϕ, the goal is to find out whether the system fulfills the property. In other words: $M \models \phi$?

For that an algorithm is needed that explores the states of the system systematically and checks whether each state satisfies the property. In principle, there are two categories of model checking algorithms namely explicit-state model checking and symbolic model checking.

2.4.5 Explicit-State CTL Model Checking

Explicit-state model checking algorithms are based on an explicit representation of the transition system, meaning that each state knows its predecessor and successor states. Basically, there are two possibilities for developing an explicit-state model-checking algorithm as described in [Fra+10].

(a) One way is to construct the transition system during the verification process, such as in the SPIN model checker [Hol97][1]. Meaning that the way in which the transition system is built up depends on the CTL formula. For example, if a state s is found that violates the specification, all subsequent states $s'_0, ...s'_n$ of s do not need to be explored. In case a cycle was found, the further search on the current path can be canceled as well. This approach has the advantage that the constructed

[1]The SPIN model checker accepts models described in Promela (Process Meta Language).

transition system consists only of paths that are actually relevant for verification. Conversely, this also means that the transition system must be reconstructed for each new specification.

(b) The other possibility is a two-step approach, meaning that the transition system is constructed prior to the verification. Thus first of all a transition system must be created, which in most cases is much larger than it is necessary for verification. On-the-fly model-checking algorithms stop the exploration on a path if the specification is violated in the current state, while algorithms based on a two-step approach compute the state space completely. Even though the search space might be too large, this approach has some significant advantages. Firstly, the decoupling leads to a clear separation between the model description and verification. Therefore, the model-checking algorithm is not limited to just one specific modeling language, because the verification needs only a clear interface to the transition system. Secondly, the complete transition system enables to verify different specifications against the system model without the need for re-simulation. Model checkers like CADP follow this approach [Gar+11].

A well-known algorithm for CTL model checking is shown in Algorithm 1. The algorithm is based on a recursive procedure that determines the satisfaction sets for all sub-formulae of the original CTL formula Φ in ENF. A satisfaction set is defined by the set of states $s \in S$ in which Φ holds. The calculation of satisfaction sets starts at the leaves in the parse tree, which are either the constant true or an atomic proposition $a \in AP$. For the calculation of satisfaction sets for intermediate nodes, the results of the sub-formulae are used. The result of $Sat(\Phi)$ contains **all** states in which Φ is fulfilled and not just the initial states. This is described as global model checking. In the end, if all initial states are included in the satisfaction set, the system meets the specification: $TS \models \Phi$ iff $I \subseteq Sat(\Phi)$

For example, for a formula in the form $\Phi = \Phi_1 \wedge \Phi_2$, first the satisfaction sets for $Sat(\Phi_1)$ and $Sat(\Phi_2)$ are calculated (recursive bottom-up computation). $Sat(\Phi 1)$ contains all states $s \in S$ in which Φ_1 holds and $Sat(\Phi 2)$ contains all states $s \in S$ in which Φ_2 holds. $Sat(\Phi)$ results from the states contained in both, $Sat(\Phi_1)$ and $Sat(\Phi_2)$: $Sat(Phi) = Sat(\Phi_1) \cap Sat(\Phi_2)$. If $I \subseteq Sat(\Phi)$, the system meets the specification.

Satisfaction sets implicitly also contain witnesses respectively counterexamples. For example, counterexamples for formulae of the form $\mathsf{AG}\,\Phi$ can be calculated by performing a backward search starting at states in which $\neg\Phi$ holds. The states in which $\neg\Phi$ holds can be determined using the satisfaction set $Sat(\Phi)$ in the following way: $Sat(\neg\Phi) = S \setminus Sat(\Phi)$. The backward search looks for path fragments $s_0, s_1 \ldots s_n$ where $s_i \models \Phi$ for $0 \leq i < n$ and $s_n \models \neg\Phi$.

For formulae of the form $\mathsf{A}\,\Phi\,\mathsf{U}\,\Psi$ counterexamples are calculated using the graph $G = (S, E)$ where

$$E = \{(s, s') \in S \times S \mid s' \in Post(s) \wedge s \models \Phi \wedge \neg\Psi\}$$

First, the SCCs in graph G are calculated. Finally, counterexamples are all paths in G starting in an initial state and leading to a terminal SCC.

Algorithm 1 Algorithm for computation of the satisfaction sets of a CTL formula Φ for a transitions system TS with state set S. [BK08]

```
function Sat($\Phi, TS$) returns the satisfaction sets
    switch $\Phi$ do
        case true : return S;
        case a : return $\{s \in S \mid a \in L(S)\}$;
        case $\Phi_1 \wedge \Phi_2$ : return $Sat(\Phi_1) \cap Sat(\Phi_2)$;
        case $\neg\Psi$ : return $S \setminus Sat(\Psi)$;
        case $\mathsf{EX}\,\Psi$ : return $\{s \in S \mid Post(s) \cap Sat(\Psi) \neq \emptyset\}$;
        case $\mathsf{E}(\Phi_1 \,\mathsf{U}\, \Phi_2)$ :
            T := Sat($\Phi_2$); // compute the smallest fix point
            while $\{s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \emptyset\} \neq \emptyset$ do
                let $s \in \{s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \emptyset\}$;
                $T := T \cup \{s\}$;
            end while
            return T;
        case $\mathsf{EG}\,\Phi$ :
            T := Sat($\Phi$); // compute the greatest fix point
            while $\{s \in T \mid Post(s) \cap T = \emptyset\} \neq \emptyset$ do
                let $s \in \{s \in T \mid Post(s) \cap T = \emptyset\}$;
                $T := T \setminus \{s\}$;
            end while
            return T;
end function
```

$Sat(\mathsf{EG}(\Phi))$ can be solved by using a backward search (see Algorithm 2). The principle of the algorithm is quite simple: Initially, the satisfaction set T of the expression Φ is calculated. For each state in T the number of successors is assigned to *count*. The counters are used to capture for each state the number of successors that satisfies Φ. The set E contains all states in which $Sat(\mathsf{EG}(\Phi))$ is not fulfilled. For each state $s' \in E$ it is checked whether the predecessor of s' is in the set T. If so, this means that this state has a successor that does not satisfy Φ. Therefore, the counter for this state must be decremented. If the counter for this state has the value 0, no successor satisfies Φ. This falsifies the semantic of EG and therefore this state is removed from the satisfaction set T.

One big challenge in explicit-state model checking is the state space explosion problem. Consider the following situation: A system with just 20 Boolean variables has $2^{20} =$ 1.048.576 states. If we add 5 variables with a domain of 6 potential values, the state space grows to $1.048.576 * 6^5 = 8.153.726.976$ states. With concurrent processes, the number of states grows even exponentially. This phenomenon is also known as the state explosion problem. Large state spaces can only be handled with efficient model-checking algorithms. Therefore, a set of optimization strategies in explicit-state model checking like partial order reduction or exploiting symmetry exist. With partial order reduction independent concurrent processes (given by transition systems) are represented only once [Pel98].

Two processes are defined as independent if the order execution does not influence the resulting system state. Using symmetries is a method that tries to reduce the state space by avoiding redundancies. Model checkers using these clever algorithms can handle state spaces up to 10^9 states [BK08]. Despite these optimization strategies, explicit-state model checking is applicable only for quite small models.

Algorithm 2 Enumerative backward search to compute $Sat(\mathsf{EG}\,\Phi)$ [BK08]

```
function Sat($\mathsf{EG}\,\Phi, TS$) returns the satisfaction sets
    $E := S \setminus Sat(\Phi)$;
    $T := Sat(\Phi)$;
    for all $s \in Sat(\Phi)$ do
        $count[s] := |Post(s)|$;
    end for
    while $E \neq \emptyset$ do
        let $s' \in E$;
        $E := E \setminus \{s'\}$;
        for all $s \in Pre(s')$ do
            if $s \in T$ then
                $count[s] := count[s] - 1$;
                if $count[s] = 0$ then
                    $T := T \setminus \{s\}$;
                    $E := E \cup \{s\}$;
                end if
            end if
        end for
    end while
    return $T$;
end function
```

2.4.6 Symbolic Model Checking

In symbolic model checking the transition system is not represented explicitly, but rather symbolically using boolean formulae or binary decision diagrams (BDD)[Ake78]. The advantage of this is that the state explosion can be avoided (for most systems). Symbolic model checking, however, requires a symbolic representation of the state space, transition relation, and specification. Furthermore, algorithms must work on symbolic representation

A binary decision diagram (BDD) is a data structure based on directed acyclic graphs for the representation of boolean formula f in an efficient way. The diagram is created in a way that each row in the truth table of f has a correspondent path in the diagram from the root to a terminal node. Terminal nodes are labeled with the corresponding outcome of f (*0* or *1*). Each other node k is labeled with a binary variable and has exactly two successors: $lo(k)$, if $x_i = 0$ and $hi(k)$, if $x_i = 1$. Figure 2.9 shows a boolean formula represented using a BDD. A formula with n variables is represented by a directed graph with $2^{n+1} - 1$ nodes and 2^n paths.

$f = (x_1 \wedge x_2) \vee x_3$			
x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

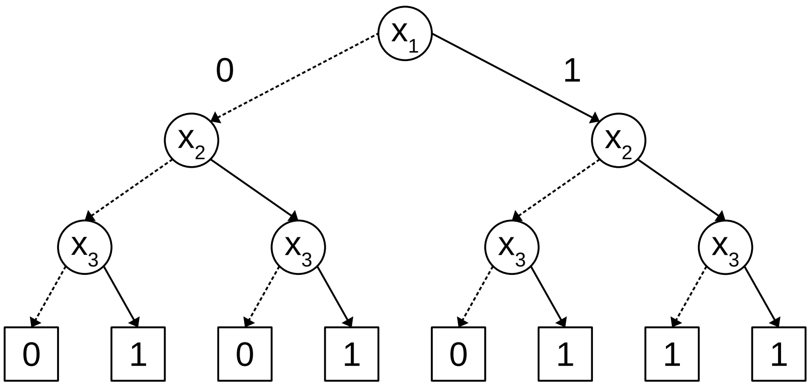


Figure 2.9: Representation of a Boolean formula using a BDD

Bryant introduced in [Bry86] two concepts for a more efficient representation of BDDs. The first concept is based on the order of the variables in the BDD. A BBD is called ordered (OBDD) if the variables appear on all paths in the same order. The order of the variables in BDDs has a high impact on the number of required nodes for representing a boolean formula. Therefore, for huge systems, it is very important to find the optimal variable order. Finding the optimal variable order is NP-complete. However, there exist various heuristic approaches. The second optimization technique reduces the size of OBDDs by transformation. [Bry86] describes a transformation rule called elimination that is based on removing duplicate nodes and subgraphs (reduced OBDDs). Figure 2.10 shows the reduced OBDD (variable order $x_1 < x_2 < x_3$) of the BDD introduced in Figure 2.9.

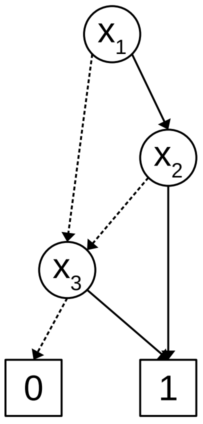
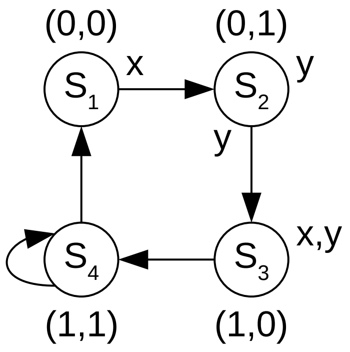


Figure 2.10: Reduced OBDD of the BDD shown in Figure 2.9

The main idea in symbolic model checking is to use OBDDs for representing the Kripke structure. To describe a transition system using an OBBD, a suitable representation of the states is required. The solution is to represent a state by a boolean vector. This enables a system to be represented by OBDDs. Consider the Kripke structure shown in Figure 2.11. A vector with two components (x_2, x_1) is sufficient to represent this Kripke structure with its four states. For example, state S_2 is represented by the vector $(0, 1)$. State transitions can be represented as a pair of states $((x_2, x_1), (x'_2, x'_1))$. For example, the state transition from s_2 to s_3 is described by $((0, 1), (1, 0))$. The labeling functions are described in the same way.



Explicit representation

$$\begin{aligned} M =& (S, S_0, R, L) \\ AP =& \{x, y, z\} \\ S =& \{S_1, S_2, S_3, S_4\} \\ S_0 =& \{S_1\} \\ R =& \{(S_1, S_2), (S_2, S_3), (S_3, S_4), \\ & (S_4, S_4), (S_4, S_1)\} \\ L(S_1) =& \{x\} \qquad L(S_2) = \{y\} \\ L(S_3) =& \{x, y\} \qquad L(S_4) = \{\emptyset\} \end{aligned}$$

Symbolic representation

$$\begin{aligned} S_1 \equiv& (0, 0) S_2 \equiv (0, 1) \\ S_3 \equiv& (1, 0) S_4 \equiv (1, 1) \\ R =& \{((0, 0), (0, 1)), ((0, 1), (1, 0)), \\ & ((1, 0), (1, 1)), ((1, 1), (1, 1)), \\ & ((1, 1), (0, 0))\} \\ L(x) =& \{(0, 0), (1, 0)\} \\ L(y) =& \{(0, 1), (1, 0)\} \end{aligned}$$

Figure 2.11: Symbolic representation of a Kripke structure

The transition relation R of the Kripke structure can be expressed with the following boolean formula:

$$\begin{aligned} R =& (((\neg x_2 \wedge \neg x_1) \wedge (\neg x'_2 \wedge x'_1)) \vee ((\neg x_2 \wedge x_1) \wedge (x'_2 \wedge \neg x'_1)) \vee ((x_2 \wedge \neg x_1) \wedge (x'_2 \wedge x'_1)) \vee \\ & ((x_2 \wedge x_1) \wedge (x'_2 \wedge x'_1)) \vee ((x_2 \wedge x_1) \wedge (\neg x'_2 \wedge \neg x'_1))) \end{aligned}$$

This boolean formula can be represented as an OBDD as shown in Figure 2.12. The question arises, how this efficient representation of a transition system can be used to verify a CTL formula? OBDDs provide a set of efficient operations such as *Apply* or *AndExist*. In the worst case, the OBDD operations have an exponential run-time, which, however, hardly occurs in practice. The function *Apply* connects two boolean functions with any boolean operator while *AndExists* performs an existential quantification over the conjunction of two OBDDs. Symbolic model-checking algorithms are based on these functions. The input of the evaluation function is a CTL formula, and the result of the verification is an OBDD that represents the states in which the specification is fulfilled. Thanks to these efficient algorithms systems with 10^{20} states and more can be handled. Note that in the area of asynchronous systems and communication systems, explicit-state model checkers outperform symbolic model checkers [Don+99].

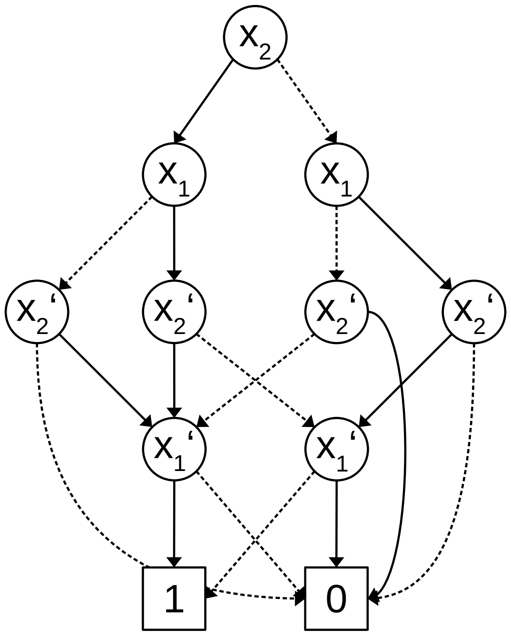


Figure 2.12: Transition relation R described by an OBDD

2.5 Requirements to Real-Time Systems

Over the last decades, several formalisms for safety requirements specifications for real-time systems have been developed. The most popular logic in this context are probably *Metric Temporal Logic* (MTL), *Timed Computation Temporal Logic* (TCTL), and *Real-Time Computation Tree Logic* (RTCTL).

2.5.1 Metric Temporal Logic

MTL is an extension of LTL (Linear Temporal Logic) in which the temporal operators G (**G**lobally), F (**F**inally), and U (**U**ntil) are extended by timing constraints [Koy90]. The extension of LTL operators leads to the following syntax:

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \mathrm{U}_I \varphi$$

where I is an interval with integer endpoints that defines the time interval for the formula. MTL formula can be interpreted over both, discrete (e.g., $\mathbb{N}$) and dense (e.g., $\mathbb{R}$) time domains. These extended operators enable the specification of different temporal properties as shown by the following practical examples:

- G(brakePedalPressed $\rightarrow \mathsf{F}_{(0,5)}$ brakingResponse) defines a **maximal distance** of 5 time units between pressing the pedal and the brake operation.
- G(crash $\rightarrow \mathsf{F}_{\{5\}}$ airbagActivated) specifies a **distance of exact** 5 time units between a crash situation and the airbag activation.
- G(failure $\rightarrow \neg\,\mathsf{F}_{(0,20)}$ failure) defines a **minimal distance** of 20 time units between two failure situations.
- $\mathsf{F}_{\{5\}}$(selfCheck) $\wedge$ G(selfCheck $\rightarrow$ ($\neg$selfCheck $\mathsf{U}_{\{5\}}$ selfCheck) states that every 5 time units a self-check is performed (**periodically**).

There exist different semantics for timed logic like MTL. The two most common semantics for MTL are the point-based semantic and the interval-based (continuous) semantic [OW08].

2.5.2 Timed Computation Tree Logic

Timed Computation Tree Logic (TCTL) is an extension of CTL where path quantifiers are extended with clock constraints to specify properties for timed automata. Let C be a set of clocks, the syntax of a TCTL state formula is created by

$$\phi ::= \top \mid a \mid g \mid \neg\phi \mid \phi \wedge \phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ is an atomic proposition and $g \in ACC(C)$ is an atomic clock constraint over the clocks in C. The path formula ϕ extended with clock constraints is defined by $\varphi ::= \phi U^J \phi$. $J \subseteq \mathbb{R}_{\geq 0}$ is an interval with natural numbers that specifies the clock constraints over the clocks in C.

Consider the following property to the traffic light example described in Section 2.3.3: "The red light will stay on for exactly 60 seconds and then switch to green". This property can be expressed in TCTL with the following formula:

$$\mathsf{AG}(s == red \wedge (x = 0) \rightarrow (\mathsf{AG}^{\leq 60}\, s == red \wedge \mathsf{AF}^{>60}\, s == green)$$

2.5.3 Real-Time Computation Tree Logic

Expressions like "φ must be fulfilled within the next 3 successor states" can be specified using the next-operator: $\mathsf{EX}(\varphi \vee \mathsf{EX}(\varphi \vee \mathsf{EX}(\varphi)))$. Of course, this is only feasible for shorter periods of time. RTCTL (Real-Time Computation Tree Logic) [Eme+92] addresses this problem by introducing special operators:

- ABG $i..j\ \Phi$: For all paths Φ holds at a each point in time k, with $i \leq k \leq j$

- ABF $i..j\,\Phi$: For all paths Φ holds at some point in time k, with $i \leq k \leq j$
- EBG $i..j\,\Phi$: There is at least one path on which Φ holds at a each point in time k, with $i \leq k \leq j$
- EBF $i..j\,\Phi$: There is at least one path on which Φ holds at some point in time k, with $i \leq k \leq j$

For example, the RTCTL expression ABF0..5φ states that within the next 5 steps a state s is reached where $s \models \varphi$.

2.6 Strongly Connected Components in Directed Graphs

Directed graphs are special graphs where the edges have a direction [Die00]. Formally, a directed graph $G = (V, A)$ consists of

- a set of vertices V and
- a set of edges E

An important property of directed graphs for this work is called *strongly connected*. A directed graph G is strongly connected if there exists a path from n_1 to n_2 and a path from n_2 to n_1 for each pair of different nodes $(n_1, n_2) \in V^2$. Strongly connected components are subgraphs of a directed graph that are strongly connected. Tarjan's algorithm (see Algorithm 3) determines all strongly connected components in directed graphs in linear time.

Algorithm 3 Tarjan's algorithm for determination of strongly connected components[Tar71]

```
    Input directed graph $G = (V, E)$
    Output set of strongly connected components (sets of vertices)
 1: result = {};
 2: index = 0;
 3: U = V;
 4: S = {};
 5: for each v ∈ U do
 6:     tarjan(v);
 7: end for
 8:
 9: function tarjan(v)
10:     v.index = index;
11:     v.lowlink = index;
12:     index = index + 1;
13:     S.push(v);
14:     U = U \ {v};
15:     for each (v, w) ∈ E do
16:         if w ∈ U then
17:             tarjan(w)
18:             v.lowlink = min(v.lowlink, w.lowlink);
19:         else if w ∈ S then
20:             v.lowlink = min(v.lowlink, w.index);
21:         end if
22:     end for
23:     if v.lowlink == v.index then
24:         scc = {};
25:         repeat
26:             scc = scc ∪ S.pop;
27:         until w == v
28:         result = result ∪ scc;
29:     end if
30: end function
```

3 Related Work

Current approaches to MBSA differ in many aspects. In [LKN11] Lisagor et al. defined two important distinguishing criteria by which approaches to MBSA can be differentiated: The first criterion introduced by Lisagor et al. concerns the semantics of component interfaces. In principle, there are two approaches, namely failure effect modeling (FEM) and failure logic modeling (FLM). The idea of FLM is to model only failures and how they are propagated through the components in a system. In other words, components only share information about deviations from intended behavior (e.g., commission or omission of a signal). Components can react to failures, repair them, or simply pass them on to other connected components. In FEM the nominal behavior is extended with failure behavior. Component failures, for example, may affect the output signal of a component.

The second criterion introduced by Lisagor et al. relates to model construction. Standalone safety assessment models are, as the name already implies, system models that are created just for the purpose of safety analysis. Such models incorporate only that knowledge that is required for safety analysis. Thus, complexity can be reduced significantly. The other concept is called failure injection (FI). In failure injection already existing system models are extended with fault models. This allows designers and safety engineers to work on the same model.

Beyond that, two additional criteria are introduced in [HL14]. One essential aspect is the level of abstraction of the system models. Quantitative approaches (e.g., Simscape) will obviously produce considerable detailed results than qualitative approaches. At the same time, however, the computational effort is much higher. The level of abstraction used by the modeling formalism has a high impact on both, the expressiveness of the models and the computational effort.

Another distinguishing criterion is the type of connection modeling. Connections between components can be directional or bidirectional. Directed connections can lead to problems in failure situations where the cause-effect relationship reverses. With directed connections such situations can only be handled by creating artificial connections, which obviously increases model complexity. Bidirectional connections keep the model structure close to reality, especially if physical interactions are essential in system behavior.

Several approaches to model-based safety analysis exist already. This chapter presents some relevant approaches to MBSA that are particularly applicable for automated safety analysis of technical systems in the early stages of product development. At the end of the chapter, the approaches are classified according to the criteria presented above.

3.1 Qualitative Order of Magnitude Reasoning

Neal Snooke and Mark Lee present in [SL14] an energy-flow-based qualitative modeling approach based on order of magnitudes for automated FMEA. The approach is premised on a two-level modeling strategy. On the lower level, a qualitative algorithm is used to derive flow and effort values in a resistance network. Depending on these values, state transitions are performed which affect the component's behavior.

3.1.1 Flow Direction Determination

The power network is represented by a Graph $G(T, A)$ with a set of edges A and nodes T. Each edge $e \in A$ is assigned a resistance value $R(e)$ and each edge always connects two nodes. Resistance values are reduced to their relative order of magnitude (OM). The authors introduced the notation $q^{>n}$ to specify a value that is **n** orders of magnitude **below reference value q**. Conversely, $q^{<n}$ indicates **n** orders of magnitude **above reference value q**. Negative order of magnitude values can be converted into a positive representation and vice versa ($q^{>n} = q^{<(-n)}$). $q^{<x+1}$ dominates $q^{<x}$ for all $x > 0$. To make the approach applicable to different domains, a generalized version of Ohm's law is applied. That means, that the simulation task is not to determine exact current or voltage values but rather to calculate the flow and effort for each edge. Flow is measured through an edge $F(e)$, and the effort is measured between two nodes $E(t1, t2)$. Flow and effort values are also expressed with the order of magnitude representation. Table 3.1 shows the relationship between effort, resistance, and flow.

Effort, E	Resistance, R	Flow, F
0	0	? (qualitative ambiguity)
0	r	0
0	∞	0
u	0	$\sqcup$ (impossible case)
u	r	f
u	∞	0

Table 3.1: Qualitative current assignment

To calculate the flows and efforts in a system, first of all, the power network is reduced to a single equivalent resistor. Subsequently, the flow through the equivalent resistor is calculated. The calculated flow value is then used to calculate the flow and effort values on the upper levels by successively expanding the network back to the original state.

Network Reduction

The network is reduced by series and parallel circuit simplifications. For non-serial or parallel reducible networks an order of magnitude version of the qualitative star-delta ($Y - \Delta$) transformation is used. The following reduction rules are used:

Two edges e_1, e_2 in series $(e_1 \,\vdots\, e_2)$ are replaced by an equivalent edge where

$$R(e_1 \,\vdots\, e_2) = max(R(e_1), R(e_2)) \tag{3.1}$$

Parallel edges $(e_1 \,||\, e_2)$ are replaced by an equivalent edge where

$$R(e_1 \,||\, e_2 \,||\, ... \,||\, e_n) = min(R(e_1), R(e_2), ..., R(e_n)) \tag{3.2}$$

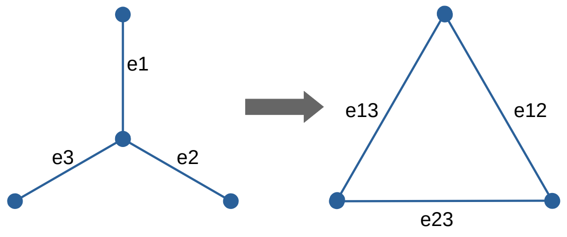


Figure 3.1: $Y - \Delta$ transformation

Nonserial or parallel reducible networks are transformed in such a way that the result is serial/parallel reducible (see Figure 3.1). This is achieved by replacing all edges $e_1 ... e_n$ that are connected to a non-serial and parallel reducible star of degree n by new edges e_{jk} where $1 \leq j \leq n, 1 \leq k \leq n, 1 \leq m \leq n, k > j$ and

$$R(e_{jk}) = \frac{R(e_j)R(e_k)}{min(R(e_m))} \tag{3.3}$$

The multiplication is defined as follows:

$$r^{>n} * r^{>m} = r^{>(n+m)} \tag{3.4}$$

Conversely, the division is defined as follows:

$$r^{>n} / r^{>m} = r^{>(n-m)} \tag{3.5}$$

Flow Assignment

The approach utilizes a generalized version of Ohm's law $(E = F * R)$. In general, the flow is always identical for edges that are part of a series.

$$F(e_1) = F(e_2) = F(e_1 \,\vdots\, e_2) \tag{3.6}$$

For parallel edges, the effort is equal:

$$E(e_1) = E(e_2) = E(e_1 \,||\, e_2) \tag{3.7}$$

The effort at an edge, given the flow and resistance, can be calculated using the following formula:

$$E(e_1) = F(e_1)R(e_1) \tag{3.8}$$

For edges of a star-delta transformation, we have to use the following formula (for a n star node: $1 \leq j \leq n, 1 \leq k \leq n, k > j$):

$$F_{e_k} = \sum_{m=1}^{k-1} F(e_{mk}) - \sum_{m=k+1}^{n} F(e_{km}) \tag{3.9}$$

The rules for order of magnitude addition are listed in Table 3.2.

$+$	0	$+p^{>n}$	$-p^{>n}$	∞	?
0	0	$+p^{>n}$	$-p^{>n}$	∞	?
$+p^{>m}$	$+p^{>m}$	$+p^{>min(m,n)}$	$-p^{>n}$ if $n < m$ $+p^{>m}$ if $m < n$? if n == m	∞	?
$-p^{>m}$	$-p^{>m}$	$+p^{>n}$ if $n < m$ $-p^{>m}$ if $m < n$? if n == m	$-p^{>n}$	∞	?
∞	∞	∞	∞	∞	?
?	?	?	?	?	?

Table 3.2: Rules for OM addition

Figure 3.2 shows an example for the reduction and assignment steps for flow calculation in a simple resistance network. For reduction, the parallel resistors *R2* and *R3* are replaced by an equivalent resistor with value $r^{<2}$ first. The remaining resistors in series are replaced by an equivalent resistor with value $r^{<4}$ (R4 dominates). Now the flow through the single remaining resistor can be calculated using formula 3.7: $F = e^{<0}/r^{<4} = f^{<(0-4)} = f^{>4}$. In step *5* the series reduction is expanded and the flow is assigned to each individual edge which is simply the flow through the reduced resistor. Finally, the flow through the parallel resistors can be determined. For this purpose the effort across the resistors needs to be calculated first using formula 3.7: $E = f^{>4} * r^{<2} = e^{<(-4+2)} = e^{>2}$. The flow through resistors *R2* and *R3* can be calculated using the effort and resistance value. For example, the flow through *R2* is $e^{>2}/r^{<2} = f^{<(2-(-2))} = f^{>4}$. The flow through *R2* is one order of magnitude higher compared with the flow through *R3*. This is plausible because *R3 > R2*.

The approach also supports multiple effort or flow sources by utilizing the principle of superposition. This means that each network is analyzed separately for each source $s_1 ... s_n$. The flow results from the maximum of all flow analysis steps:

$$\sum_{s_1}^{s_n} F(e) = max(F_{s_1}(e) ... F_{s_n}(e))$$

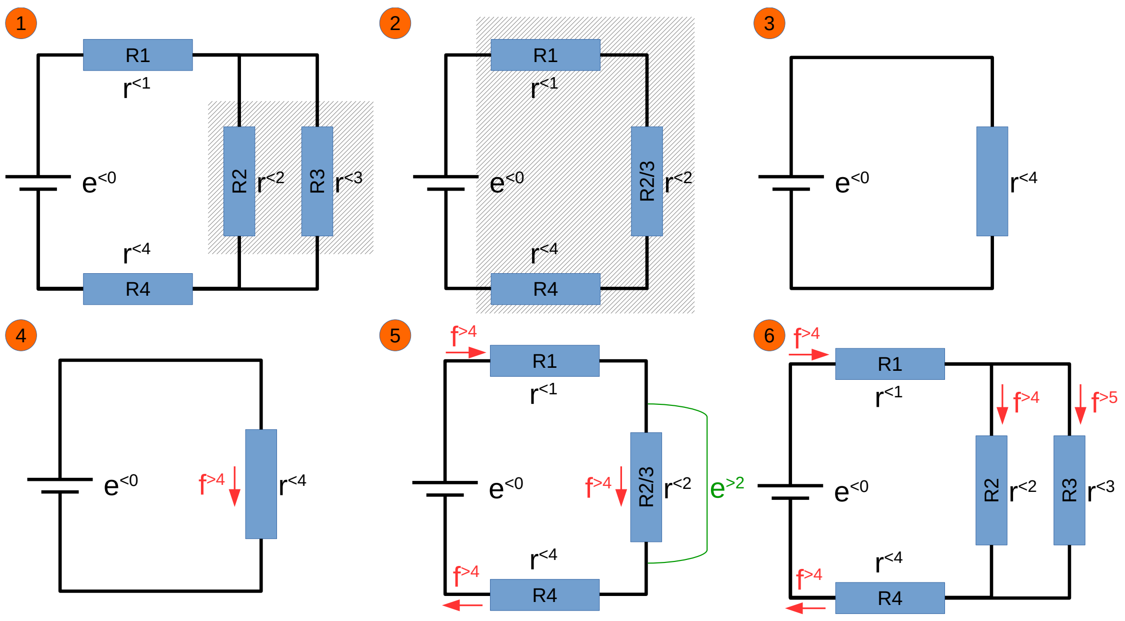


Figure 3.2: Qualitative flow calculation in a simple resistance network

3.1.2 Component Behavior

On the second layer, the component behavior is described in terms of finite-state machines. State transitions are triggered by flow and effort changes in the power network. In addition, there are also input events to initiate state changes from outside the system. State variable changes may affect the structure or the resistance value of connections. The behavioral model is separated from the structural model. Both are described in a graphical notation. A special subset of the UML state diagrams is used to describe the finite state machines.

Figure 3.3 shows the component model of a simple tank with a vent and an inlet. The behavioral model consists of two states namely **empty** and **full**. There are state entry actions that are executed when entering a state. For example, when entering the state **empty**, the level is set to **0**, and information about the substance is propagated. Event conditions are used to describe the situations in which transitions are triggered. In the tank example, a flow of substance fluid into the tank will cause a state change from **empty** to **full** after some time. This event is called **filling**. The time at which the transition is triggered depends on the local variable *volume* of the tank and the order of magnitude of flowing water into the tank. A tank with a volume value of order of magnitude < 3, for instance, takes three times longer to be filled in comparison to a tank with a volume value of order of magnitude < 1.

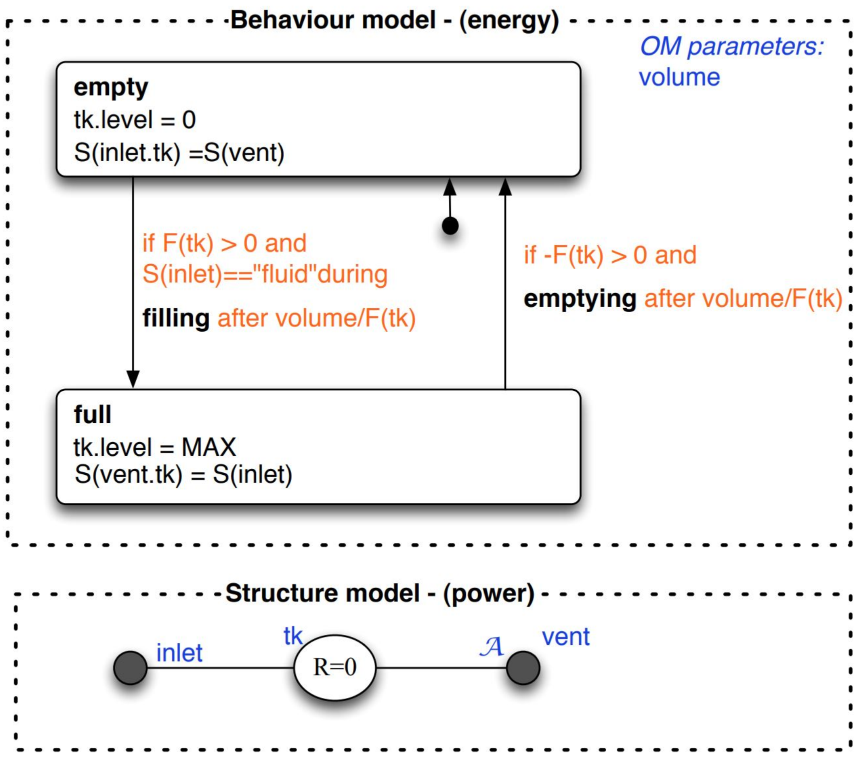


Figure 3.3: Component model of a tank (taken from [SL14])

3.1.3 Simulation and FMEA Generation

The simulation is based on a time-ordered priority list that contains all component events whose event conditions are satisfied. The events in the queue are ordered by their time delay. Event conditions with minimal delay are triggered first. Event conditions that are no longer fulfilled are removed from the queue. If the queue is empty, a steady state has been reached and the simulation is finished.

The models can be used for an automated FMEA generation. In principle first the system is simulated without failures and after that, each possible component failure is simulated. The result of a simulation run are the values of all system variables in each state of the simulation. A separate model is used to assign each system state a functional state which can be *Achieved, Failed, Unexpected Behavior, or Inoperative.* To find the effects of a component failure, the results of the nominal and failure behavior need to be compared.

3.2 HiP-HOPS

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) is a method for automated safety analysis using an extremely high level of abstraction [PM99]. HiP-

HOPS aims to simplify the analysis of complex hierarchical systems, such as the generation of Fault Trees, and to provide a mechanism that leads to consistent results. The idea originates from traditional analysis tasks like FMEA and FFA (Functional Failure Analysis) and Failure Propagation and Transformation Notation (FPTN).

The analysis process in HiP-HOPS is comprised of two steps. First, a functional failure analysis is performed to identify possible faults (single or multiple faults). This is helpful to get a first impression of the critical components of the systems. After that, the dependencies between the components are analyzed. This is an important step since component failures may not only be caused by internal or external faults but also by deviations at the input caused by other components. In other words, components can generate, detect, or respond to failures. In HiP-HOPS this step is called IF-FMEA (Interface Focused-FMEA). For each component output, it is described how a deviation can occur in dependence on internal faults or deviations at the inputs.

The result of an IF-FMEA is a table where each row describes the failure logic to each possible output deviation. The definition of input/output deviation consists of two parts. The first part is the failure class. As this can be anything, meaning that there are no limitations or predefined classes. The second part is the name of the port. Failure logic is described using boolean expressions consisting of input deviation or internal failure combined with the logical operators *AND* or *OR*. Table 3.3 shows an example of component characterization in HiP-HOPS.

Output Deviation	**Description**	**Failure logic**
OmissionFlow-Out	Omission of flow of fluid at the output of Valve1 can be caused either by omission of flow at the input, by a failure of Valve1 or by omission of the command at input	*OmissionFlow-In or Omission-Cmd or ValveFailure*
Commission-Out	Commission of flow of fluid at the output of Valve1 can be caused either by commission of command at input	*Commission-Cmd or ComponentFailure*
...	...	...

Table 3.3: Component characterization in HiP-HOPS

These component failure models can then be used for generating Fault Trees. In HiP-HOPS the algorithm for Fault Tree generation is called Fault Tree Synthesis (FTS). This algorithm traverses the hierarchical model from a hazard back to the basic failure by following the propagation of failures in the reverse direction.

Models in HiP-HOPS can be created with Matlab Simulink [PM01] or the simulation tool SimulationX [1]. In the case of Simulink a tool that enables annotation of blocks (e.g., simple subsystems with a failure model) with a failure model is used.

[1] `https://www.simulationx.com/simulation-software/beginners/safety-designer.html`

3.3 AltaRica

AltaRica is a formal language for automation of safety-related tasks developed by the computer science laboratory of Bordeaux (Laboratoire Bordelais de Recherche en Informatique) at the end of the nineties [GA99]. The modeling language is component-oriented and supports hierarchical models. In AltaRica each component in a system is represented by a node. A node consists of a set of variables, events, transitions, and assertions. There are two types of variables namely flow variables and state variables. Flow variables are shared variables used for communication between components. In contrast, state variables are visible only inside components. State variables are used to describe the internal state of components. Both, flow and state variables are typed (e.g., the types Boolean or Integer are supported). The explicitly defined (external) events cause state changes. The reaction to these events is described by transitions. A transition consists of the name of the event that induces the state change, a guard expression, and a list of state variable assignments. The guard expression is a boolean formula that must be true to trigger the transition. If the guard expression is not fulfilled, the transitions will not be executed. AltaRica is an asynchronous language which means that only one transition can be fired at a time. However, there exists a synchronization mechanism to trigger multiple transitions simultaneously. Assertions describe the assignment of values to flow variables depending on the state of the node. The following listing shows the model of a simple electrical switch in AltaRica:

```
node Switch
  flow
    p1,p2 : bool;

  event
    Open, Close, Failure;

  state
    open : bool;
    ok : bool;

  init
    open := true;
    ok := true;

  trans
    not open and ok |- Open -> open := true;
    open and ok |- Close -> open := false;
    ok |- Failure -> ok := false;

  assert
    (not open) => (p1=p2);
edon
```

Listing 3.1: Model of an electrical switch the in AltaRica language

In this example, electrical flow is represented by a simple boolean signal. Either there is flow (value true) or not (value false).

AltaRica supports bidirectional flows between components. A constraint solver is used to update the value of the flow variables depending on the current state of the system (state variables and their values). Models with loops are not supported. Moreover, the approach based on the constraint solver is rather inefficient and is therefore not applicable for industrially relevant systems [Bat+13].

AltaRica Data-Flow [Boi+06] has been developed to address these problems. The first version of AltaRica version allowed bidirectional flows. To make the calculations more efficient, AltaRica Data-Flow restricts the flow variables to a specific direction (input or output). Flow variable values are updated by propagation in a fixed order (the direction of propagation is determined at compile time). So-called transfer-functions describe the output values depending on the values of the inputs. This reduces the computational effort significantly. However, cyclic system models are still not supported. The syntax of AltaRica Data-Flow has significantly changed over time. Among other changes, the latest version of AltaRica Data-Flow describes the components by classes instead of nodes, and declarations of transitions have changed. The AltaRica Data-Flow language is used by many tools such as Cecilia OCAS or Dassault Systems Safety Designer. The lack of support for looped systems and the limitation to directed flow variables was the motivation for an improved version.

The latest version, **AltaRica 3.0** [Bat+13], changed in two aspects. On the one hand, new concepts were added to the language (e.g., systems can also be modeled using *blocks* for prototyping). On the other hand, the mathematical model was changed to Guarded Transition Systems (GTS) [Rau08]. Formally, a Guarded Transition System is the tuple (V, E, T, A, ι), where:

- V is a set of flow variables (F) and state variables (S)
- E is a set of symbols for events
- T is a set of transitions
- A is a set of instructions over F, called assertions
- ι is the initial variables assignment

A transition consists of an event $e \in E$, a boolean formula built over V (= guard condition), and a set of actions. The guard decides whether a transition is fireable or not. A transition can only be fired in a state if the guard condition is fulfilled. The actions of a transition are instructions over S, i.e., variable value assignments. Optionally, events can be synchronized as in the previous versions. Flow variables in AltaRica 3.0 allow flow in both directions. Assertions describe the value of flow variables depending on the current state. Assertions are executed immediately after the transitions are fired. During runtime, it is decided for each component which flow variables operate as inputs/outputs. For assertions, there are operators for bidirectional (`:=:`) and unidirectional (`:=`) assignments between flow variables.

The compilation into Fault Trees starts with a flattening function that converts a hierarchical model into a single GTS. During this step, bidirectional assignments are converted into two unidirectional assignments, and for synchronized events, a simple transition is created. Thereafter, the flattened GTS is partitioned into independent parts using Tarjan's algorithm (described in Algorithm 3) and the corresponding reachability graph is calculated. A reachability graph is a Kripke structure, i.e., a graph in which nodes correspond to variables and edges are labeled with event names. The graph is an explicit representation of all reachable states of a GTS (the GTS implicitly describes the states by the state variables). A reachability graph can be obtained from a GTS in the following way: Initially, all variables are set to their default/reset value. Subsequently, the fireable transitions are determined and executed resulting in a set of new states (=nodes). For each new state, flow variables are updated using the assertions, and transitions are executed. This procedure is repeated until a fixpoint is reached, i.e., no more node or vertex can be added to the graph [BPR17]. Flow variable updates are also based on fix point calculations. After the calculation of the reachability graphs, each reachability graph is converted into boolean equations. The set of boolean equations already represents the set of Fault Trees of the system [PR14]. Top events can be defined in the model by using observers. An observer is a simple boolean expression that quantifies a property to be observed. Thanks to this new technique looped systems as well as bidirectional flows can be handled.

Example 3.3.1 A simple source-valve-consumer system
Listing 3.2 shows the AltaRica 3.0 model of a simple system consisting of a source, a controllable valve, and a consumer. The consumer is only supplied if the valve is opened. Figure 3.4 shows the corresponding reachability graph.

```
block SimpleSystem

  block Valve
    Boolean closed (init = false);
    Boolean working (init = true);
    Boolean in (reset = false);
    Boolean out (reset = false);
    event Open;
    event Close;
    event Failure;

    transition
      Open: working and closed -> closed := false;
      Close: working and not closed -> closed := true;
      Failure: working -> working := false;

    assertion
      if not closed then out :=: in;
  end

  block Source
    Boolean out (reset = true);

    assertion
      out := true;
  end
```

```

    block Consumer
        Boolean in (reset = false);
    end

    assertion
        Source.out :=: Valve.in;
        Valve.out :=: Consumer.in;

    observer Boolean ConsumerNotSupported = not Consumer.in;
end
```

Listing 3.2: A simple source-valve-consumer system in AltaRica 3.0. The implementation of the valve is taken from [Bat+13].

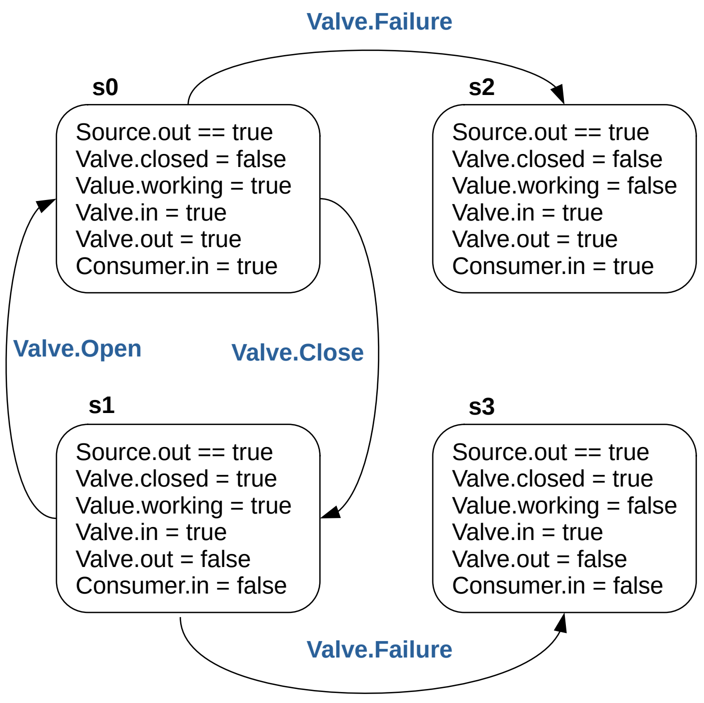


Figure 3.4: Reachability graph of the source-valve-consumer system

(a) The boolean equations for each node in the reachability graph in Figure 3.4 are as follows:

- $\phi_{s_0} = \overline{Open} \wedge \overline{Close} \wedge \overline{Failure}$
- $\phi_{s_1} = \overline{Open} \wedge Close \wedge \overline{Failure}$
- $\phi_{s_2} = \overline{Open} \wedge \overline{Close} \wedge Failure$
- $\phi_{s_3} = \overline{Open} \wedge Close \wedge Failure$

(b) The boolean equations for each variable and its value are as follows:

$\phi_{(Source.out,\text{true})} = \phi_{s_0} \vee \phi_{s_1} \vee \phi_{s_2} \vee \phi_{s_3}$	$\phi_{(Valve.closed,\text{true})} = \phi_{s_1} \vee \phi_{s_3}$
$\phi_{(Valve.working,\text{true})} = \phi_{s_0} \vee \phi_{s_1}$	$\phi_{(Valve.in,\text{true})} = \phi_{s_0} \vee \phi_{s_1} \vee \phi_{s_2} \vee \phi_{s_3}$
$\phi_{(Valve.out,\text{true})} = \phi_{s_0} \vee \phi_{s_2}$	$\phi_{(Valve.out,\text{false})} = \phi_{s_1} \vee \phi_{s_3}$
$\phi_{(Consumer.in,\text{true})} = \phi_{s_0} \vee \phi_{s_2}$	$\phi_{(Consumer.in,\text{false})} = \phi_{s_1} \vee \phi_{s_3}$
...	...

The Fault Tree for the top event “Consumer is not supplied”, defined by the observer `ConsumerNotSupported` (see observer declaration in line 36) can be constructed quite simply using boolean equations in **(a)** and **(b)**. It follows from **(b)** that the observer property is violated in ϕ_{s_1} and ϕ_{s_3}. It follows from boolean equations **(a)** that ϕ_{s_1} is reached if the valve is closed (by event *Close*) and ϕ_{s_3} is reached by the events *Close* and *Failure*. ■

Another concept added to AltaRica 3.0 is timed GTS. Events can be assigned deterministic or stochastic delays. Delays are used to determine the firing date of an event. There exists a so-called oracle that determines the sequence of events that are allowed to be triggered based on the delays [Pro14]. Note that in AltaRica 3.0 only one transition is executed at once. Therefore, there is an expectation value that can be assigned to events. This weight is used in situations when there exist several transitions that can be triggered at the same time.

To conclude, Table 3.4 shows an overview of the successive versions of AltaRica.

Variant	**AltaRica**	**AltaRica Data-Flow**	**AltaRica 3.0**
Mathematical	Constraint Automata	Mode Automata	GTS
Flow propagation method	Constraint solving	Propagation	Fix point
Efficiency	Low	High	High
Looped system supported?	No	No	Yes
Bidirectional flows?	Yes	No	Yes
Object-oriented?	Yes	Yes	Yes
Prototype-oriented?	No	No	Yes
Available tools	AltaRica Studio	Cecilia OCAS	OpenAltaRica

Table 3.4: Overview of the AltaRica versions

3.4 Approaches based on Formal Verification

Some approaches to automated safety analysis utilize formal verification tools. These approaches follow the principle of failure-injection.

3.4.1 Approach by Joshi et al.

The approach proposed by Joshi et al. is based on Mathworks Simulink and the model checking tool NuSMV [JWH][Jos+05]. The safety assessment starts with creating a model of the nominal system behavior in Simulink. After that, this model is extended with a fault model. The fault model is directly integrated into the nominal system behavior and created using Simulink blocks.

Example 3.4.1 Fault model of a stuck-at failure
Figure 3.5 shows a possible implementation of a fault model of a stuck-at failure. The

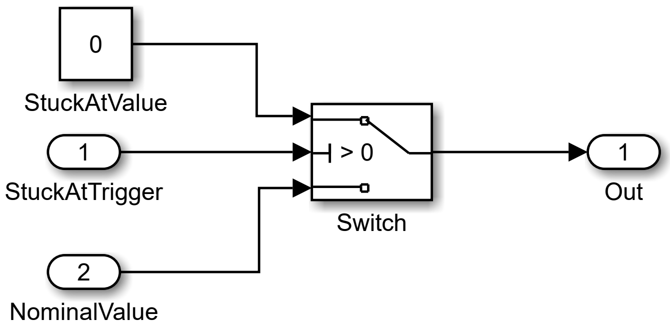


Figure 3.5: Fault modeling in Simulink

failure behavior is defined in a subsystem with two input ports and one output port. The input port `StuckAtTrigger` is used to activate the failure behavior. Depending on the value at this port, the output value is either constant (e.g., constant 0) or defined by the input `NominalValue`. This subsystem can, for example, be connected to the output of a valve to realize a stuck-at-closed failure. ■

To perform the safety analysis with model checking, the extended system model created in Simulink must be converted into the correct input language (input language of the model checker). The transformation algorithm fully automatically creates an equivalent SMV module for each Simulink block. After the conversion to the input language of NuSMV, the model can be verified against a safety requirement specified in CTL. If the specification is not fulfilled by the model, NuSMV will return just one single counterexample even though there could exist more. Because of the limited functions of failure modeling in Simulink, Joshi and Heimdahl introduced LustreFM [JH07]. LustreFM is a dataflow language based on Lustre, which provides special functions for failure modeling like conditional fault activation. Thanks to this language, the fault model can be separated from the nominal system model (defined in Simulink). Despite these improvements, the limitation to directed connections in Simulink remains.

3.4.2 SAML

Another approach based on formal verification techniques is SAML. SAML (Safety Analysis and Modeling Language) enables the construction of models with probabilistic and non-deterministic behavior [GO11][GO10]. SAML is used as an intermediate layer between engineering and verification. System models in Simulink, for example, are transformed into equivalent SAML models. During this transformation step, a fault model is added. With SAML as an intermediate layer, various analysis tools can be used. Among others, the transformation from SAML into the input language of model checking tools NuSMV and PRISM is possible. Thus, qualitative and quantitative analysis is possible.

SAML is based on finite stochastic state automata, which are all executed in parallel. Models in SAML are constructed using so-called modules, which consist of variables and transitions. State variables are either enumerations, bounded integers, doubles, or boolean variables. Transitions between states can be specified in non-deterministic and probabilistic ways. Components can be nested to model hierarchical systems. Note that there are no explicit events in SAML (as for example in AltaRica 3.0) [Lip+15].

Example 3.4.2 Example system model in SAML
The following SAML code shows a simple model consisting of a power source, a switch, and an actuator.

```
component main

constant double f_powersource := 1E-13;
constant double r_powersource := 1E-6;

formula is_ps_on := switch.state=CLOSED & power_source.failure_mode=OK;

component power_source
  enum FM_STATE := [OK, FAILURE];
  failure_mode : FM_STATE init OK;

  failure_mode=OK -> choice:(f_powersource:(failure_mode'=FAILURE) +
     (1-f_powersource):(failure_mode'=OK));
  failure_mode=FAIL -> choice:(r_powersource:(failure_mode'=OK) +
     (1-r_powersource):(failure_mode'=FAILURE));
endcomponent

component switch
  enum SW_STATE := [OPENED, CLOSED];
  state : SW_STATE init OPENED;

  state=OPENED -> state'=CLOSED;
  state=CLOSED -> state'=OPENED;
endcomponent

component actuator
  enum AC_STATE := [OFF, ON];
  state : AC_STATE init OFF;

```

```
30    state=OFF & ps_on -> state'=ON;
31    state=ON & !ps_on -> state'=OFF;
32  endcomponent
33  endcomponent
```

Constants and formulae (see lines 3-6) are macros that can be used in any subcomponent. The component model of the power source consists of a variable that specifies the failure mode of the component (see line 10). Initially, the component is in state `OK`. State variable updates (see lines 12-15) are defined by non-deterministic choices. For example, the power source changes its state from `OK` to `FAILURE` with a probability of `f_powersource` (=$1E-13$) or it remains in state `OK` with a probability of `1-f_powersource`. The actuator operates if the switch is closed and the power source is in state `OK`. ■

3.5 Classification

The presented approaches to MBSA have different characteristics in different aspects. Figure 3.6 categorizes the approaches based on the distinguishing criteria presented at the beginning of the chapter.

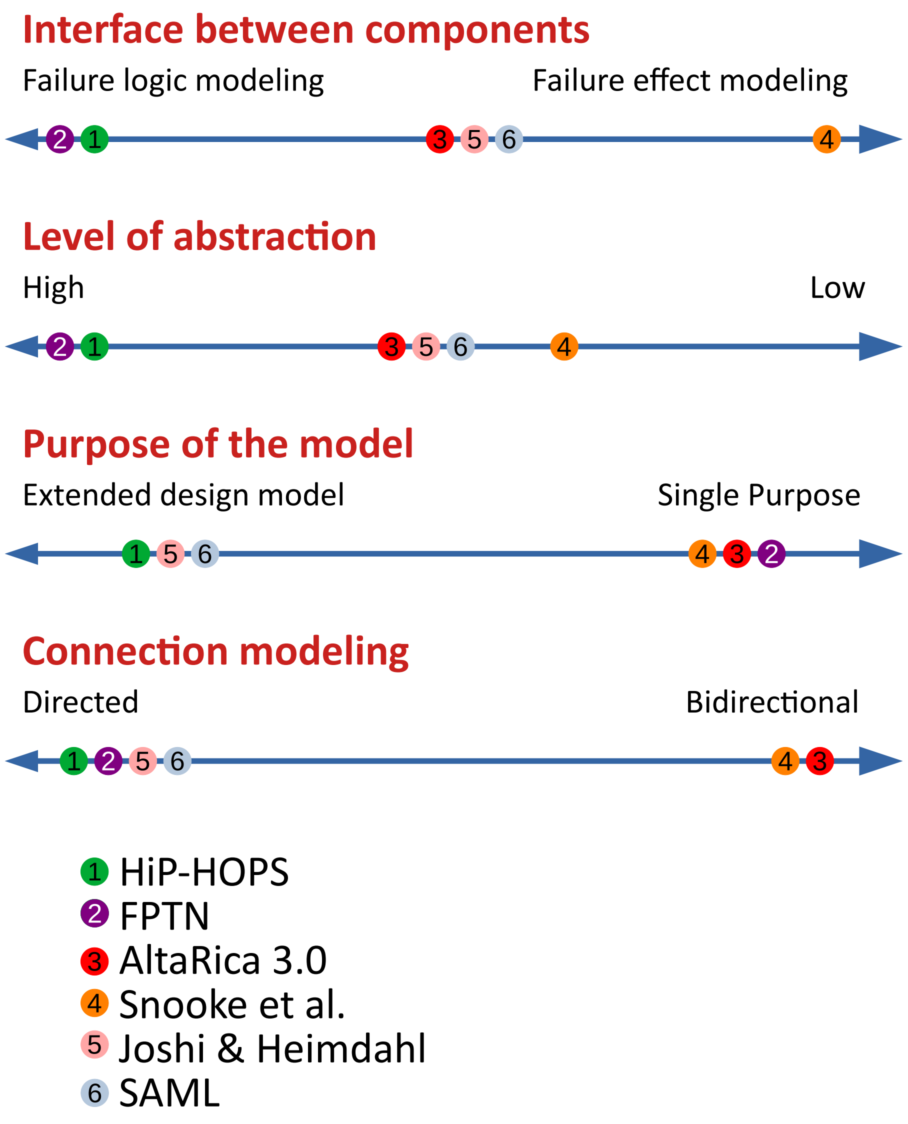


Figure 3.6: Classification of relevant approaches to MBSA

4 The smartIflow Formalism

"Any verification using model-based techniques is only as good as the model of the system."

Principles of Model Checking [Cla08]

This chapter introduces the new smartIflow formalism and its underlying concepts. The chapter is structured as follows: After stating the motivation for a new approach, it is discussed which basic concepts a new approach should feature. The combination of all these concepts forms a modeling formalism called smartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs). Subsequently, the main aspects of the smartIflow language are presented. After that, the smartIflow system models are formally defined and their semantics as well as the unfolding to transition systems are described. Finally, an approach for efficient graphical modeling is presented.

4.1 Why yet another Approach?

Today there exist many tools for automated reasoning about faults are available in early phases of product development. However, there is no silver bullet. But what are the reasons for this issue?

The greatest challenge in the automation of safety-related tasks is probably the choice of an appropriate level of abstraction. HiP-HOPS uses an extremely high level of abstraction to perform a steady-state analysis. Components only share information about possible failures (e.g., the omission of a signal) via directed channels. The behavioral description only specifies how a failure at the inputs affects the outputs. Therefore, component models can be constructed quite easily, models remain lightweight, and there are probably fewer problems with computational effort. However, as attractive as this kind of modeling looks, some drastic limitations need to be considered. One problem is the lack of nominal behavior. In real systems, faults can also occur due to a certain combination of operational modes. Consider for instance two actuators working against each other due to an operational mistake and as a result causing physical damage. Such situations cannot be handled with HiP-HOPS. Another problem is the lack of state variables. Therefore, it is not possible to specify state-dependent behavior, which is problematic for many situations. Consider for example a fuse. Normally, a fuse only breaks when the current exceeds a certain value for a certain duration. A broken fuse behaves totally differently compared to a deactivated fuse (the electric circuit is interrupted if the fuse is blown). HiP-HOPS cannot describe this behavior. Another problem arises with the limitation to directed connections. Situations in which the cause-effect relationship may reverse obviously cannot be handled. A way to get rid of this problem is artificial connections. Every time the cause-effect relationship may reverse, artificial connections need to be created. These artificial connections are also needed for components that are not directly connected but may influence each other. Unfortunately, these artificial connections cause structural differences between the real system and the model and models become more and more complex.

A great advantage of formal verification is the exhaustive analysis and reliable results (if the model is correct). Once the model and specification are defined, the method works fully automated. Model checking is a formal verification method used by several approaches for automated safety analysis. Model checking can find relevant effects that are invisible in simulation-based approaches. Compared to HiP-HOPS, approaches based on model checking incorporate more knowledge into the models so that much more detailed results can be obtained. However, most model checkers (e.g., NuSMV) deliver only one counterexample if the model does not fulfill a certain specification even though there could exist more. This is a rather drastic limitation. Safety engineers know that their system will eventually fail (with a certain number of failure events). For safety engineers, the reasons for a system failure are way more important. A single counterexample indicates only one scenario that led to the specification violation but does not list all possible causes. Another problem is the input languages of model checkers. Most model checkers limit the communication between components to directional channels. This results in the same problems that occur with HiP-HOPS. Timing is crucial in many safety-critical systems. However, there are only limited possibilities in the input languages of model-checking tools to specify temporal behavior. For example, delays must be specified in NuSMV using counters. Another problem arises when specifying the safety requirements in temporal logic such as CTL or LTL. The temporal operators of CTL or LTL are often not sufficient to specify safety requirements for real-world applications. Consider the following example: $\mathsf{AF}\,\Phi$. The expression only states that Φ must hold somewhere in the future. This could be in the next state or after any number of intermediate states. Understandably, this is not accurate enough, since system reactions must often occur within a certain duration. Another issue is the state space explosion. The state space grows exponentially with an increasing number of variables. This is problematic with larger systems since verification is often not possible at all or only with very high computational effort. The approach by Joshi et al. also suffers from limited fault modeling in Simulink. Transient faults and the differentiation between external or internal failure are not supported. The number of failures that are triggered can only be limited to an upper bound using an additional expression in CTL. There is no possibility to specify classes for the different failure events. Furthermore, the incorporation of failure behavior into the system model increases the complexity and leads to clutter. From a high-level perspective, no indicator shows which component has already been extended with failure behavior. Component changes (e.g., adding an additional failure behavior) require a lot of work since each component must be updated manually.

In [Bie+04], Bieber et al. present lessons learned of using AltaRica (the original version) for the safety assessment of two aircraft systems. Apart from problems with cycles in the system topology (such systems are rejected), they also found weaknesses in the definition of safety requirements using observers. AltaRica significantly evolved over time, and at first glance AltaRica 3.0 seems to be the perfect tool for safety analysis in the early product life cycle stages. Connections between components are bidirectional and looped systems are supported. Even though the expressiveness of AltaRica 3.0 is much higher in comparison to its original version, (but also in comparison to HiP-HOPS) there are still some limitations. In AltaRica 3.0, flow variables are used to share information with other components. Flow variables are restricted to a specific domain and can transfer only a single value. This leads to problems when implementing a bus-like communication between components where different information (e.g., different sensor values or logical signals) can be shared. A possible solution could be an enumeration type where all potential values are defined, though it is not possible to combine numbers with enumeration types. Another example where this limitation leads to difficulties is the modeling of a leakage in a hydraulic circuit. In order to indicate a leakage properly, information about fluid and pressure must be propagated. AltaRica 3.0 is not capable to propagate this pair of information. Even though AltaRica 3.0 supports bidirectional flows, there is no possibility of physical flow direction determination. In AltaRica 3.0 all state transitions are triggered completely by external events. There exist only limited features for event synchronization. Even though there are guard expressions to trigger events only in certain situations, no mechanism to define an upper bound for the activation of events exists. For example, there is no possibility to set a limit of double faults or to trigger operation events only (nominal behavior). Therefore, the state space can become very large in bigger systems. Complexity can also become a problem when building the reachability graph of a GTS. In order to manage the high complexity, GTS are split into independent subsystems. However, if a system cannot be divided into independent parts, which pertains to many real word systems, the state space of the reachability graph explodes. Another problem arises when specifying safety requirements using observers. For example, an observer cannot differentiate between a temporal or permanent loss of electrical power:

```
observer Boolean ConsumerIsNotSupplied = Consumer.input == false
```

Last but not least, physical system models (e.g., Simscape) are too detailed to obtain widespread results.

This short discussion reveals that there is a great demand for a new approach. But how could a new approach that is capable of overcoming the limitations of the existing tools look like?

4.2 Towards a new Modeling Formalism

This section implements *Contribution C1*.
To overcome the weaknesses of the existing tools, a new modeling formalism called smartIflow (State Machines for Automation of Reliability-related Tasks using Information FLOWs) was developed [HL14]. smartIflow tries to eradicate the flaws of existing tools by combining concepts from existing approaches and adding some new features.

The main idea behind smartIflow is quite simple: Each component in a system is considered as a finite state machine. Therefore, a component model consists of several typed variables that represent, for example, the operational or failure mode of a component. Variables take values of discrete domains. Each variable has a defined initial value. So far, this is nothing new. Differences occur in the mechanisms for state changes. While

AltaRica 3.0 only uses external events, smartIflow has two different mechanisms for state changes. The reason for this is that internal and external influences require quite different mechanisms for state changes. In addition, this strict separation between internal and external influences enhances the readability of the models. External influences are for example the activation of a failure situation or changing the operational model of a component. In smartIflow they are modeled using events and corresponding transition statements, much like in AltaRica 3.0. The events of a component are just a set of symbols that define possible external influences while transition statements specify the reaction to them. Each transition statement defines the effects to a specific event. Additional guard conditions can be added to the transition statements to trigger external events only in specific states, similar to guards in finite state machines. Consider for example an electrical switch that can be turned on or off. Changing the operational modes can be easily modeled using external events `TurnOn` and `TurnOff`. Triggering the event `TurnOn` only makes sense if the switch is in the off state, which can be ensured by the guard condition `state==off`. Guards are an important mechanism to reduce search space.

Internal influences are variable value changes or information shared by other components. The reaction to internal influences is also specified by means of transition statements. A transition statement for internal events consists of at least a triggering condition and a set of effects. The triggering condition is a logical expression of comparisons between symbolic values and all kinds of variables including information shared by other components. Transition statements are only triggered if the value of the triggering condition changes from false to true. Figure 4.1 illustrates this rising edge semantics which ensures that transition statements are not executed more often than necessary. Transition statements for internal influences can be equipped with guards, too. A transition statement is only executed if the value of the triggering condition changes from false to true and the guard condition is fulfilled. The effects of transition statements for internal and external events are defined by assignments to at least one variable. In many situations, the action effect of an event is not known. Therefore, smartIflow supports nondeterministic state transitions. Non-deterministic state transitions are modeled by assigning all possibly arising values to a variable: `var = Value1 or Value2;`

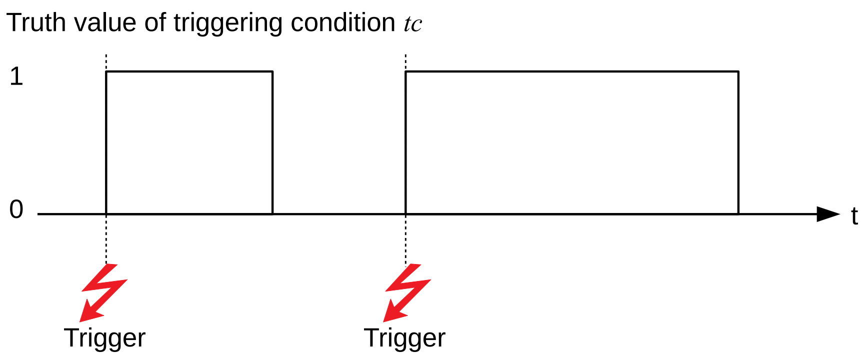


Figure 4.1: Rising edge semantics of transition statements in smartIflow

Let us continue with the component behavior. Components in a model need to somehow communicate with each other. Consider for example a valve that controls a flow depending on the control signal. The control signal can be provided by a controller that receives and processes data from sensors. Information shared between components is quite diverse. Components can for instance share information about flow, logical signals, or sensor values. In smartIflow the communication between the components is achieved by means of ports and connections. Each component has a set of ports that represent the possible connection points of a component. The ports in smartIflow are typed, and, unlike AltaRica, a port can be assigned multiple values. Connections are used as communication channels between components. Of course, these connections must propagate information in both directions. Otherwise, situations in which the cause-and-effect relationship reverses cannot be handled. In smartIflow, components publish messages in terms of so-called properties (properties are going to be addressed later on in this chapter). Once a component publishes a property at a port, the property is then propagated to all other connected ports. Sometimes it makes sense to restrict propagation at the port level, for example when modeling a sensor. A sensor could be modeled by publishing the captured value at a port. In principle, the sensor is not dependent on any information from the connected components. Therefore, information flow at the sensor can be limited in such a way that only the captured sensor values can be propagated, but no information from connected components can be read by the sensor. For this reason, ports in smartIflow can be restricted to inputs or outputs.

Example 4.2.1 Inputs and outputs in smartIflow
Figure 4.2 shows a simple model consisting of three components. Each component is equipped with one port. Port `p1` of component `B` allows flow of information in both directions (no restriction). Component `A` has an output port and component C has an input port. Let us assume each component sets a property on its port:

- Component `A`: `stateA=active`
- Component `B`: `stateB=active`
- Component `C`: `stateC=active`

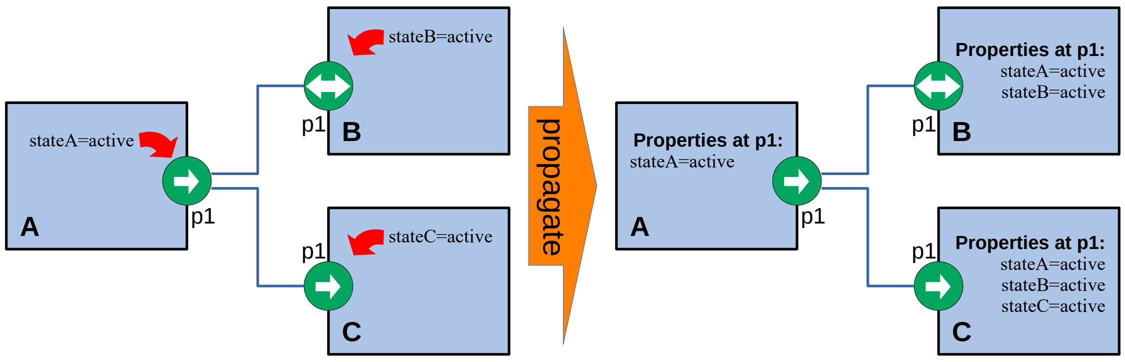


Figure 4.2: Effects of inputs and outputs in smartIflow

Figure 4.2 shows the properties available at ports inside components before and after propagation. Although components `B` and `C` are not directly connected to each other, the property published by component `B` (`stateB=active`) is available inside component `C`, since the property from component `B` is propagated to component `A` and finally to component `C`. Note that the properties available on a port can be different inside and outside of its component. For example, at port `p1` inside component `A` only property `stateA=active` is available, because `p1` does not let any information flow inside. When accessing port `p1` of component `A` from outside, properties `stateA=active`, `stateB=active` and `stateC=active` are available. ■

The components should be able to establish the connection structure depending on their current state. Consider for example a simple electrical switch that connects its two ports in state closed. The switch could be modeled as follows: The state of the switch is described by a variable with values `Opened` and `Closed`. Changes in the operation mode are initiated by external events and corresponding transition statements. Two ports (`p1` and `p2`) define the connection points of the switch. A connection between `p1` and `p2` may only be established in the state `Closed`. Therefore, the behavior description of a component in smartIflow consists of a set of conditional branches, in which the behavior of the respective state is described. Conditions only consist of logical combinations between variables and their possible values. The behavior of a component is described in terms of modifications of the network structure and the publication of properties across the network. A connect function is provided to link two ports. Once two ports are connected, properties can be passed through them.

The kind of information the components exchange with each other has not been discussed in detail yet. In AltaRica 3.0 the possible messages exchanged between components are limited to the domain of flow variables. Therefore, only two flow variables of the same type can be connected. In addition, as already mentioned, a flow variable can always be assigned only one specific value. This kind of message exchange is quite limited. For example, it is not possible to propagate information about different sensors simultaneously using a single flow variable. For this reason, smartIflow provides a flexible mechanism for message exchange between components. Components communicate with other components by writing properties to ports which are then propagated to all other connected ports. In fact, properties are simple key-value pairs. The properties are used to abstract from physical signals. For example, the values of an acceleration sensor can be abstracted by the properties `sensor1=low` and `sensor1=high`. The key (`sensor1`) indicates the origin of the information. Thus, it is possible to distinguish between several sensor values. Components have access to the properties at ports and can react to them using transition statements. Depending on the shared information, a component may update its variable values, modify the connection structure, or add additional properties. Components may lose their context freedom. The benefit of this approach however is an extremely flexible mechanism for message exchange.

Example 4.2.2 State-dependent behavior in smartIflow
Consider a simple valve that controls the flow of fluids. The valve can either be closed or opened. Only in state `Open`, fluid can flow through the valve. The component can be

modeled in the smartIflow language[1] as follows:

```
1  class Valve {
2      Ports:
3          Fluidal p1;
4          Fulidal p2;
5          Signal ctrl;
6  
7      Variables:
8           Enum[Open, Closed] state;
9           Enum[Ok, StuckOpen, StuckClosed] fm;
10 
11     Events:
12         ActivateStuckOpen {type=failure};
13         ActivateStuckClosed {type=failure};
14 
15     Behavior:
16         if (state == Open)
17             connect(p1, p2);
18 
19     Transitions:
20         when (ctrl.exists(cmd=Open) && fm != StuckClosed)
21             state=Open;
22          when (ctrl.exists(cmd=Close) && fm != StuckOpen)
23             state=Closed;
24 
25     EventHandlers:
26          when (ActivateStuckOpen) [fm == Ok && state == Open]
27              fm = StuckOpen;
28           when (ActivateStuckClosed) [fm == Ok && state == Closed]
29              fm = StuckClosed;
30 }
```

In smartIflow, component models are defined by means of classes. Ports `p1` and `p2` define the connection points between which the flow of fluid or gas is controlled. The transition statements in lines 20-23 define the reaction to control signals on port `ctrl` (published by other components). The property `cmd=Open` defines the command to open the valve. If the valve receives this property, the value of variable `state` is changed to `Open`. However, this transition may only be executed if the valve is not stuck at position `StuckClosed`. Stuck-at faults are modeled by the events `ActivateStuckOpen` and `ActivateStuckClosed` and corresponding transition statements (see lines 26-29). Note the guard conditions in the square brackets that limit triggering the events to specific situations. For example, a stuck-at-closed failure may only be activated if the valve is closed and works properly (`fm=Ok`). The state-dependent behavior of the valve is quite simple: A connection between port `p1` and `p2` is established only in state `Open`. Other components that are somehow connected to port `ctrl` can publish the property `cmd=Open` respectively `cmd=Close` to control the valve (e.g., `set(valve1, [cmd=Open])`). ■

An important aspect that has been ignored so far is timing. The components in real

[1]The smartIflow language is described in detail in Section 4.3

systems react to events at quite different speeds. For example, emptying a reservoir filled with hydraulic oil takes much longer than a controller's response to changes at inputs. To model such delayed reactions, internally and externally triggered transition statements need to be extended with timing information. smartIflow allows to specify order-of-magnitude (OM) values in transition statements for internal events to quantify delays. Consider for example the transitions statements ts_1 with $delay_{ts_1} = 4$ and ts_2 with $delay_{ts_2} = 2$. If ts_1 is triggered, corresponding state changes will happen after a delay of four orders of magnitude above reference. The effects of ts_2 will happen after a delay of two orders of magnitude above reference. If ts_1 and ts_2 are triggered simultaneously, effects of ts_2 happen immediately, while ts_1 must be delayed (the delay of ts_1 is larger). Therefore, all internally triggered transition statements are queued and only transitions with the minimum value (=active transition statements) are executed in the next step. All active transition statements are executed simultaneously leading to at least one subsequent state. In the case of nondeterministic state transitions, the result is a set of subsequent states.

Example 4.2.3 Delayed transitions in smartIflow
A security alarm system as depicted in Figure 4.3 consists of a set of sensors (e.g., motions sensors, glass-break, or infrared detectors) that are connected via bus technology or wirelessly to a central controller (premises control unit).

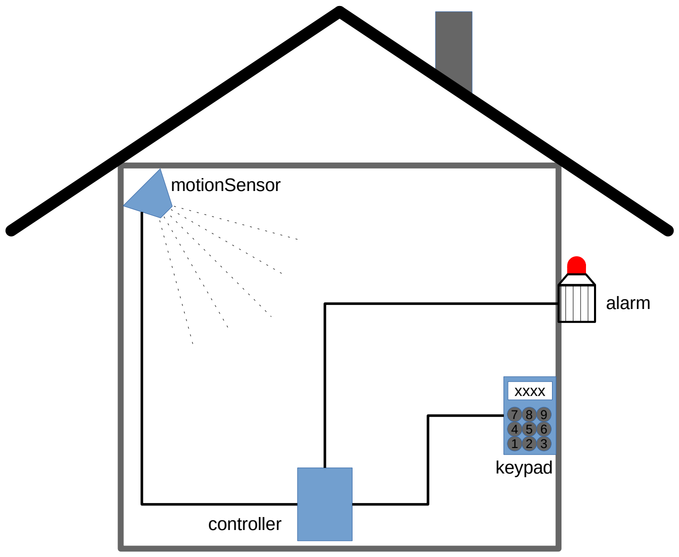


Figure 4.3: Architecture of a security alarm system

The premises control unit reads the sensor/detector states and signals intrusions. An intrusion must only be signaled if the security alarm system is armed. An authorized person must be able to disarm the alarm system. Many alarm systems can be disarmed by entering a pin on a keypad that is installed inside the building. In order to prevent an unwanted alarm from going on when entering the building, the alarm system triggers with a predefined delay. Modeling this control logic in smartIflow requires appropriate delay values and guards:

```
1 // ...
2 Transitions:
3    when {delay=3} (motionSensor.exists(motion=true)) [state ==Armed]
4            alarm=On;
5    when {delay=1} (keypad.exists(disarmed=true) && state == Armed)
6            state=Disarmed;
7 // ...
```

If a motion is detected, intrusion is signaled with a delay of three orders of magnitude (above reference), but only if the security alarm system is still armed. Order of magnitude value of three represents several seconds, while the order of magnitude value of one is about a second. Within this delay, the alarm system can be disarmed by entering the correct pin. If the alarm system is disarmed, the guard (`state == Armed`) will prevent executing the delayed transition statement (line 3). This example demonstrates that the guard condition of a delayed transition statement must be rechecked before it is actually executed. ■

Each state is stable with respect to a certain order of magnitude (OM) value. A state is stable to order of magnitude value t_{min} if all internally triggered transition statements have a delay of OM value of t_{min}. To make sure that every delayed transition statement is executed after it is triggered, no infinite path of states with low OM values must exist. Transition statements are only executed if the guard condition is fulfilled. Therefore, it is essential to recheck the guard conditions of delayed transition statements before executing them. A delayed transition statement is removed from the queue once the guard condition becomes false.

A delay value for externally triggered transition statements does not make sense. Once an external event is triggered, the effect should take place immediately. However, it makes sense to define a filter condition to limit the activation of external events to states that are stable at least with respect to the specified value. Executing an external event in all conceivable situations (where the guard condition is satisfied) will certainly explode the state space. Therefore, externally triggered transitions can also contain an OM value that defines a stable condition. The stable condition does not cause delays in execution but filters triggers for external events. This is an important feature to control the frequency of external events and consequently to keep the search space manageable.

Example 4.2.4 Stable condition in transition statements for externally triggered events
The listing below shows a transition statement for the external event `ActivateStuckOpen`. The stable condition limits the execution of `ActivateStuckOpen` to states that are stable at least with respect to five orders of magnitude above reference.

```
when {stable=5} (ActivateStuckOpen) [fm == OK && state == Open]
   fm = StuckOpen;
```

■

A stable condition can also be specified for transition statements for internally triggered events. The semantics is the same as for the external events. Thus, the activation of internal events can be limited to states that are stable to a certain value. Note that the stable condition in internally triggered transition statements will disable the rising edge semantics.

Properties offer a quite flexible communication between components. For some scenarios, however, a behavioral description on a physical level would be desirable. For this reason, smartIflow integrates the qualitative order of magnitude energy-flow-based reasoning introduced in [SL14]. The concept is implemented in smartIflow in the following way: Each connection that is part of the power network is assigned a resistance value by means of features:

```
// connect p1 and p2 with a resistance of two orders of magnitude above
    reference:
connect{r=1}(p1, p2);
```

Sources are created by adding built-in features to a component class. The following component class represents a bipolar effort source for electrical circuits with a power of four orders of magnitude above reference:

```
class PowerSupply {flowComponent=bipolarEffortSource, om=4}{
   Ports:
      Electrical{flowPort=source} plus;
      Electrical{flowPort=drain} minus;
}
```

After qualitative energy flow analysis, the flow and effort values are stored in terms of simple properties at ports, which can be considered in the triggering conditions of the transition statements:

```
when(flow.om(p1,p2) > 1) {
  //...
}
```

To summarize, smartIflow is a formalism for model-based safety analysis with the following characteristics:

- smartIflow incorporates more knowledge into the models compared to other competitors (see Figure 4.4)
- Components in a system are considered as a finite state machines:
  - Each component consists of a set of typed variables with discrete domains
  - Events and corresponding transition statements specify reactions to external influences
  - Internally triggered transition statements define the reactions to changes on ports or variables
  - Internally and externally triggered transition statements can be extended with information about timing in terms of order of magnitude values (delays/stable conditions)
- Each component in a system is described by a separate smartIflow system model
- Behavior is generally described qualitatively and object-oriented

- Ports represent the connection points of a component
- The dynamic connection structure relies on bidirectional connections
- Components communicate with each other by writing properties to ports which are propagated to all other connected ports
- Systems behavior can also be described at the physical level using the qualitative energy flow analysis

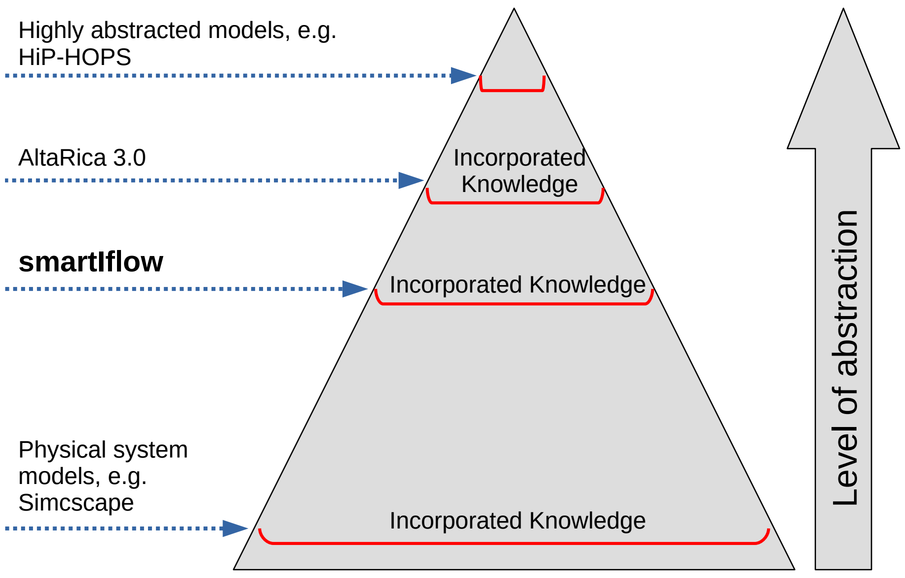


Figure 4.4: Positioning of smartIflow at level of abstraction with respect to competitors

In fact, smartIflow system models are extended/modified program graphs that implicitly describe the transition system. The explicit representation of the transition system can be obtained by unfolding. In unfolding, the following steps are performed alternately:

- Process behavior: Establish dynamic connection structure based on current variables values, add properties to ports, and propagate them
- Determine subsequent states: Execute possible transition statements for internal and external influences

Before focusing on the formal basics of smartIflow system models and the unfolding function in detail, first the smartIflow language will be introduced in the following.

4.3 The smartIflow Language

For automated reasoning about faults, smartIflow models must be specified in a formal and computer-understandable way. This chapter introduces the smartIflow language that is used to create smartIflow system models.

System models are typically created with the help of modeling languages or formalism. Joshi et al. [JWH][Jos+05] use for example logic blocks and the Simulink system editor for modeling. Afterward, the Simulink models are transformed into the input language of the symbolic model checker NuSMV. This approach is attractive to those who are not familiar with programming languages or textual modeling formalism. AltaRica 3.0 [Bat+13] is an example of an approach using a textual modeling language. The language was designed specifically for the formalism. Models remain lightweight and all features of the modeling formalism are supported. Both types of languages have their benefits. Therefore, the smartIflow approach will provide both a textual and a graphical modeling language. The textual modeling language is called smartIflow language and will be described in this section. Later a concept for efficient modeling based on a graphical formalism will be presented in this chapter (see Section 4.8).

Java is a beginner-friendly, modern, and popular programming language. The smartIflow language[2] is inspired by Java and therefore shows some common characteristics. The smartIflow language is an object- and component-oriented language in which components of a system are represented by classes. Components of the same type need to be modeled only once and can be instantiated several times. A regular source file that defines a component class, port, or enumeration must be named after the type it contains, ending with *".sif"*. For example, the component class *Switch* must be named *Switch.sif.* In addition, there are special source files that may contain multiple type definitions. These source files must be named *"definitions.sif"* and can be placed in any package (the package concept will be introduced afterward). The definition files are especially helpful for the elements that are used in several components (e.g., port types or enumerations). A single import statement can be used to import several types (described in a definitions file) at the same time.

First, some basic aspects of the smartIflow language will be described. After that, the different types (class, port, enum) will be explained in detail.

4.3.1 Fundamentals

Identifiers As in most programming languages, identifiers in smartIflow are names given to classes, variables, events, or transitions to refer to the element from somewhere else. The rules for defining a valid identifier in smartIflow are equivalent to Java:

- Reserved keywords cannot be used as an identifier (e.g., `class`, `Enum`, `if`, or `when` cannot be used as an identifier)
- Identifiers are case-sensitive
- Allowed characters are digits, alphanumeric characters, and the underscore

[2]The entire grammar of smartIflow is shown in Extended Backus Naur Form (EBNF) in Appendix A.1

- Identifiers must not start with a digit

Properties Properties are, as already described in Section 4.2, simple key-value pairs. The key can be any identifier. The value can be an identifier or a number (integer or double). Properties are separated by an equal sign. The key is placed on the left side and the corresponding value is placed on the right of the equal sign. Valid properties are for example `key1=value1` or `e=1e-9`. A set of properties is comma separated and grouped in curly brackets: `{k1=v1, k2=v2, ...}`

Features Features extend, for example, variable values, port instances, or events with additional data such as information about timing or probabilities. Features are, like properties, key-value pairs where the key and value are separated by an equal sign. The key can be an arbitrary identifier, the value of an identifier, a number (integer or double), or a string literal. Like properties, a set of features is comma separated and grouped in curly brackets: `{f1=v1, f2=1e-9, f3="Some text", ...}`

Packages Same as in Java, packages in smartIflow language are used to group related definitions together. Packages can be nested into each other. The names of nested packages are separated by a dot. Definitions from other packages can be used either by using the full qualified name or by preceding an import statement. The source code of any type starts with a package declaration, followed by some import statements and after that, the actual definition of the type follows. Packages not only provide a better overview but also help with naming conflicts, since only elements within a package need to be uniquely named.

Comments In smartIflow language, there are two possibilities for writing comments:

1. Block comments start with `/*` and end with`*/` and comments out everything in between. The text in the block comment must not contain any `*/` itself, as block comments must not be nested
2. Line comments start with two slashes `//` and comment out the remaining content of a line

Statements and Blocks In smartIflow language, the semicolon (`;`) is used to indicate where a statement ends. Function calls, declarations, or variable assignments must be terminated with a semicolon. A block is a statement that combines a sequence of statements in curly braces `{ }` to a new statement. In principle, the complete code of a component class in smartIflow language could be written in one single line without indentation or line breaks.

4.3.2 Type *port*

Connection points of smartIflow components (=ports) are typed. The port type defines the set of admissible properties that could be present at the port. Note that different properties can be present at a port at the same time. A port type definition consists of a set of property domain specifications. The property domain specification defines for the key of a property the set of admissible values. In the smartIflow language, a port type is defined using the keyword `port`. For example, a port type definition allowing the properties `key1=value1`, `key1=value2`, `key2=value1`, `key2=value2`, and `key2=value3` to be present at a port is defined as follows:

```
port PortType1 {
  Properties:
    key1 = {value1, value2};
    key2 = {value1, value2, value3};
}
```

A port type without any property domain specifications allows any kind of property to be assigned to a port of this type.

4.3.3 Type *enum*

Type `enum` is used to define a set of predefined values for variables. The value of a variable must always be one of the values defined in the predefined constants. There is no limit on the number of variable values, but at least one value must be defined. Optionally, variable values can be extended with features. The following example shows the definition of an enumeration type:

```
enum Type1 {
   A, B, C, D, E{feature=value}
}
```

A variable of type `Type1` can be assigned the value `A`, `B`, `C`, `D`, or `E`.

4.3.4 Component Classes

In smartIflow language, classes are used to represent components in a system. A component class is defined using the keyword `class`. For the sake of clarity, the various characteristics of a component model (e.g., ports, variables, state-dependent behavior, or transitions) are separated into seven sections, namely `Components, Ports, Variables, Events, Behavior, Transitions`, and `EventHandlers`. Each section is optional, however, the order of the sections is fixed:

```
class Test {
    Components: //...
    Ports: //...
    Variables: //...
    Events: //...
    Behavior: //...
    Transitions: //...
    EventHandlers: //...
}
```

Components: The smartIflow language supports hierarchical modeling. The subcomponents of a component are defined in section `Components`. A component class instance is defined by the name of the class followed by a unique identifier. Naturally, instances of a class in the class itself cannot be created (this would lead to infinite recursion). The following example demonstrates the instantiation of two components:

```
Components:
   Type1 instance1;
   Type2 instance2;
```

Ports: In this section, the connection points of a component to communicate with each other are specified. To instantiate a port in smartIflow, a port type followed by an identifier must be added to section `Ports`. Optionally, the information flow at a port can be restricted to a certain direction by using the built-in features `direction=input` respectively `direction=output`.

```
Ports:
  PortType1 input{direction=input}; // Port that allows flow inside
  PortType1 output{direction=output}; // Port that allows flow outside
  PortType1 port; // Port that allows flow in both directions
```

Variables: The variables of a component are specified in section `Variables`. Basically, variables can be created in two different ways. Either the variable with the corresponding domain is defined and instantiated directly in the class itself using an anonymous type ① or a variable type is defined in a separate file and in section `Variables` only instances are created ②. The first method is quite useful for special variable types that are only required once because they can be defined and instantiated directly within the classes without any additional variable type definition. The latter has the advantage that in case of multiple instances, the variable type must only be defined once, and several instances can be created.

The instantiation of a variable inside a class using an anonymous type ① starts with the definition of the domain using the keyword `Enum` and a set of constants in square brackets followed by an identifier. A variable instance of a predefined enum ② is created by using the type name followed by an identifier. The assignment of the initial value is optional for both methods. If no initial value is specified, the first value of the domain is used. The variables within a class must be uniquely identified. Optionally, features can be added to variable values or instances.

The following listing shows examples of both methods:

```
Variables:
   Enum[Value1, Value2, Value3] var1 = Value2; // ①
   Enum[Value1, Value2] var2 = Value2; // ①
   Type1{feature=value} var3 = A; // ②
```

Events: The set of external stimuli is defined in section `Events`. An event is described by an identifier with an optional set of features. The features can be used to categorize the events in different groups (e.g., failure, operational, or repair events) or to specify the probability of occurrence. The following listing demonstrates the declaration of some events:

```
Events:
   Event1{type=opm};
   Event2{type=opm};
   FailureEvent{type=failure, p=1e-7};
   RepairEvent{type=repair};
```

Events within a class must be uniquely identified. In section `Events` the events are just declared, while the reaction to them is described in section `EventHandlers`.

Behavior: Section `Behavior` is used to describe the state-dependent behavior of a component. The behavior is expressed in terms of set- and connect- functions. Function `set` is used to publish a set of properties to a port. Therefore *set* requires two parameters, namely the name of the port and a set of properties. Function `connect` establishes a connection between two ports and optionally publishes a set of properties. Only ports of the same type can be interconnected. The first two parameters of `connect` are mandatory, namely the name of the ports to be connected, while the set of properties is optional. The order of the parameters is fixed for both functions. set- and connect-functions declared directly in section `Behavior` are executed independently of the state of a component (=default behavior).

A connect-function with simultaneous property publication (using a single statement) is semantically equivalent to a simple connect-action and a set-action to one of the two connected ports (two statements):

```
Behavior:
   connect(input, output, {key1=value1, key2=value2});
   // is semantically identical to:
   connect(input, output);
   set(input, {key1=value1, key2=value2});
```

The state-dependent behavior is expressed in terms of conditional branches. smartIflow provides if- and if-else statements for this purpose. An if statement consists of the keyword `if`, followed by a logical expression in parentheses. After that, either a single statement or a block with several statements can follow. The optional keyword `else` can be used to specify a statement or a set of statements in a block that is executed if the condition from the if part is false. In addition, smartIflow also supports nested conditional statements. Conditions for if-statements can be any logical combinations of comparisons between variables and their values using the relational operators `==` and `!=` as well as logical operators `!`, `||`, and `&&`.

```
Behavior:
   connect(input, output); // default behavior

   if (var1 == Value1 && var2 == Value2) {
      connect(input, port, {key1=value1, key2=value2};
   } else {
    if (var3 != E || var1 == Value2)
      set(port, {key3=value3});
    else
       set(port, {key4=value4});
  }
```

Transitions: The section `Transitions` consists of a set of when-expressions describing the variable value changes depending on signal changes on ports, variable assignments, or combinations of both. When-expressions have the following structure:

```
when{<features>①}(<triggering condition>②)[<guard>③] {
  <variable value assignments>④
}
```

When statements start with the keyword `when`, followed by an optional set of features ①, a triggering condition in parentheses ②, and an optional guard condition in brackets ③. After that, the variable value changes are specified using assignments ④. In smartIflow language, variables can be updated using the assignment operator (=) that assigns a value on its right to the variable on its left. Nondeterministic variable value changes are defined by assigning a variable a set of values separated by `or`. The set of features ① is used to enrich when statements with additional parameters. There exists a set of predefined features:

- **delay** specifies the delay of a transition statement
- **stable** is used to restrict the activation of a transition to states that are stable at least with respect to the specified value
- **name** is used to identify active transitions in the transition system

Triggering conditions ② are logical expressions similar to the logical expressions used in the behavior section, the difference being that property changes at ports can also be taken into account in trigger conditions. Function `exists` is a boolean function applied to a port, which tests the existence of a property (`port.exists(key=value)`) or a key (`port.exists(key)`) at the specified port. The function returns `true` if the property respectively the property key is available at the specified port in the current state, otherwise it returns `false`. The guard condition ③ is also a logical expression with the same rules as for the triggering condition.

```
Transitions:
  when{delay=5, name=Transition1}(input.exits(key=value))[var1 == Value1] {
    var2 = Value2;
    var3 = A or B or C;
  }

  when{stable=10}(var_1 == Value2 || var_2 == Value1)[var3 == A]
    var1 = Value1;
```

EventHandlers: In this section, the reaction to external events declared in section `Events` are described using when-expressions. Each when expression describes the reaction to an event and is constructed as follows:

```
when{<features>①}(<event>②)[<guard>③] {
   <variable value assignments>④
}
```

After the keyword `when`, an optional set of features ① can be defined. The only predefined feature in this context is `stable`, which is used to restrict the activation of a transition to states that are stable at least with respect to the specified value. Afterwards, the name of the event is specified ②, followed by a guard expression ③. This is followed by a set of variable value changes ④ like in section `Transitions`. Nondeterministic transitions can be described in the same way as with internal transitions. Only events declared within the component can be accessed in an event handler.

```
EventHandlers:
   when{stable=5}(Event1)[var1 == Value1] {
      var3 = D or E;
   }

   when{stable=10}(Event2) {
      var1 = Value1;
   }
```

Inheritance and Generics In the smartIflow language, it is possible to inherit components, ports, variables, events, behavior, and transition statements from one component class to another. The subclass can access the ports, variables, and events of its parent class. The inheritors can add further information and/or behavior, but can never override transition statements or behavior. To inherit from a class, the keyword `extends` is used. Note that a component class in the smartIflow language can inherit features from only one parent component class. The following example demonstrates how an existing component class can be extended by the use of inheritance:

```
class A {
    Ports:
        PortType p1;
        PortType p2;

    Variables:
        Enum[Value1, Value2] var = Value1;
}

class B extends A {
    Ports:
        PortType1 p3;

    Behavior:
        connect(p1,p2);

    Transitions:
        when (var == Value1)
            var = Value2;

}
```

Inheritance in the smartIflow language is an essential feature to avoid redundancy. Another helpful feature in the smartIflow language is the possibility to create generic component classes. A generic component class is created by adding type parameters to the class:

```
class GenericClass<T1, T2, ..., Tn> {
    // ...
}
```

Type parameters can be used only in section `Ports`. A type parameter can be any port type. The following listing shows a generic component class with two type parameters, `T` and `V`:

```
class GenericClass<T, V> {
    Ports:
        T p1;
        V p2;
}
```

When instantiating a generic component class, the type parameters must be replaced with some concrete value:

```
class AnotherClass {
    Components:
        GenericClass<Logical, Electrical> instance;
}
```

In this case, port `p1` of component `instance` is of type `Logical` and port `p2` is of type `Electrical`. Type parameters in the smartIflow language provide a way to reuse the same component class with different inputs for port types. For example, a valve can be modeled by a generic component class where the port type is specified using a type variable (e.g., Air or Hydraulic). Without generics, a separate component class would be necessary for each substance.

Example 4.3.1 Generic component and inheritance
The following smartIflow class defines a generic component for bipolar effort sources. Special class features (built-in features) are used to define the type of source and its order of magnitude. Class `BipolarEffortSource` defines a bipolar effort source whose power is two orders of magnitude above reference. Instead of using specific port types for the plus and minus port, we use the type parameter `T`.

```
class BipolarEffortSource<T> {flowComponent=bipolarEffortSource, om=2} {
    Ports:
        T{flowPort=source} plus;
        T{flowPort=drain} minus;
}
```

This type variable makes it possible to use the class `BipolarEffortSource` in quite different domains. For example, this bipolar effort source can be used as a battery in an electrical system:

```
class ElectricalNetwork {
    Components:
        BipolarEffortSource<Electrical> battery;
        Switch switch1;
    // ...
    Behavior:
      connect{r=1}(switch1.p1, battery.plus);
}
```

When instantiating class `BipolarEffortSource`, the type parameter `T` is replaced with the concrete port type `Electrical`.

However, it is also possible to use an instance of class `BipolarEffortSource` as a hydraulic pressure source in hydraulic power networks:

```
class HydraulicPowerNetwork {
    Components:
        BipolarEffortSource<Fluid> source;
        Line line1;
    // ...
    Behavior:
      connect{r=1}(line1.p1, source.plus);
}
```

■

4.4 smartIflow System Models

This section implements *Contribution C2*.
If the correctness of a system design is proven by showing the correctness of its model, it is extremely important that the meaning of the model is clearly defined. And this is exactly what is done in this section. After the formal definition of the smartIflow system models, the flattening and the semantics of the underlying transition systems are described.

Definition 4.4.1 smartIflow System Models
The set of all smartIflow System Models *SSM* is defined as a set of tuples of the following form:

$$ssm = (Comp, Port, Var, Init, Event, PA, Conn, TS_{\text{int}}, TS_{\text{ext}},)$$

where:

- $Comp$ is a function which maps part names to SSM instances,
- $Port$ is a set of typed ports,
- Var is a set of typed variables,
- $Init \in Eval(Var)$ is a set of initial variable value assignments,
- $Event$ is a set of symbols for external events,
- $PA \subseteq Cond_{Action}(Var) \times Effects_p$
  where $Effects_p = \{effect \mid effect : Eval(Port) \to Eval(Port)\}$
  is a set of conditional property assignments to ports,
- $Conn \subseteq Cond_{Action}(Var) \times P \times P$ where
  $P = \Big\{\langle p, \text{false}\rangle \mid p \in \bigcup\limits_{c \in Comp} c.Port\Big\} \cup \Big\{\langle p, \text{true}\rangle \mid p \in Port\Big\}$
  is a set of conditional connections between ports,
- $TS_{\text{int}} \subseteq Delay \times Stable \times Cond_{Trigger}(Var, Port) \times Cond_{Guard}(Var) \times Effects_v$
  where $Effects_v = \{effect \mid effect : Eval(Var) \to 2^{Eval(Var)}\}$
  is a set of transition statements for internal events, and
- $TS_{\text{ext}} \subseteq Event \times Cond_{Guard}(Var) \times Stable \times Effects_v$
  where $Effects_v = \{effect \mid effect : Eval(Var) \to 2^{Eval(Var)}\}$
  is a set of transition statements for external events.

$\blacksquare$

The set Var of variables is used to describe the possible states of a component. This includes operational modes and failure modes. Variables are typed and have finite domains. Initially, each variable $v_i \in Var$ is assigned its initial value specified by the set $Init \in Eval(Var)$. $Eval(Var)$ is the set of functions where each function maps each variable to a value of its domain. Let $Values = \{\bigcup\limits_{var \in Var} dom(var)\}$.

Formally, $Eval(Var)$ is defined by:

$$Eval(Var) = \{eval \mid eval : (Var \to Values \wedge \forall var \in Var : (eval(var) \in dom(var)))\}$$

Event denotes the set of symbols for external events, and *Port* denotes a set of connection points. Ports are typed and they can be assigned a set of properties. $dom(p)$ of port p defines the type of p. $dom(p)$ contains all allowed key-value pairs of the port type including the empty property. A port can only be assigned the set of properties that is defined in its type. Properties are tuples of the form $(key, value)$ where key is a symbol and $value$ is a number or a symbol. $Eval(Port)$ denotes the evaluations of properties to ports. The evaluation $e_p \in Eval(Port)$ is the mapping that assigns a set of properties to any port $p_i \in Port$. Let $KVM = \{kvm \mid kvm : KEY \to VALUE\}$. Formally, $Eval(Port)$ is defined by:

$$Eval(Port) = \{eval \mid eval : (Port \to KVM \wedge \forall p \in Port : (eval(p) \in dom(p)))\}$$

Example 4.4.1 3/3 directional control valve
Consider a 3/3 directional control valve equipped with three ports namely one source (s) and two outputs ($o1$ and $o2$). The valve has three spool positions. Depending on the position of the spool, the flow of fluid (e.g., gas or liquid) is controlled. As illustrated in Figure 4.5, the flow can be restricted or permitted to one of the outputs.

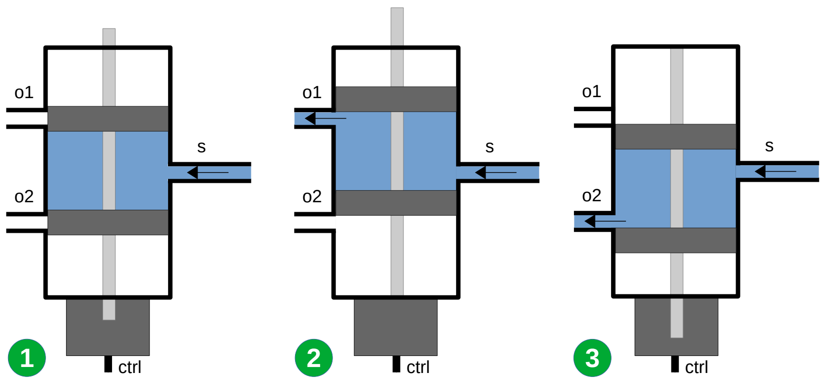


Figure 4.5: States of a three-position valve

The valve is characterized by the two variables om and fm with domain

$$dom(om) = \{Closed, Pos1, Pos2\}, \ dom(fm) = \{Ok, StuckAt, Leakage\}$$

Variable om is used to represent the different operational modes of the valve and fm captures the different failure modes. The initial variable assignment is

$$Init = \{om{=}Closed, fm{=}Ok\}$$

The set of symbols for external events are

$$Event = \{ActivateStuck, ActivateLeakage\}$$

For set $Port = \{s, o1, o2, ctrl\}$ two different types are required. The ports $s, o1, o2$ are of type *Fluidal*, while *ctrl* is of type *Logical*. The logical signals at port *ctrl* are used to control the spool and therefore to control the flow of fluid. Port types allow to restrict published properties to specific key-value pairs. For the port type *Logical* the properties *cmd=close*, *cmd=pos1* and *cmd=pos2* are allowed. For the port type *Fluidal* the properties *flow=low*, *flow=normal* and *flow=high* are allowed. ■

Let $Pred(Var)$ and $Pred(Port)$ be predicates. $Pred(Var)$ are predicates built on comparisons between variables and their values using comparison operators ($==, != $). $Pred(Port)$ are predicates built on calls of function `exists` that check whether a property is available at a port.

Formally, the propositional symbols of $Pred(Var)$ and $Pred(Port)$ are of the form "$var == val_{var}$" respectively "$p.exists(property_p)$" where $var \in Var$, $val_{var} \in dom(var)$, $p \in Port$ and $property_p \in dom(p)$.

$Cond(Pred(Var))$ is the set of Boolean expressions over Var formulated with logical operators ($\wedge, \vee, \neg$) and predicates $Pred(Var)$.

$Cond_{Trigger}(Var, Port) = Cond(Pred(Var) \cup Pred(Port))$ is the set of arbitrary combinations of conditions from Var and $Port$ that defines the triggering conditions for the internal events.

$Cond_{Guard}(Var) = Cond(Pred(Var))$ defines the guard conditions for the internal events and $Cond_{Action}(Var) = Cond(Pred(Var))$ is the set of conditions for the property assignments to ports as well as for conditional connections between ports.

Let $Cond(AP)$ be the set of boolean expressions over the set of atomic propositions AP with logical operators $\wedge$, $\vee$ and $\neg$, also including constants true and false.

Example 4.4.2 Conditions for transition statements and conditional behavior
A triggering condition for the valve described in Example 4.4.1 that is fulfilled if the property *cmd=Pos2* exists at port *ctrl* and the valve is not defective, can be formulated as follows:

$$fm{=}{=}Ok \wedge ctrl.exists(cmd{=}Pos2)$$

In the operational mode *Pos2* a connection between *s* and *o2* must be created. The condition for this conditional connection can be expressed as follows:

$$fm{=}{=}Ok \wedge om{=}{=}Pos2$$

■

$Stable \in \mathbb{N}_0$ are the stability values for the transition statements and $Delay \in \mathbb{N}_0$ defines the delay values for the transition statements.

Comp is a function which maps part names to *SSM* instances to enable hierarchical modeling. Hierarchical system models are in a tree structure. The set *Conn* of conditional connections is not limited to connections between the ports of a component itself. Connections to ports of any subcomponent can be established, too. Since information flows at ports can be limited to input or output, it is important to know whether two ports are connected inside or outside a component. For this reason, the two connection points of a component are extended with position information that indicates whether the connection takes place outside (true/false). Therefore, connections are pairs of the form (pc_1, pc_2) where

$$pc_i \in (\{\langle p_i, \text{false}\rangle \mid p_i \in \mathit{Port}\} \cup \{\langle p_i, \text{true}\rangle \mid p_i \in \bigcup_{c \in \mathit{Comp}} c.\mathit{Port}\})$$

The effect of conditional property assignments to ports is defined using the mapping

$$PA \subseteq \mathit{Cond}_{\mathit{Action}}(\mathit{Var}) \times \mathit{Effects}_p$$

which describes how the evaluation of properties at ports is changed by the conditional property assignments to ports depending on the current variable evaluation. We will use η_v for variable evaluations and η_p for evaluations of properties at ports.

Example 4.4.3 Variable and port evaluation
The valve example introduced in Example 4.4.1 consists of two variables, *fm* and *om*. As already mentioned, *Eval*(*Var*) is a set of functions that maps each variable to a value of its domain. Regarding the example, the following variable evaluations are possible:

$$\begin{aligned}\mathit{Eval}(\mathit{Var}) = \Big\{&\{\mathit{om}{=}\mathit{Closed}, \mathit{fm}{=}\mathit{Ok}\}, \{\mathit{om}{=}\mathit{Closed}, \mathit{fm}{=}\mathit{StuckAt}\}, \{\mathit{om}{=}\mathit{Closed}, \mathit{fm}{=}\mathit{Leakage}\},\\ &\{\mathit{om}{=}\mathit{Pos1}, \mathit{fm}{=}\mathit{Ok}\}, \{\mathit{om}{=}\mathit{Pos1}, \mathit{fm}{=}\mathit{StuckAt}\}, \{\mathit{om}{=}\mathit{Pos1}, \mathit{fm}{=}\mathit{Leakage}\},\\ &\{\mathit{om}{=}\mathit{Pos2}, \mathit{fm}{=}\mathit{Ok}\}, \{\mathit{om}{=}\mathit{Pos2}, \mathit{fm}{=}\mathit{StuckAt}\}, \{\mathit{om}{=}\mathit{Pos2}, \mathit{fm}{=}\mathit{Leakage}\}\Big\}\end{aligned}$$

Eval(*Port*) will generate the following evaluations:

$$\begin{aligned}\mathit{Eval}(\mathit{Port}) = \Big\{&\{\mathit{ctrl} = \{\mathit{cmd}{=}\mathit{pos1}\}, s = \{\mathit{flow}{=}\mathit{normal}\}, o1 = \{\mathit{flow}{=}\mathit{normal}\}, o2 = \{\}\},\\ &\{\mathit{ctrl} = \{\mathit{cmd}{=}\mathit{pos2}\}, s = \{\mathit{flow}{=}\mathit{normal}\}, o1 = \{\}, o2 = \{\mathit{flow}{=}\mathit{normal}\}\},\\ &\{\mathit{ctrl} = \{\mathit{cmd}{=}\mathit{pos1}\}, s = \{\mathit{flow}{=}\mathit{normal}\}, o1 = \{\}, o2 = \{\}\},\\ &\{\mathit{ctrl} = \{\mathit{cmd}{=}\mathit{close}\}, s = \{\mathit{flow}{=}\mathit{normal}\}, o1 = \{\}, o2 = \{\}\}, \ldots\Big\}\end{aligned}$$

■

Now we take a closer look at the transition statements. Let

- $e \in \mathit{Event}$
- $\mathrm{c_t} \in \mathit{Cond}_{\mathit{Trigger}}(\mathit{Var}, \mathit{Port})$
- $\mathrm{c_g} \in \mathit{Cond}_{\mathit{Guard}}(\mathit{Var})$
- $\mathit{effect} \in \mathit{Effects}_v$
- $\varsigma \in \mathit{Stable}$
- $\Delta \in \mathit{Delay}$

The notation $\eta_v \xhookrightarrow{\mathrm{c_t},\mathrm{c_g},\varsigma,\Delta}_{\mathrm{t}_{int}} {\eta_v}'$ is used as abbreviation for $(\eta_v, \mathrm{c_t}, \mathrm{c_g}, \varsigma, \Delta, \mathit{effect}) \in \mathit{TS}_{\mathrm{int}}$ and ${\eta_v}' \in \mathit{effect}(\eta_v)$. Note that *effect* is the effect function of a transition statement that updates variable values. $\eta_v \xhookrightarrow{\mathrm{e},\mathrm{c_g},\varsigma}_{\mathrm{t}_{ext}} {\eta_v}'$ is the shorthand for $(\eta_v, \mathrm{e}, \mathrm{c_g}, \varsigma, \mathit{effect}) \in \mathit{TS}_{\mathrm{ext}}$ and ${\eta_v}' \in \mathit{effect}(\eta_v)$. η_p denotes the evaluations of properties at ports. We access elements of transitions by dot notation. For example, for $ts_i \in \mathit{TS}_{\mathrm{int}}$ the expression $ts_i.\Delta$ delivers $\Pi_2 ts_i$ - the second element of ts_i – which is the delay value.

4.4.1 Flattening

For unfolding, smartIflow System Models are reduced to a single system model without any sub-components.

Definition 4.4.2 Flattened smartIflow System Models
The set of all flattened smartIflow system models *FSSM* is defined as a set of tuples of the following form:

$$fssm = (\mathit{Var}, \mathit{Init}, \mathit{Event}, \mathit{Port}, \mathit{TS}_{\mathrm{int}}, \mathit{TS}_{\mathrm{ext}}, \mathit{Conn}, \mathit{PA})$$

The *fssm* of a *ssm* is the result of a flattening function:

$$\mathit{Flatten} : \mathit{SSM} \to \mathit{FSSM}$$

Algorithm 4 shows the definition of the flattening function. ■

Name conflicts can occur if elements on different levels have identical names. Therefore, each symbol in the system model (e.g., events, variables, ports, . . .) is prefixed with a path name (the absolute path in the hierarchy is used). On the first call of the flattening function, an empty path is passed.

Algorithm 4 Flattening algorithm

```
1: function flatten(ssm, path) returns the flattened version of a hierarchical system model
2:     extendSymbols(ssm, path);
3:     $\mathrm{c_f} = \bigcup_{\mathrm{name} \in \mathit{Domain}(\mathrm{ssm.Comp})} \Big\{\mathrm{flatten}\big(\mathrm{ssm.Comp(name), concat(path, name)}\big)\Big\}$;
4:     $\mathrm{fssm} = \Big(\mathrm{ssm.Var} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.Var}), \mathrm{ssm.Event} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.Event}),$
           $\mathrm{ssm.Port} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.Port}), \mathrm{ssm.TS}_{int} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.TS}_{int}),$
           $\mathrm{ssm.TS}_{ext} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.TS}_{ext}), \mathrm{ssm.Conn} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.Conn}),$
           $\mathrm{ssm.PA} \cup (\bigcup_{\mathrm{c}\in\mathrm{c_f}} \mathrm{c.PA})\Big)$;
5:     return fssm
6: end function
```

4.4.2 Transition System Semantics

Like program graphs or timed automaton, *ssm*s respectively *fssm*s can be interpreted as a transition system (see Section 2.3.1). The transition system results from the unfolding function described in Algorithm 5. The transition system of a *ssm* differs from the transition system of a program graph in the following aspects: In the transition system of a program graph, a state is characterized by the location and an evaluation of the variables. In smartIflow, the states of the underlying transition system consist of an evaluation η_v of the variables, the trigger conditions of all transition statements for internal events of the previous state ($\text{state}_{ts_{\text{prev}}}$), and the set of delayed transition statements for internal events ($\text{state}_{ts_{\text{del}}}$). States are thus triples of the form $(\eta_v, \text{state}_{ts_{\text{prev}}}, \text{state}_{ts_{\text{del}}})$. This modification is necessary for the rising edge semantics and the delay values of the transition statements. Compared to the transition system of a program graph, the transition relation in smartIflow is quite different. The transition relation of a program graph describes how the system changes its locations and the actions that are performed when a location is entered. In smartIflow the transition relation relies on internally and externally triggered transition statements, rising edge semantics, delays, and stable conditions describing how variables change their values. Each transition in the transition system is labeled with a delay value and a symbol of the event that triggered the transition. ι is a symbol that is used to label a transition that originates from internally triggered transition statements. A transition that originates from an externally triggered transition statement is labeled with the according symbol of the external event. I denotes the initial state in which all variables are assigned their initial value, and both, $\text{state}_{ts_{\text{del}}}$ and $\text{state}_{ts_{\text{prev}}}$, are empty. AP is the set of atomic propositions and L is the labeling function. Function *Behavior*(.) is responsible for establishing the connection structure and propagation of properties along the network structure (processing behavior will be described in detail in Section 4.5.2).

Definition 4.4.3 Transition System Semantics of a smartIflow System Model
The transition system $TS(\mathit{fssm})$ of a flattened smartIflow system model

$$\mathit{fssm} = (\mathit{Var}, \mathit{Init}, \mathit{Event}, \mathit{Port}, \mathit{TS}_{\text{int}}, \mathit{TS}_{\text{ext}}, \mathit{Conn}, \mathit{PA})$$

is the tuple $(S, I, AP, L, \longrightarrow)$ where

- $S = \mathit{Eval}(\mathit{Var}) \times \text{State}_{ts} \times \text{State}_{ts}$
  where $\text{State}_{ts} = 2^{TS_{\text{int}}}$
- $I = (\mathit{Init}, \emptyset, \emptyset)$
- $AP = \mathit{Cond}(\mathit{Pred}(\mathit{Var}))$
- $L : S \to 2^{AP}$ where
  $$L((\eta_v, \text{state}_{ts_{\text{prev}}}, \text{state}_{ts_{\text{del}}})) = \{c_v \in \mathit{Cond}(\mathit{Pred}(\mathit{Var})) \mid \eta_v \models c_v\} \cup \{c_p \in \mathit{Cond}(\mathit{Pred}(\mathit{Port})) \mid \mathit{Behavior}(\eta_v, \mathit{fssm}) \models c_p\}$$
- $\longrightarrow \subseteq (S \times S \times \mathbb{N}_0 \times E)$ defines the transition relation
  let

- $s \in S$, $s' \in S, d \in \mathbb{N}_0$ and $e \in \mathit{Event} \cup \{\iota\}$
- $s \xrightarrow{d}_e s' \iff (s, s', d, e) \in\longrightarrow$
- $(\eta_v, \text{state}_{ts_\text{prev}}, \text{state}_{ts_\text{del}}) = s$ and $(\eta_v{}', \text{state}_{ts_\text{prev}}{}', \text{state}_{ts_\text{del}}{}') = s'$
- $\eta_p = \mathit{Behavior}(\eta_v, \mathit{fssm})$ *//Available properties at ports after behavior execution*
- $ts_{ho} = \{ts_i \mid ts_i \in TS_\text{int} \wedge\ \eta_v, \eta_p \models ts_i.\,\text{c}_\text{t}\}$ *//Transition statements whose condition are fulfilled*
- $ts_{tr} = \{ts_i \mid ts_i \in ts_{ho} \wedge ts_i \notin \text{state}_{ts_\text{prev}}\} \cup \text{state}_{ts_\text{del}}$ *//All transition statements, which are triggered in s or were triggered before and were not executed yet*
- $ts_{tr_g} = \{ts_i \mid ts_i \in ts_{tr} \wedge \eta_v \models ts_i.\,\text{c}_\text{g}\}$ *//Transition statements whose guard conditions are fulfilled*
- $t_{min} = \min\Big(\bigcup_{ts_i \in ts_{tr_g}} \{ts_i.\Delta\} \cup \bigcup_{ts_i \in ts_{tr_g}} \{ts_i.\varsigma\}\Big)$ *//Minimum of all delay and stable values from all transition statements in ts_{tr_g}*
- $ts_{ho_{ns}} = \{ts_i \mid ts_i \in ts_{ho} \wedge \neg exists(ts_i.\varsigma)\}$ *//All transition statements of ts_{ho} for which no stable value is defined*
- $ts_{del} = \{ts_i \mid ts_i \in ts_{tr_g} \wedge ts_i.\Delta > t_{min}\}$ *//Delayed transition statements*
- $ts_{ex} = \{ts_i \mid ts_i \in ts_{tr_g} \wedge (ts_i.\Delta == t_{min} \vee ts_i.\varsigma == t_{min})\}$ *//Internally triggered transition statements that are executable based on their stability or delay values*
- $ts_{ext_{ho}} = \{ts_i \mid ts_i \in TS_\text{ext} \wedge ts_i.\varsigma \leq t_{min} \wedge \eta_v \models ts_i.\,\text{c}_\text{g}\}$ *//Admissible externally triggered transition statements*

the internal transition $(\eta_v, \text{state}_{ts_\text{prev}}, \text{state}_{ts_\text{del}}) \xrightarrow{d}_e (\eta_v{}', \text{state}_{ts_\text{prev}}{}', \text{state}_{ts_\text{del}}{}')$ holds iff

a) $\text{state}_{ts_\text{prev}}{}' == ts_{ho_{ns}}$ and
b) $\text{state}_{ts_\text{del}}{}' == ts_{del}$ and
c) $\eta_v{}' \in \mathit{Effect}(\eta_v, ts_{ex})$ and
d) $d == t_{min}$
e) $e == \iota$

the external transition $(\eta_v, \text{state}_{ts_\text{prev}}, \text{state}_{ts_\text{del}}) \xrightarrow{d}_e (\eta_v{}', \text{state}_{ts_\text{prev}}{}', \text{state}_{ts_\text{del}}{}')$ holds iff

a) $\exists ts_i \in ts_{ext_{ho}}.\, \eta_v{}' \in \mathit{Effect}(\eta_v, e)$ and
b) $\text{state}_{ts_\text{prev}}{}' == ts_{ho_{ns}}$ and
c) $\text{state}_{ts_\text{del}}{}' == ts_{del} \cup ts_{ex}$ and
d) $d == ts_i.\varsigma$ and
e) $e == ts_i.event$

Finally, the transition relation $\longrightarrow$ is defined as the union of all internal and external transitions.

$TS(ssm)$ is finite if S, AP, and $\rightarrow$ are finite, too. Note, that we write $s \xrightarrow{d}_e s'$ instead of $(s, s', d, e) \in\longrightarrow$. ■

In simplified form, the behavior of a transition system can be described as follows: The transition system of a smartIflow system model starts in the initial state $s_0 = I$. For s_0 the transition system evolves according to the transition relation $\longrightarrow$ depending on the property evaluations at ports which is the result of function $\mathit{Behavior}(.)$. From state s all admissible transitions $s \xrightarrow{d} s'$ are taken, and the procedure is repeated in state s'. The procedure terminates if in a state s' no outgoing transition statements are available. Function $\mathit{Effect}(.)$ combines effects of a set of internally or externally triggered transition statements leading to a set of successor states.

Definition 4.4.4 Direct Successors and Predecessors
The set of direct *successors* for $s \in S$ is defined as:

$$\mathit{Post}(s) = \{s' \in S \mid \exists d, e : s \xrightarrow{d}_e s'\}$$

Accordingly, the set of *predecessors* for $s \in S$ is defined by:

$$\mathit{Pre}(s) = \{s' \in S \mid \exists d, e : s' \xrightarrow{d}_e s\}$$

Let $Q \subseteq S$ subset of S. In this case, the sets of direct successors are defined in the following way:

$$\mathit{Post}(Q) = \bigcup_{s \in Q} \mathit{Post}(s)$$

The sets of predecessors for $Q \subseteq S$ can be defined similarly:

$$\mathit{Pre}(Q) = \bigcup_{s \in Q} \mathit{Pre}(s)$$

■

Definition 4.4.5 Terminal States
Terminal states in $TS(\mathit{fssm})$ are states without outgoing transitions. In other words, $s \in S$ is a terminal state iff $\mathit{Post}(s) = \emptyset$. ■

Definition 4.4.6 Delays and Stability Values in Transition Systems
Each transition $s \xrightarrow{d} s'$ from the transition relation is assigned an order of magnitude delay value. Function $\mathit{Delay}(s, s')$ enables access to the delay value of a transition in the following way:

$$\mathit{Delay}(s, s') = d \text{ iff } s, s' \in S,\ s' \in \mathit{Post}(s), \text{ and } (s, s', d) \in s \xrightarrow{d}_\iota s'$$

Each state $s \in S$ in a transition system $TS(\mathit{fssm})$ is stable with respect to a certain OM value. The stability value of a state $s \in S$ can be determined using function $\mathit{Stable}(s)$. Formally, $\mathit{Stable}(s)$ is defined as follows:

$$\mathit{Stable}(s) = \min\Bigl(\Bigl\{\exists s' \in \mathit{Post}(s) : s \xrightarrow{d}_\iota s'\Bigr\} \cup \infty\Bigr)$$

Terminal states are stable with respect to OM value of ∞ (above reference). ■

4.5 Unfolding Algorithm

As already mentioned, the transition system of a smartIflow system model can be obtained by unfolding. Algorithm 5 shows an implementation of an unfolding function that creates an explicit representation of the transition system for a smartIflow system model (in the following called system).

Algorithm 5 The unfolding function

```
function unfold(system) returns a node (=initial state of the transition system)
    create $\mathrm{s_{init}}$ with $\mathrm{state_{ts_{prev}}(s_{init})} = \{\}$, $\mathrm{state_{ts_{del}}(s_{init})} = \{\}$,
        $\mathrm{state_{port}(s_{init})} = \{(p, \{\}) \mid p \in \mathit{Port}\}$,
        and $\mathrm{state_{var}(s_{init})}$ defined by initial variable values;  // Initial state
    queue = [];  // Queue of not yet processed state nodes
    lookup = {};  // Map, which maps processed states to corresponding nodes
    root = createNode($\mathrm{s_{init}}$);
    lookup.addEntry($\mathrm{s_{init}}$, root);
    queue.add(root);

    while queue not empty do
        $\mathrm{n_{curr}}$ = queue.remove();
        $\mathrm{s_{curr}}$ = state($\mathrm{n_{curr}}$);

         // establish connection structure, set properties to ports and propagate them
        $\mathrm{state_{port}(s_{curr})}$ = processBehavior(system, $\mathrm{state_{var}(s_{curr})}$);

         // process transition statements
        successors = processEvents(system, $\mathrm{n_{curr}}$);

        for each (events, $\mathrm{s_{succ}}$) in successors do
            if $\mathrm{s_{succ}}$ not in lookup then
                $\mathrm{n_{succ}}$ = createNodeFor($\mathrm{n_{curr}}$, events, $\mathrm{s_{succ}}$);
                queue.add($\mathrm{n_{succ}}$);
                lookup.addEntry($\mathrm{s_{succ}}$, $\mathrm{n_{succ}}$);
            else
                createReferenceFor($\mathrm{n_{curr}}$, events, lookup.get($\mathrm{s_{succ}}$));
            end if
        end for
    end while
    return root;
end function
```

The resulting transition system is a directed graph of nodes, where the root node refers to the initial state, and successor nodes refer to successor states. The graph is directed and rooted since in smartIflow there is only one initial state. The initial state is defined by the default variable values. The unfolding function returns only the root node (initial state), as all subsequent states can be reached by traversing the graph. Each node consists of a model state and child nodes that refer to successor states. Nodes can also be marked as stable with respect to an order of magnitude value. A model state $s = state(node)$ is characterized by four components:

- $\mathrm{state}_{\mathrm{var}}(\mathrm{s})$ describes the value of all variables. It maps each variable to its current value.
- $\mathrm{state}_{\mathrm{ts}_{\mathrm{prev}}}(\mathrm{s})$ describes the truth values of all transition conditions without guards of the parent state. It is a set and contains all fulfilled internally triggered transition statements.
- $\mathrm{state}_{\mathrm{ts}_{\mathrm{del}}}(\mathrm{s})$ is a set of transition statements and describes the delayed transition statements.
- $\mathrm{state}_{\mathrm{port}}(\mathrm{s})$ describes the properties of all ports. Each port is mapped to the set of (key, value) pairs that are currently present at the port. When initializing a new state, each port is mapped to the empty set.

$\mathrm{state}_{\mathrm{port}}(\mathrm{s})$ is determined by $\mathrm{state}_{\mathrm{var}}(\mathrm{s})$ (conditional behavior is executed and properties are propagated). Therefore, only $\mathrm{state}_{\mathrm{var}}(\mathrm{s})$, $\mathrm{state}_{\mathrm{ts}_{\mathrm{del}}}(\mathrm{s})$, and $\mathrm{state}_{\mathrm{ts}_{\mathrm{prev}}}(\mathrm{s})$ are used to decide whether two states are equal or not.

The unfolding function utilizes a queue and a lookup table. The queue manages not yet processed nodes and the lookup table maps states to corresponding nodes. The latter is necessary to avoid multiple analyses of the same state. The unfolding function first determines the initial state where all state variables are set to their default value. For each node in the queue, first, the behavior is processed, and afterward, subsequent states are determined. The function terminates once there are no more nodes in the queue.

The following two sections explain the processing of internal and external events and the behavior in more detail.

4.5.1 Process Events

Internally and externally triggered transition statements are processed by use of function processEvents (used in function unfold(system) in line 17). The result of function processEvents is a set of (event-list, state) pairs covering all possible subsequent states. Note that the states are not yet completely computed. $\mathrm{state}_{\mathrm{var}}$, $\mathrm{state}_{\mathrm{ts}_{\mathrm{del}}}$, and $\mathrm{state}_{\mathrm{ts}_{\mathrm{prev}}}$ are computed, but not $\mathrm{state}_{\mathrm{port}}$. $\mathrm{state}_{\mathrm{port}}$ is computed by the function processBehavior. The lookup table is used to check whether a successor has already been explored. For already explored successors a reference is used instead of a parent-child relationship. Only successors that are not yet part of the lookup are added to the queue and processed further. References and parent-child links are labeled with the events that caused the state change. This makes it possible to distinguish whether a transition was caused by internal or external events.

A possible implementation of function processEvents is shown in Algorithm 6. First, $\mathrm{ts}_{\mathrm{ho}}$ the set of all internally triggered transition statements whose conditions hold in the current state ($\mathrm{state}(\mathrm{n}_{\mathrm{curr}})$) is determined. $\mathrm{ts}_{\mathrm{tr}}$ contains delayed transition statements from

Algorithm 6 Process internally and externally triggered transition statements

```
1: function processEvents(system, $n_{curr}$) returns a list of (event-list, state) pairs
2:     $s_{curr}$ = state($n_{curr}$);
3:     result = {};
4:     $ts_{ho}$ = all internally triggered transition statements in system which hold in
           $state_{var}(s_{curr})$, $state_{port}(s_{curr})$;
5:     $ts_{tr} = (ts_{ho} - state_{ts_{prev}}(s_{curr})) \cup state_{ts_{del}}(s_{curr})$;
6:     test guards of all transition statements in $ts_{tr}$ and remove transition statements
           of unfulfilled guards in $state_{var}(s_{curr})$, $state_{port}(s_{curr})$;
7:     $t_{min}$= minimum of all delay and stable values of transition statements in $ts_{tr}$,
           infinity if $ts_{tr}$ == {};
8:     remove all transition statements from $ts_{ho}$ where stable exists;
9:     remove all transition statements from $ts_{tr}$ where stable > $t_{min}$;
10:    $ts_{ac}$ = transition statements from $ts_{tr}$ with delay = $t_{min}$ or stable = $t_{min}$;
11:    $ev_{ho}$ = externally triggered transition statements for $n_{curr}$ with stable <= $t_{min}$;
12:    mark $n_{curr}$ as stable with respect to $t_{min}$;
13:
14:    if $ev_{ho}$ exist then
15:        // process possible externally triggered transition statements (external events)
16:        for each e in $ev_{ho}$ do
17:            assignments = computeNextStateAssignments(system, $s_{curr}$, {e});
18:            for each assignment in assignments do
19:                create $s_{succ}$ with $state_{var}(s_{succ})$ = assignment, $state_{ts_{prev}}(s_{succ}) = ts_{ho}$,
                       $state_{ts_{del}}(s_{succ}) = ts_{tr}$;
20:                result.add((list(e), $s_{succ}$));
21:            end for
22:        end for
23:    end if
24:
25:    if $ts_{ac}$ exist then
26:        // process internally triggered transition statements (internal events)
27:        assignments = computeNextStateAssignments(system, $s_{curr}$, $ts_{ac}$);
28:        for each assignment in assignments do
29:            create $s_{succ}$ with $state_{var}(s_{succ})$ = assignment, $state_{ts_{prev}}(s_{succ}) = ts_{ho}$,
                   $state_{ts_{del}}(s_{succ}) = ts_{tr} - ts_{ac}$;
30:            result.add((collectInternalEvents($ts_{tr}$), $s_{succ}$));
31:        end for
32:    end if
33:    return result
34: end function
```

the parent state as well as all transition statements from ts_{ho} that follow the rising edge semantics. Next, guard conditions are checked. Only transition statements for which the guard condition is satisfied remain in ts_{tr}. Afterward, the transition statements that may be executed with respect to their stable or delay value are identified. t_{min} is the minimum of all delay and stable values from transition statements in ts_{tr}. The current node n_{curr} is marked as stable with respect to t_{min}. Note that all transition statements with specified stable value are removed from ts_{ho} (see line 8). This is necessary because the rising edge semantics is only applied to transition statements without a stable value. In addition, all transition statements in ts_{tr} for which a stable condition is defined and not fulfilled are deleted (see line 9). This prevents inactive transition statements (i.e., transition statements where the stable condition is not fulfilled) from being delayed. Finally, ts_{ac} contains the transition statements that are executed. ev_{ho} is the set of externally triggered transition statements that are admissible to be executed with respect to t_{min}.

After that, the successor states can be determined. First, the successor states for each externally triggered transition statement are determined (only one external event is executed at the same time). Subsequently, all internally triggered transitions are processed. Transition statements that are in ts_{tr} but not in ts_{ac} are delayed. This corresponds to all transition statements whose trigger and guard conditions are true but may not yet be executed due to their delay values.

Internally and externally triggered transition statements allow nondeterministic variable value updates. Therefore, the execution of an internally or externally triggered transition statement can lead to multiple subsequent states. Function computeNextStateAssignments (see Algorithm 7) computes all next state assignments for a set of transition statements (ts).

Algorithm 7 Computation of next state assignments

```
1: function computeNextStateAssignments(system, s, ts)
          returns a set of (variable, value) maps
2:    result = {$state_{var}(s)$};
3:    compute a (variable, value-set)-map $map_{var}$ using the assignments belonging to ts;
4:    for each entry ($variable_{curr}$, $value\text{-}set_{curr}$) $\in$ $map_{var}$ do
5:        $result_{new}$ = {};
6:        for each $value_{curr}$ $\in$ $value\text{-}set_{curr}$ do
7:            for each assignmentMap $\in$ result do
8:                $assignmentMap_{copy}$ = create copy of assignmentMap;
9:                update $variable_{curr}$ in $assignmentMap_{copy}$ by $value_{curr}$;
10:               $result_{new}$.add($assignmentMap_{copy}$);
11:           end for
12:       end for
13:       result = $result_{new}$;
14:   end for
15:   return result;
16: end function
```

Example 4.5.1 Unfolding example
Consider a simple system consisting of a controller and a valve. The controller is responsible for opening and closing the valve periodically. For this purpose, the controller sends a

control signal to the valve. Depending on the control signal, the valve is opened or closed. This kind of control logic is needed, for example, in watering systems. The following listing shows the component model of the controller in the smartIflow language.

```
class Controller {
    Ports:
        Logical vCtrl;

    Variables:
        Enum[Idle, Supply] opm = Idle;

    Behavior:
        if (opm == Idle)
            set(vCtrl, {cmd=close});
        if (opm == Supply)
            set(vCtrl, {cmd=open});

    Transitions:
        when {delay=2, name=ts_supply} (opm == Idle)
            opm = Supply;
        when {delay=2, name=ts_idle} (opm == Supply)
            opm = Idle;
}
```

The valve can be modeled as follows:

```
class Valve {
    Ports:
        Logical ctrl;

    Variables:
        Enum[Opened, Closed] opm = Closed;

    Transitions:
        when {delay=1, name=ts_close} (ctrl.exists(cmd=close)) [opm == Opened]
            opm = Closed;
        when {delay=1, name=ts_open} (ctrl.exists(cmd=open)) [opm == Closed]
            opm = Opened;
}
```

In class `Main` both component types are instantiated and connections are established.

```
class Main {
    Components:
        Controller contr;
        Valve valve;
    Behavior:
        connect(contr.vCtrl, valve.ctrl);
}
```

The control signal for the valve is modeled using the properties `cmd=open` to open and `cmd=close` to close the valve. The transition statements `ts_close` and `ts_open` in class `Valve` specify the reaction to changes on the control signal. The controller constantly toggles between the states `Idle` and `Supply` using the transition statements `ts_idle` and `ts_supply`. The behavioral description of the controller is quite simple. The controller publishes either property `cmd=open` or `cmd=close` depending on its current state (defined by the `opm` variable). Note that two different OM values are used for the delays of the transition statements. The valve reacts to changes on the control signal with a delay of OM value of one (some milliseconds). The controller changes its state with a delay of OM value of two (some minutes).

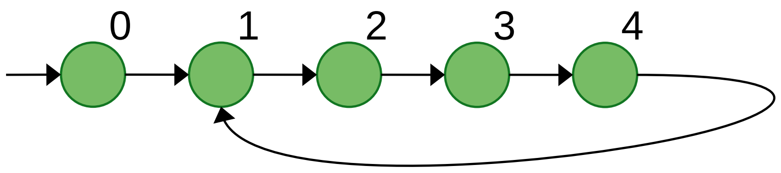


Figure 4.6: Transition system obtained by unfolding represented as directed rooted graph

Figure 4.6 shows the transitions system represented as directed rooted graph obtained by unfolding. For the sake of clarity, each state is assigned a unique number. Table 4.1 gives an overview about the variable values, properties at ports, triggering states, delayed transition statements, and stable values of the states of the transition system.

	$\text{state}_{\text{var}}(s)$		$\text{state}_{\text{port}}(s)$		$\text{state}_{\text{ts}_{\text{del}}}(s)$				$\text{state}_{\text{ts}_{\text{prev}}}(s)$				
State	contr.opm	valve.opm	contr.vCtrl	valve.ctrl	ts_supply	ts_idle	ts_close	ts_open	ts_supply	ts_idle	ts_close	ts_open	**Stable**
0	Idle	Closed	cmd=close	cmd=close	0	0	0	0	0	0	0	0	2
1	Supply	Closed	cmd=open	cmd=open	0	0	0	0	1	0	1	0	1
2	Supply	Opened	cmd=open	cmd=open	0	1	0	0	0	1	0	1	2
3	Idle	Opened	cmd=close	cmd=close	0	0	0	0	0	1	0	1	1
4	Idle	Closed	cmd=close	cmd=close	1	0	0	0	1	0	1	0	2

Table 4.1: States of the transition system obtained by unfolding

The unfolding function first determines the initial state (variables are set to their default values, and both, $\text{state}_{\text{ts}_{\text{prev}}}$ and $\text{state}_{\text{ts}_{\text{del}}}$ are empty). After that, the behavior is executed and the next states are determined: In the initial state (state 0), the triggering conditions of transition statements ts_supply and ts_close are fulfilled. However, only the transition statement ts_supply is executed because the guard condition of ts_close is false. The execution of ts_supply leads to a new successor state (state 1). $\text{state}_{\text{ts}_{\text{prev}}}$ of state 1 contains ts_supply and ts_close because their triggering conditions were satisfied in the previous state. After the prediction of behavior in state 1, the possible successor states can be determined. In state 1, the triggering conditions of ts_idle and ts_open are fulfilled and the corresponding guard conditions are fulfilled. ts_idle has a delay of OM value of 2 and ts_open has a delay of OM value of 1. Therefore, ts_open is executed immediately and ts_idle is delayed (ts_idle is added to $\text{state}_{\text{ts}_{\text{del}}}$ of the successor states). The execution of ts_open leads to one successor state (state 2). This procedure is repeated until no new successor state is discovered. In our example, this occurs in state 4. The successor state that results from the execution of ts_supply in state 4 is identical to state 1. Therefore, state 1 is referenced and the unfolding procedure terminates.

■

The example system just shown was rather simple (there were only deterministic variable value changes and no external events). The obtained transition system was also simple, consisting only of a sequence of states with a reference back to the second state. With more complex systems, probably there will be many branches and overlapping edges. The latter leads to the problem that the graphical representation of the transition systems as a directed graph can become very confusing. To counteract this problem, an alternative representation is needed. Figure 4.7 shows two different representations for the transition system of the diagonal car braking system (see Section 6.1). The representation as a directed graph (left part) contains many overlapping edges. In the representation as a tree structure (right part), references to already existing states are represented by reference nodes (highlighted in blue). This results in a tree structure without overlapping edges. The root of the tree corresponds to the initial state.

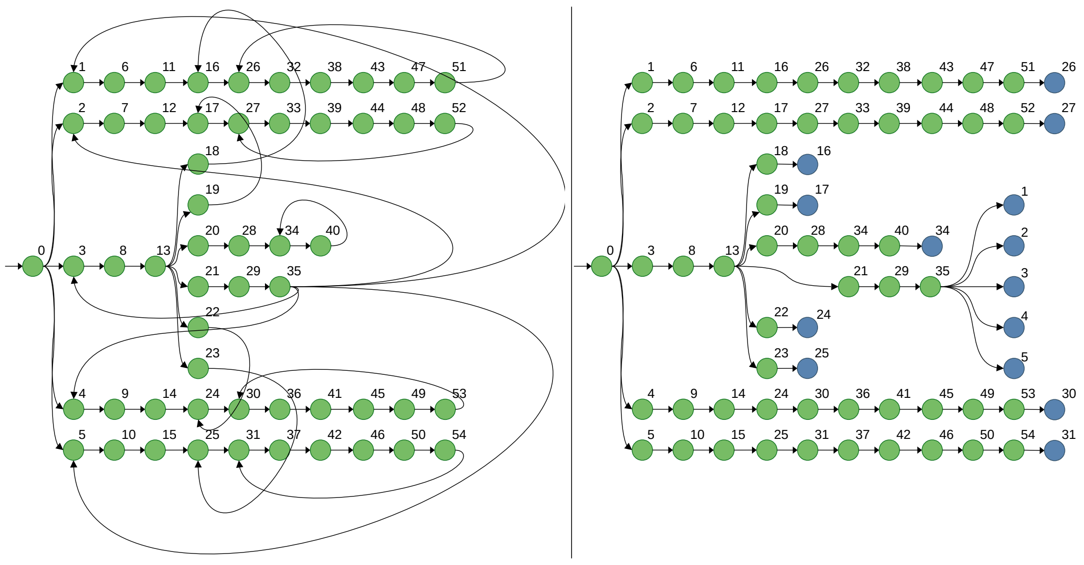


Figure 4.7: Transition system represented as a directed graph and in a tree structure

4.5.2 Process Behavior

Function processBehavior (used in function unfold(system) in line 14) is responsible for the construction of the connection structure, adding properties to ports, and propagation of properties published by the components to all reachable ports. Building the connection structure is simple: Collect all conditional connection statements from each component in the model, determine the set of active connections based on the current variable evaluation, and finally execute them. Before propagation, all active conditional property assignments to ports must be executed. In principle, conditional property assignments to ports can be performed in the same way as the connection structure setup. First, all active property assignments based on the current variable evaluation are determined. This set contains all property assignments for which the corresponding condition is fulfilled. Each conditional property assignment publishes a set of properties to a port. Properties can be published to ports of its own component as well as to the ports of subcomponents. As information flow at ports can be restricted to inputs or outputs (see Section 4.2), properties on a port can be quite different depending on the point of view (access outside or inside). Therefore, for each port, the properties accessible inside and outside the component must be stored separately. In the next step, the properties at each port must be propagated to all reachable ports. Note that connections let the information flow in both directions, while ports can limit the flow of information to a certain direction (input/output). A very simple approach for the propagation of properties could look like this: For each port in a system, follow all paths to other connected ports (port directions must be considered!). During the traversal, collect the properties from the active conditional property assignments and the predecessor state and assign this set to every subsequent port. The traversal of a path can be stopped as soon as a cycle is detected or when a terminal port is reached. Basically, the algorithm performs for each port in a system a depth-first search and tries to propagate at each port all available properties to all other connected ports. All kinds of network topologies are supported and the algorithm is rather efficient ($O(|Ports|^2)$). However, there are scenarios where propagation can be done more elegantly and efficiently. Consider the connection structure shown in Figure 4.8. In this example, every port is reachable from every other port. Therefore, the properties at each port in this system are identical (the set of properties at each port is equal to the collection of all published properties). In this example, it is sufficient to collect the properties of each port and then assign the combination to each port. But how can such scenarios be detected?

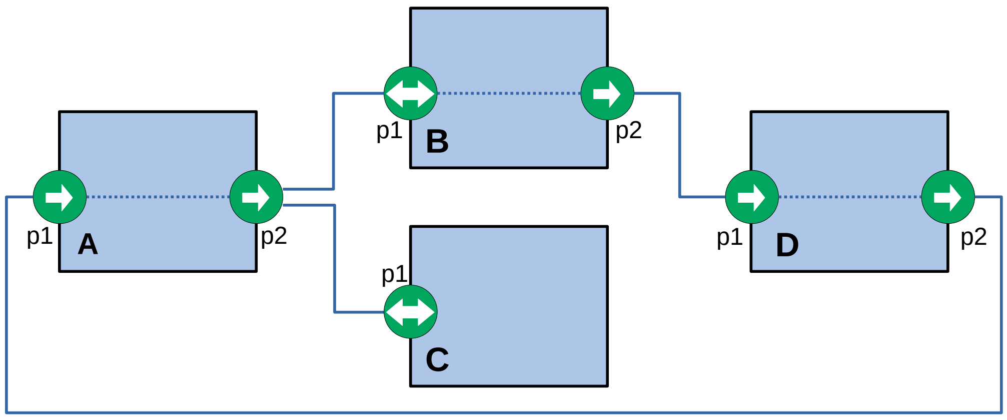


Figure 4.8: Connection structure example

As defined in literature, a graph is strongly connected if every vertex is reachable from every other vertex (see Definition 2.6). The topology of the connection structure shown in Figure 4.8 also fulfills this property (every port is reachable from every other port). Tarjan's algorithm can determine the strongly connected components in a directed graph (see Section 2.6). Unfortunately, the connectivity structure is neither a directed nor an undirected graph (connections are bidirectional, while ports can be directed). However, it is possible to represent the connection structure as a directed graph.

Definition 4.5.1 Connection Structure Representation as a Directed Graph
In principle, the connection structure of a smartIflow system model can be interpreted as a directed graph $G = (V, E)$ where V is a set of vertices and E is a set of edges. Every (bidirectional) connection between two ports A and B is represented by two edges $A \rightarrow B$ and $B \rightarrow A$. Each port $p \in \mathit{fssm.Port}$ is represented by two artificial vertices (p_{in} and p_{out}). If P is an input port, an edge $P_{out} \rightarrow P_{in}$ is created. If P is an output port, an edge $P_{in} \rightarrow P_{out}$ is created. Otherwise, (unrestricted port) two edges are created. ■

The idea of an improved behavior prediction algorithm is quite simple: First, the connection structure is converted into a directed graph (linear in the number of ports and connections: $O(|Ports| + |Conns|)$). Afterward, all strongly connected components in this graph are determined using Tarjan's algorithm ($O(|V| + |E|)$). As already mentioned, the properties at all ports that are part of strongly connected components are identical. The set of properties of a strongly connected component is defined by the execution of all active conditional property assignments to ports that are part of the strongly connected component. Strongly connected components can be linked via edges. If strongly connected components are connected by an edge, properties must be propagated. We will denote the edges between strongly connected components as *bridges*. Bridges between the strongly connected components can result in arbitrarily directed acyclic graphs, but never in a cyclic graph, since this would have been detected by Tarjan's algorithm. In fact, these bridges between strongly connected components are always based on restricted ports (i.e., input and output ports). In order to perform the information exchange between strongly connected components, each strongly connected component is replaced with a single vertex. This condensation results in a directed acyclic graph. In the last step, properties between vertices in this graph must be propagated. First, the vertices without predecessors are determined and then the properties are propagated to all other reachable edges. Instead of writing the properties to the ports themselves, the properties of a port are stored in its corresponding strongly connected component. Note that every vertex that is not part of a strongly connected component already forms a strongly connected component itself.

Example 4.5.2 Propagation of properties
The example shown in Figure 4.9 illustrates the three steps of behavior prediction (directed graph construction, determination of strongly connected components, and propagation of properties between strongly connected components). In step 1, the connection structure is converted into a directed graph. Connections are replaced by two opposing edges. For each port, two vertices are created. Depending on the port type, one or two edges between each pair of vertices are created (input/output port = one edge; bidirectional port = two edges). In step 2, strongly connected components are determined using Tarjan's algorithm. The

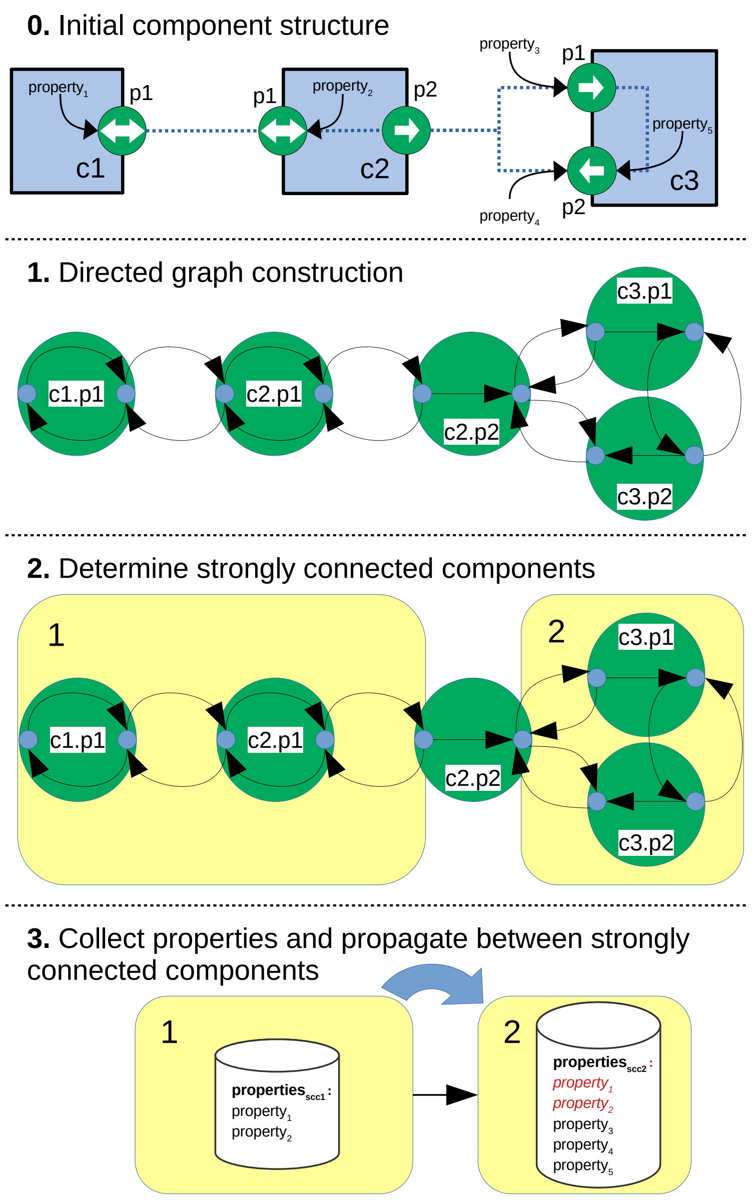


Figure 4.9: Property propagation example

connection structure in the example system contains two strongly connected components, with port `p2` of component `c2` acting as a bridge between them. In the last step, the properties between the strongly connected components are propagated. ■

The translation of a connection structure into a directed graph is straightforward. Algorithm 8 shows a possible conversion implementation. The algorithm utilizes lookup tables that map ports to corresponding vertices. $\text{portMap}_{\text{in}}$ maps ports to vertices, that represent the connection point inside the component of the port, and $\text{portMap}_{\text{out}}$ maps ports to vertices, that represent the connection point outside the component of the port.

Algorithm 8 Converting the connection structure into a directed graph

Input: Ports of a smartIflow system model ($\text{ports}_{\text{ssm}}$) and conditional connections whose conditions are fulfilled ($\text{conns}_{\text{active}}$)

Output: The tuple (adjacencyList, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$) where adjacencyList is the directed graph G = (E,V) represented as adjacency list, and the maps $\text{portMap}_{\text{in}}$ and $\text{portMap}_{\text{out}}$ that map ports to vertices

```
function constructDirectedGraph($\text{ports}_{\text{ssm}}$, $\text{conns}_{\text{active}}$)
    $\text{portMap}_{\text{in}}$= create empty map;
    $\text{portMap}_{\text{out}}$= create empty map;
    adjacencyList[ ] = create empty map;

    for each $\text{port}_{\text{i}} \in \text{ports}_{\text{ssm}}$ do
        $v_{p_{in}}$= create vertex for connections linked to $\text{port}_{\text{i}}$ inside;
        $v_{p_{out}}$= create vertex for connections linked to $\text{port}_{\text{i}}$ outside;
        adjacencyList.addEntry($v_{p_{in}}$, {});
        adjacencyList.addEntry($v_{p_{out}}$, {});
        $\text{portMap}_{\text{in}}$.addEntry($\text{port}_{\text{i}}$, $v_{p_{in}}$);
        $\text{portMap}_{\text{out}}$.addEntry($\text{port}_{\text{i}}$, $v_{p_{out}}$);
        if $\text{port}_{\text{i}}$ is of type output then
            adjacencyList.get($v_{p_{in}}$).add($v_{p_{out}}$);
        else if $\text{port}_{\text{i}}$ is of type input then
            adjacencyList.get($v_{p_{out}}$).add($v_{p_{in}}$);
        else
            adjacencyList.get($v_{p_{out}}$).add($v_{p_{in}}$);
            adjacencyList.get($v_{p_{in}}$).add($v_{p_{out}}$);
        end if
    end for

    for each $\text{conn}_{\text{i}} \in \text{conns}_{\text{active}}$ do
        $v_{p_1}$= getVertex($\text{conn}_{\text{i}}$, $\text{conn}_{\text{i}}$.p1, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$);
        $v_{p_2}$= getVertex($\text{conn}_{\text{i}}$, $\text{conn}_{\text{i}}$.p2, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$);
        adjacencyList.get($v_{p_1}$).add($v_{p_2}$);
        adjacencyList.get($v_{p_2}$).add($v_{p_1}$);
    end for
    return (adjacencyList, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$);
end function

function getVertex($\text{conn}_{\text{i}}$, $\text{port}_{\text{i}}$, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$)
    if $\text{conn}_{\text{i}}$ is linked inside to $\text{port}_{\text{i}}$ then
        return $\text{portMap}_{\text{in}}$.get($\text{port}_{\text{i}}$);
    else
        return $\text{portMap}_{\text{out}}$.get($\text{port}_{\text{i}}$);
    end if
end function
```

The directed graph resulting from the conversion is represented as an adjacency list. As for each port, two vertices are created, and the list contains at the end twice as many entries as there are ports. The algorithm first iterates over all ports in the model. For each port, two vertices are created and added to the adjacency list (lines 7-10). The lookup tables are also filled with the port and corresponding vertex (lines 11-12). Afterward, the type of the port is inspected. If the port is unrestricted, then each vertex is added as a neighbor to the opposite vertex. For input or output ports only one edge between both vertices is added (direction depends on port type). After that, each connection is replaced by two opposing edges (lines 23-28). Function getVertex is used to determine the corresponding vertex for the connection point of a connection.

Algorithm 9 shows a possible implementation for processing the behavior. Once all active connections and property assignments are determined (i.e., all conditional connections and conditional property assignments to ports whose condition is fulfilled), the connection structure is represented as a directed graph. Afterward, strongly connected components in this graph are determined using Tarjan's algorithm (described in Algorithm 3). The properties of each active conditional property assignment (pa) are added to the strongly connected component that contains the port of pa (lines 8-11). Afterward, each edge $e = (v, w)$ of the directed graph is analyzed. If the vertices v and w of edge e are in part of different strongly connected components, then edge e is a bridge. Therefore, the strongly connected component containing w is inserted in an adjacency list as the successor of the strongly connected component containing v. Function propagate recursively traverses the condensed graph and propagates properties between strongly connected components. The set sources is used to identify the strongly connected components without any predecessor. Initially, all strongly connected components from SCCs are inserted. Once a strongly connected component ($\mathrm{scc}_{\mathrm{dest}}$) is inserted as a successor into the adjacency list, $\mathrm{scc}_{\mathrm{dest}}$ is removed from set sources. Each port side exactly belongs to one strongly connected component. After propagation, the available properties of each port are available in the corresponding strongly connected component.

Algorithm 9 Process behavior

```
1: function processBehavior(system, $\text{state}_{\text{var}}(\text{s}_{\text{curr}})$) returns maps that represent the
   properties currently present at each port inside and outside
2:     $\text{PA}_{\text{active}}$= property assignments from $system.PA$ whose conditions are fulfilled;
3:     $\text{conns}_{\text{active}}$= connections from $system.Conn$ whose conditions are fulfilled;
4:
5:     (G, $\text{portMap}_{\text{in}}$, $\text{portMap}_{\text{out}}$) = constructDirectedGraph($system.Port$, $\text{conns}_{\text{active}}$);
6:     SCCs = tarjan(G); // *Determine strongly connected components*
7:
8:     for each pa $\in$ $\text{PA}_{\text{active}}$ do
9:         scc = find strongly connected component for $pa$;
10:        scc.addProperties(pa.properties);
11:    end for
12:
13:    SCCAdjacencyList = create map and add all entries of $\{(s, \{\})|s \in SCCs\}$;
14:    sources = create set of length size(SCCs) and add all entries of SCCs;
15:
16:    for each $(v, w)$ in G.edges do
17:        $\text{scc}_{\text{src}}$ = get strongly connected component for $v$ in SCCs;
18:        $\text{scc}_{\text{dest}}$ = get strongly connected component for $w$ in SCCs;
19:        if $\text{scc}_{\text{src}} \neq \text{scc}_{\text{dest}}$ then
20:            SCCAdjacencyList.get($\text{scc}_{\text{src}}$).add($\text{scc}_{\text{dest}}$);
21:            sources.remove($\text{scc}_{\text{dest}}$);
22:        end if
23:    end for
24:
25:    for each scc$\in$ sources do
26:        propagate(scc)
27:    end for
28:
29:    $\text{props}_{\text{inside}}$ = create empty map;
30:    $\text{props}_{\text{outside}}$ = create empty map;
31:    for each $\text{port}_{\text{i}}$ $\in$ $\text{ports}_{\text{ssm}}$ do
32:        $\text{scc}_{\text{pin}}$ = get strongly connected component for $\text{portMap}_{\text{in}}$.get($\text{port}_{\text{i}}$) in SCCs;
33:        $\text{scc}_{\text{pout}}$ = get strongly connected component for $\text{portMap}_{\text{out}}$.get($\text{port}_{\text{i}}$) in SCCs;
34:        $\text{props}_{\text{inside}}$.addEntry($\text{port}_{\text{i}}$, $\text{scc}_{\text{pin}}$.getProperties());
35:        $\text{props}_{\text{outside}}$.addEntry($\text{port}_{\text{i}}$, $\text{scc}_{\text{pout}}$.getProperties());
36:    end for
37:    return ($\text{props}_{\text{inside}}$, $\text{props}_{\text{outside}}$);
38: end function
39:
40: function propagate(scc)
41:    for each (scc, scc$'$) in SCCAdjacencyList.get(scc) do
42:        scc$'$.addProperties(scc.getProperties());
43:        propagate(scc$'$);
44:    end for
45: end function
```

4.6 Qualitative Energy Flow Analysis

Properties offer an extremely flexible mechanism for communication between components. In certain situations, however, the property concept will reach its limits. Consider the simple electrical circuit shown in Figure 4.10. The circuit consists of a voltage source, a resistor, and two thermistors. The resistance of a thermistor changes depending on the temperature and thus the current in the circuit changes too.

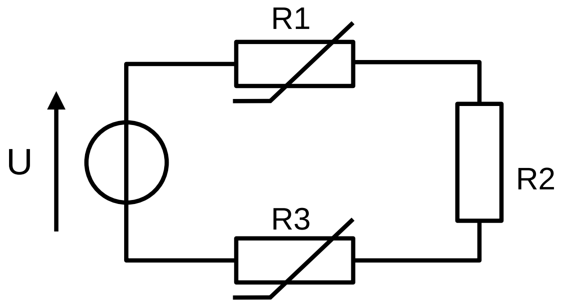


Figure 4.10: Simple electrical circuit with temperature-dependent resistors

Modeling the energy flows in this system by means of properties only is possible with some workarounds. One possible solution could look as follows: The resistors continuously send their resistance value to the voltage source (e.g., R1 sends `r1=high` for a high resistance value, R2 sends `r2=normal` for a normal resistance value, and R3 sends `r3=low` for a low resistance value). Depending on the received properties the voltage source decides about the current in the circuit. This is obviously not a good solution as the models become quite complex and this kind of modeling is anything but intuitive. Moreover, the behavior of the component models is no longer context-free. Last but not least, this solution is only applicable to small circuits.

A solution for this problem is the approach introduced by Snooke et al. [SL14] that enables qualitative reasoning about energy flows in power networks. But how can this approach be integrated into the smartIflow formalism? Basically, the smartIflow formalism has to be extended by the following aspects:

- The connections of the smartIflow system models must be assigned resistance values
- Special components for flow and effort sources are required
- Function processBehavior (see Algorithm 9) must be extended in such a way that the energy flows in the circuit (given by the connection structure) are calculated

Let us first discuss the resistance values of the power network. Connections in a smartIflow system model that are part of a power network need to be assigned a resistance value. Fortunately, the syntax of the smartIflow language does not have to be changed, since the resistance values can be assigned to the connections by use of the feature concept. The following listing shows the component class of a thermistor in the smartIflow language:

```
class Thermistor {
    Ports:
        Electrical p1;
        Electrical p2;

    Variables:
        Enum[Hot, Warm, Cold] temp = Warm;

    Events:
        HeatUp;
        CoolDown;

    Behavior:
        if(temp == Hot)
            connect{r=1}(p1,p2);
        if(temp == Warm)
            connect{r=2}(p1,p2);
        if(temp == Cold)
            connect{r=3}(p1,p2);

    EventHandlers:
        when(HeatUp)[temp==Cold]
            temp = Warm;
        when(HeatUp)[temp==Warm]
            temp = Hot;
        //...
}
```

The resistance of the connection between ports $p1$ and $p2$ changes depending on the temperature. For example, the feature `r=3` denotes a resistance that is three orders of magnitude above the reference value.

To create a current or effort source, two special properties must be added to the smartIflow component class. The first feature is used to define the type of the source (e.g., `flowComponent=bipolarFlowSource` denotes a bipolar flow source). smartIflow supports two types of sources namely bipolar effort sources and bipolar flow sources. The second feature defines the power of the sources. For example, the feature `om=1` defines that the power of the source is one order of magnitude above reference. Afterward, the source and drain of the source need to be defined. This requires two ports to be created first. The port that should act as source must be assigned the feature `flowPort=source` and the port that should act as drain must be assigned the feature `flowPort=drain`. The following smartIflow class shows the definition of a power supply based on a bipolar effort source:

```
class PowerSupply {flowComponent=bipolarEffortSource, om=1}{
    Ports:
        Electrical{flowPort=source} plus;
        Electrical{flowPort=drain} minus;
}
```

The qualitative energy flow analysis can easily be integrated into the unfolding process.

The function processBehavior (introduced in Section 4.5.2 first establishes the connection structure and then propagates the properties. After these two steps, the energy flows can be calculated by passing the model state (which includes the system model) to the function calculateFlows. A possible implementation of the function calculateFlows is shown in Algorithm 10.

Algorithm 10 Qualitative engery flow analysis

```
 1: function calculateFlows(modelState)
 2:     results = []; // List of flow analysis results
 3:     for each $source_i$ in getSources(network) do
 4:         $source_i$.setActive(false);
 5:     end for
 6:     for each $source_{curr}$ in getSources(network) do
 7:         $source_i$.setActive(true);
 8:         network = createPowerNetwork(modelState);
 9:         transformations = reduceNetwork(network);
10:         $source_{curr}$.assignFlow();
11:         while transformations not empty do
12:             $transformation_{curr}$ = transformations.pop();
13:             $transformation_{curr}$.expandAndAssignFlows(network);
14:         end while
15:         results.add(network);
16:         $source_i$.setActive(false);
17:     end for
18:     flowAnalyisResult = combineResults(results);
19:     modelState.setFlows(flowAnalyisResult);
20: end function
```

First, the power network is established (the vertices correspond to the ports and the edges to the connections). In the case of multiple sources, the flow analysis must be carried out separately for each source. For the analysis of one source, all other sources are inhibited (lines 5-10). In order to calculate the flows in the network, the network must be first reduced single equivalent resistance by applying the transformation rules described in Section 3.1.1. Each performed transformation is pushed to the stack transformations. After network reduction, the flow through the remaining edge between the source and drain can be calculated (line 13). After that, transformations are expanded back and at each level, the flows are assigned to the edges (lines 14-17).

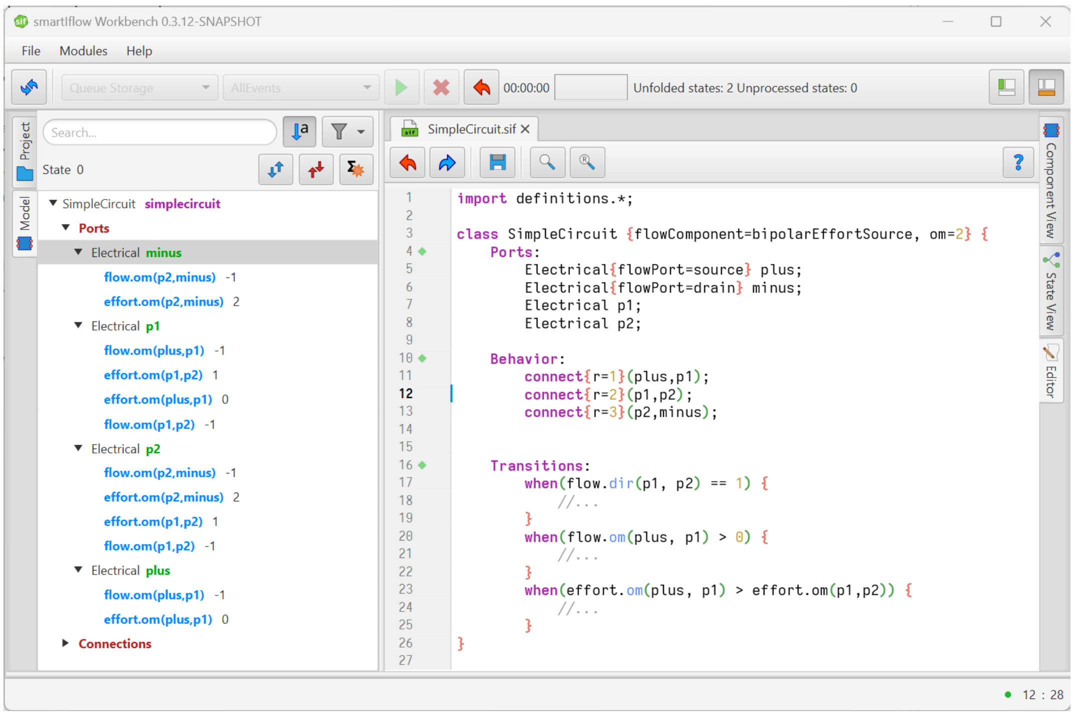
Finally, the results of the qualitative network flow analysis for each source are combined and added to the model state. The results (e.g., the flow through a connection or the effort across a connection) are stored in terms of simple properties at the corresponding ports.

Figure 4.11 shows the smartIflow Workbench with a system model of a simple electrical circuit (right part) and the corresponding results of the energy flow analysis (left part). The property `flow.om(p2,minus)` denotes a flow from port `p2` to port `minus` with one order of magnitude below reference. Similarly, the property `effort.om(p1,p2)` describes a voltage between ports `p1` and `p2` of one order of magnitude above the reference.

Of course, the energy flow analysis results can also be used in the logical expressions of transition statements of internally triggered events. For this purpose, there are some additional expressions. The expression `flow.dir(p1,p2)` can be used to determine the direction of the flow between the ports `p1` and `p2`. The return value is defined as follows:

- `1` denotes a positive flow (from `p1` to `p2`)
- `-1` denotes a flow in reverse direction (from `p2` to `p1`)
- `0` denotes that there is no flow or the flow direction is undetermined (due to ambiguities)

The expression `flow.om(p1,p2)` returns the order of magnitude of the flow through the connection between the ports `p1` and `p2`. If there is no flow, the expression returns `null`. The expressions `effort.om(p1,p2)` and `effort.dir(p1,p2)` behave similarly, except that here the effort across a connection is considered. The component class shown in Figure 4.11 demonstrates how these expressions can be used in transition statements.



```
▼ SimpleCircuit  simplecircuit
  ▼ Ports
    ▼ Electrical  minus
        flow.om(p2,minus)  -1
        effort.om(p2,minus)  2
    ▼ Electrical  p1
        flow.om(plus,p1)  -1
        effort.om(p1,p2)  1
        effort.om(plus,p1)  0
        flow.om(p1,p2)  -1
    ▼ Electrical  p2
        flow.om(p2,minus)  -1
        effort.om(p2,minus)  2
        effort.om(p1,p2)  1
        flow.om(p1,p2)  -1
    ▼ Electrical  plus
        flow.om(plus,p1)  -1
        effort.om(plus,p1)  0
  ▶ Connections
```

```
import definitions.*;

class SimpleCircuit {flowComponent=bipolarEffortSource, om=2} {
    Ports:
        Electrical{flowPort=source} plus;
        Electrical{flowPort=drain} minus;
        Electrical p1;
        Electrical p2;

    Behavior:
        connect{r=1}(plus,p1);
        connect{r=2}(p1,p2);
        connect{r=3}(p2,minus);


    Transitions:
        when(flow.dir(p1, p2) == 1) {
            //...
        }
        when(flow.om(plus, p1) > 0) {
            //...
        }
        when(effort.om(plus, p1) > effort.om(p1,p2)) {
            //...
        }
}
```

Figure 4.11: Result of the energy flow analysis of a simple circuit shown in the smartIflow Workbench

4.7 Search Space Limitation

This section implements *Contribution C3*.
The size of the state space obtained by the unfolding depends on several factors. At first glance, one might assume that the number of variables is the deciding factor about the size of the state space. In most cases, this assumption is wrong, as the following example demonstrates:

```
class TimerSwitch {
    Variables:
        Enum[Off, On] opm = Off;
        Enum[Failure, Ok] fm = Ok;
    Events:
        Repair;
        ActivateFailure;
        TurnOff;
        TurnOn;
    Transitions:
        when {delay=2} (opm == On) [opm == On && fm == Ok]
            opm = Off;
    EventHandlers:
        when {stable=1} (ActivateFailure) [fm == Ok]
             fm = Failure;
        when {stable=1} (Repair)[fm != Ok]
             fm = Ok;
        when {stable=1} (TurnOff) [opm == On]
            opm = Off;
        when {stable=1} (TurnOn) [opm == Off]
            opm = On;
}
// Main component:
class System {
    // ...
    Components:
        TimerSwitch t1;
        TimerSwitch t2;
        TimerSwitch t3;
    // ...
}
```

`TimerSwitch` is a simple timed switch. If this switch is activated, it switches off automatically after a certain time. However, the switch can also be disabled manually at any time. In addition to the events for switching on and off, there are also events for activating a failure and its repair. Although class `TimerSwitch` consists of only two variables, the transition system obtained by unfolding consists of 587 different states. Reasons for this are the internally and externally triggered transition statements, but also the combinatorics (`System` consists of three timer switches). The unnecessary execution of internally triggered transition statements is already prevented by the rising edge semantics. Internally triggered transition statements are an essential part of the component behavior and therefore often can not be reduced. Of course, appropriate trigger conditions and guard conditions also help to limit the state space to a certain degree (e.g., execution of a transition statement only if a new state results). But what about externally triggered transition statements? External events are usually used to activate failure situations or to change the operational states. In the example, external events are triggered at any possible point in time. Stable conditions are one mechanism for limiting the triggering of external events. They can limit the activation of external events to states that have a certain stability. Despite stable conditions, external events can occur any number of times on a path. In industrial systems, often only double faults or multiple faults up to a certain probability of occurrence are analyzed. This discussion demonstrates that we need a mechanism to control the triggering of external events. A rather naive approach for controlling external events could be an overall upper limit on the number of external events that are allowed to be triggered during unfolding. Despite this restriction, events can still be triggered in any possible combination (e.g., several failure events in sequence). Another approach is to define the relevant event combinations on a path. A typical rule looks as follows: “On each path, a maximum of two failure events and three events that change the operational mode of a component are allowed”. To be able to distinguish the different types of events, features are assigned to them. For example, the feature `type=failure` can be used to describe failure events and feature `type=operation` can be used to define events that change the operational mode of a component. We call these rules **Event Trigger Specifications** (ETS). Event Trigger Specifications need to be specified in a formal and computer-understandable language. In fact, the Stream API introduced in Java 8 comes very close to what is needed. A Java stream is a sequence of data that can be modified by various operations in a pipeline. These so-called intermediate operations can be chained together and applied to a stream. `filter()` is an intermediate operation that passes the elements of a stream that satisfy a certain condition. The result of each intermediate operation is again a stream. In the end, the result is collected. In the following example, the source of the stream is the list `items` which contains a set of strings.

```java
List<String> items = Arrays.asList("Failure", "Event1", "Event2", "Failure", "Fail");
long count = items.stream().filter(s -> s.contains("Fail")).filter(s -> s.length() > 4).
    count(); // count = 2
```

Two chained filters are applied to the stream. The first filter will pass only elements that contain the text “Fail”. The second filter only allows strings that have a length of at least five. In the last step, the elements remaining in the stream after filtering are counted using the function `count()`. Event Trigger Specifications are logical expressions that define the relevant event combinations on a path. An event may only be executed if the ETS is still fulfilled after triggering. The rules are constructed in the following way: The events triggered on a path can be considered as a stream. The source elements are called `path` and `hist`. `path` delivers a stream that contains all events that have been triggered on a path, while `hist` contains all events on a path, except the event to the last state. Filter conditions can be added to the stream to focus on relevant event combinations. This is quite similar to the Java Stream API. The remaining events can be counted (using function `count`) and compared with symbolic values. These expressions represent the atomic statements. Atomic expressions can be combined with logical operators. To make it easier to distinguish between multiple ETS, each specification is given a unique identifier[3]. Consider the following example:

```
max2Failures3Operations := path.filter(type=failure).count() <= 2
                        && path.filter(type=operation).count() <= 3;
```

The ETS above allows on a path at maximum two events of type failure and three events of type operation. The integration into the unfolding function (described in Algorithm 5) is quite easy. For each event that is allowed to be triggered according to its guard and stability condition, it must be checked whether the ETS is still fulfilled after execution. Only if the ETS is fulfilled after execution of an event, the event may be executed. To check an ETS at a state s, all paths back from s to the root must be analyzed.

Definition 4.7.1 Satisfaction of Event Trigger Specifications
Let $s \in S$ and ε an Event Trigger Specifications. ε is fulfilled in state s iff ε is valid on some path from initial state to s ■

Often there are several paths from a state s back to the root. It is important to use the path with the minimal number of events to check an ETS since otherwise paths can occur that contain fewer events as defined in the ETS. As a result, safety-critical situations may not be detected, as certain external events have not been triggered. Checking all paths back to the root from a state s can lead to high computational effort. In the worst case, all paths of the transition system must be traversed. The efficiency of the verification of an ETS can be significantly enhanced in the following way: For each state $s \in S$ the minimum number of relevant events triggered on the way from the root to s is captured (=event counters). Relevant events are the events that are part of the filters of an ETS. For example, for the following ETS, only the events containing features `type=failure` and `group=a` need to be considered:

```
max2FailuresOfGroupA := path.filter(type=failure).filter(group=a).count() <= 2;
```

These event counters can easily be calculated during unfolding. In the initial state, all event counters are assigned 0. If at a state s an event is triggered whose features are part of a filter condition of the ETS, the corresponding event counters are incremented in the subsequent state of s (s'). Once an already explored state is referenced, the event counters at the referenced state must possibly be updated as the following example demonstrates:

Example 4.7.1 Updating event counters
Consider the following model:

```
class Test {
    Variables:
        Enum[Off, Mode1, Mode2] state = Off;
        Enum[Start, A, B, C] pos = Start;

    Events:
        Change{type=operation};
        TurnOn{type=operation};
```

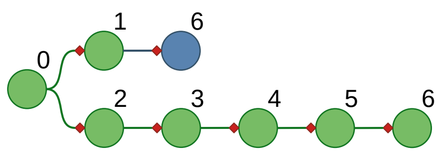
[3]The entire grammar of Event Trigger Specifications in Extended Backus Naur Form (EBNF) in Appendix A.3

```

    EventHandlers:
        when (TurnOn) [state == Off]
            state = Mode2 or Mode1;
        when(Change) [state != Off && pos == Start]
            pos = A;
        when(Change) [state != Off && pos == A]
            pos = B;
        when(Change) [state != Off && pos == B]
            pos = C;
        when(Change) [state != Off && pos == C] {
            pos = A;
            state = Mode2;
        }
}
```

The above smartIflow system model will be unfolded using a stack as a data structure for unprocessed states and with the following ETS:

```
maxFiveOperations := path.filter(type=operation).count() <= 5;
```

Figure 4.12 shows the transition system created by the unfolding and the event counter in the respective state. The execution of event `TurnOn` in the root (state 0) results in states 1 and 2. Accordingly, the event counters for both states have the value 1. Both states are added to the stack of unprocessed states (first state 1, then state 2). The unfolding continues with state 2 (the state is on the top of the stack) and a path of new states results. In state 6 the event `Change` is allowed to be triggered to its guard/stable condition. However, the event must not be triggered, otherwise the ETS would be violated. On the way to state 6, five events have already been triggered that contain the feature `type=operation`. However, the ETS allows a maximum of five events of this type on one path. Therefore, the event `Change` must not be triggered in state 6 and the unfolding procedure stops at this state. Next, the last unprocessed state (state 1) is removed from the stack. The event `Change` is allowed to be triggered according to its guard/stable condition but also based on the ETS. The execution of event `Change` results in a state that already exists (state 6). Therefore, a reference to this state is created.

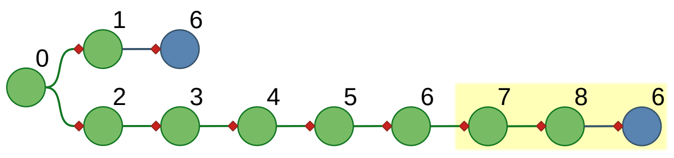


State	#type=operation
0	0
1	1
2	1
3	2
4	3
5	4
6	5

Figure 4.12: Transition system without event counters update

By this reference, the counters for this new path are no longer correct. On the path $0 \rightarrow 1 \rightarrow 6$ only two operation events were triggered. Therefore, the event counter for state 6 must be updated to value 2. Note that the ETS allows on each path a maximum of five events that contain the feature `type=operation`. By the change of the event counter of state 6, now possibly events may be executed, which were not permitted during the unfolding due to the ETS. Event `Change` has not yet been triggered in state 6 and is admissible to be triggered due to its guard/stability condition. The execution of the event results in a new state (state 7) which is then added to the stack of unprocessed states. In state 7, the event `Change` is executed once again resulting in state 8. The reference from state 8 back to state 6 caused by the event `Change` terminates the unfolding (no more new states are discovered). Figure 4.13 shows the final transition system resulting from the execution of additional events after event counters update.

As the event counters are assigned to states, the counters do not count the events actually needed to reach the state but only the minimum possible ones. In other words, the event counters represent the minimum number of events by which a state can be reached. Consequently, the graph will most likely also contain paths that do not satisfy the ETS.



State	`#type=operation`
0	0
1	1
2	1
3	2
4	3
5	4
6	2
7	3
8	4
9	5

Figure 4.13: Transition system after execution of additional events resulting from the event counters update

■

The example shows that each time an existing state is referenced, it must be checked whether the event counters of the referenced state must be updated. Updating an event counter affects not only the referenced state but also possibly the successor states. Algorithm 11 shows a possible implementation for updating event counters and execution of events retrospectively. First, the new event counters for the referenced state arising from the new path are determined (see lines 8-11). The event counters of the referenced state need to be updated if the new values differ from the current values (line 13). If the referenced state has already been processed (state is not part of the stack/queue of unprocessed state), also events may have to be executed subsequently. Since updating an event counter may also affect the event counters of the successor states, the method is called recursively for all transitions to the successor states (lines 16-18). The recursive exploration stops at successor state s' if

- s' is a terminal state or
- there are no changes at the event counters of s' or
- s' is an unprocessed state (state is part of the stack/queue of unprocessed states)

Function getMissingEvents determines the set of events that have not been triggered in a state and that are allowed according to the ETS and stable/guard conditions. The events are executed and the obtained states are added to the set of unprocessed states respectively references are created if the state already exists.

Algorithm 11 Update event counters

```
1: function updateEventCounters(transition, transitionSystem, ets)
2:     $\text{state}_{\text{src}}$ = transition.getStateSrc();
3:     $\text{state}_{\text{dest}}$ = transition.getStateDest();
4:     $\text{evtCtrs}_{\text{src}}$ = transitionSystem.getEventCounter($\text{state}_{\text{src}}$);
5:     $\text{evtCtrs}_{\text{dest}}$ = transitionSystem.getEventCounter($\text{state}_{\text{dest}}$);
6:     $\text{evtCtrs}_{\text{dest}_{\text{actual}}}$ = $\text{evtCtrs}_{\text{src}}$.add(transition.getEvent());
7:     $i = 0$;
8:     while $i <$ $\text{evtCtrs}_{\text{dest}}$.size() do
9:         $\text{evtCtrs}_{\text{dest}_{\text{actual}}}$.set(i, min($\text{evtCtrs}_{\text{dest}_{\text{actual}}}$.get(i), $\text{evtCtrs}_{\text{dest}}$.get(i)));
10:        $i = i + 1$;
11:    end while
12:    if $\text{evtCtrs}_{\text{dest}} \neq \text{evtCtrs}_{\text{dest}_{\text{actual}}}$ then // Updated required if counters differ
13:        transitionSystem.setEventCounter($\text{state}_{\text{dest}}$, $\text{evtCtrs}_{\text{dest}_{\text{actual}}}$);
14:        if $\text{state}_{\text{dest}} \notin$ transitionSystem.getUnprocessedStates() then
15:            // Update event counters of subsequent states
16:            for each $\text{trans}_{\text{next}} \in \text{state}_{\text{dest}}$.getTransitions() do
17:                updateEventCounters($\text{trans}_{\text{next}}$, transitionSystem, ets);
18:            end for
19:            events = getMissingEvents($\text{state}_{\text{dest}}$, ets, transitionSystem);
20:            execute events and insert new states into transitionSystem as unprocessed;
21:        end if
22:    end if
23: end function
```

Event Trigger Specifications help to focus on relevant situations and to keep the search space manageable. In addition, Event Trigger Specifications are also useful during model construction. When modeling, it is quite unlikely that a component model will match the behavior as described in design documents on the first attempt. Rather, modeling is an iterative task in which models emerge incrementally from alternate building and testing. Here, the Event Trigger Specifications help to focus on the nominal system behavior, for example, by disabling all failure events:

```
nominalBehavior := path.filter(type=failure).count() == 0;
```

The specification `nominalBehavior` does not allow triggering of external events that contain the feature type=failure. External events can be assigned multiple features. This also allows the events to be divided into different categories. In the following example, failure events are separated into three categories:

```
...
Events:
   ActivateFailure1{type=failure, category=a};
   ActivateFailure2{type=failure, category=b};
   ActivateFailure3{type=failure, category=c};
...
```

The following Event Trigger Specification allows at maximum two failure events of category a on a path and rejects failure events of category b and c:

```
failureCatA := path.filter(type=failure).filter(category=a).count() <= 2 &&
                path.filter(type=failure).filter(category=b).count() == 0 &&
                path.filter(type=failure).filter(category=c).count() == 0;
```

4.8 Efficient Modeling

This section implements *Contribution C6*.

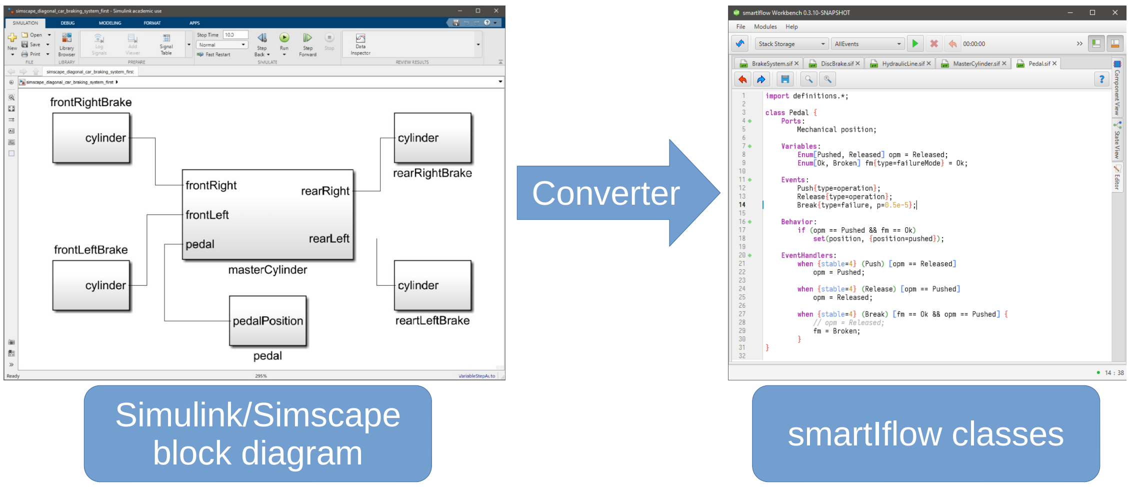
4.8.1 Challenges in Modeling

The smartIflow language is strongly inspired by Java and therefore has many common characteristics. For this reason, most engineers with programming experience will easily become familiar with the smartIflow language. However, many safety engineers have no or very limited experience with programming languages, making text-based modeling to a frustrating, time-consuming, and error-prone task. Modern IDEs (integrated development environments) offer tools such as syntax highlighting, code search, refactoring, or debugging that support the developer in writing code. However, these features are only beneficial when modeling individual components. It will still be challenging to get an overview of the model structure. Especially with larger hierarchical models, consisting of several subsystems at different levels, even people with programming experience can easily lose the overview. Graphical modeling is the way to get rid of the described problem. In principle, there exist various possibilities to provide graphical modeling for smartIflow.

One approach could be to create a graphical modeling environment from scratch or to use a framework like Eclipse Graphical Modeling Framework (GMF)[4]. The result could be a standalone tool combining a graphical editor, a simulation environment, and several analysis tools. If we neglect the high effort for the implementation of a graphical modeling environment, there is still the issue that the safety engineers have to use another tool besides their standard simulation tool (e.g., MathWorks Simulink or Simscape). Existing models cannot be reused and therefore must be created from scratch, or a translation tool is used. Every time the design documents change, both models need to be updated. Keeping the separated models consistent is a time-consuming and error-prone task. Nevertheless, a standalone tool can be perfectly designed for the language. For example, the graphical modeling can be integrated into the simulation environment. This allows the states of the components (e.g., variable assignments or properties on ports) to be displayed in real-time in the graphical representation.

Another approach is to utilize simulation tools already used by safety engineers. Safety engineers are very familiar with their simulation tools. Beyond that, most simulation tools already provide graphical modeling. Therefore, it is obvious to use such tools as a platform for graphical modeling technical systems in the smartIflow formalism. As depicted in Figure 4.14, the idea is to use existing simulation models (or parts of them) and automatically translate them into the smartIflow language. New systems can also be created in the familiar simulation environment. Ideally, there will be only a single model. Therefore, this variant seems to be the better option for graphical modeling. Simulink and Simscape developed by *The MathWorks* are de facto standards for the simulation of technical systems and are widely used in industry. Simulink and Simscape differ in several important aspects from smartIflow. In Simulink models are composed graphically using a set of continuous and discrete blocks that are connected through directed connections.

[4]http://www.eclipse.org/modeling/gmp/



```
import definitions.*;

class Pedal {
    Ports:
        Mechanical position;

    Variables:
        Enum[Pushed, Released] opm = Released;
        Enum[Ok, Broken] fm{type=failureMode} = Ok;

    Events:
        Push{type=operation};
        Release{type=operation};
        Break{type=failure, p=0.5e-5};

    Behavior:
        if (opm == Pushed && fm == Ok)
            set(position, {position=pushed});

    EventHandlers:
        when {stable=4} (Push) [opm == Released]
            opm = Pushed;

        when {stable=4} (Release) [opm == Pushed]
            opm = Released;

        when {stable=4} (Break) [fm == Ok && opm == Pushed] {
            // opm = Released;
            fm = Broken;
        }
}
```

Figure 4.14: Proposed process for graphical model composition

Simulink is based on signals. A Simulink block processes the input signals and then the processed signal is available at the output port(s). Signal values are typed and among others the data types *integer*, *floating-point* and *boolean* are supported. Reactive systems are modeled using Stateflow. For simulation Simulink utilizes numerical solver. This is quite different from smartIflow where only qualitative values exist.

Variant	**New modeling environment**	**Simulation tool integration**
Advantages	+ Modeling environment can be perfectly designed for smartIflow + Tight integration with the simulation environment	+ Modeling in familiar environment + Already existing models can be reused (or parts of them) + Tools share a single model
Disadvantages	- High effort to keep models consistent - Multiple tools have to be used	- Modeling formalism are quite different

Table 4.2: Advantages and disadvantages of the approaches to graphical modeling

Table 4.2 summarizes the advantages and disadvantages of the graphical modeling approaches just presented. Engineers who are not very familiar with programming languages will probably find it much easier to get started with smartIflow if they can model in their familiar environment. Therefore, we will follow this approach. In the next section, we will discuss what a formalism for the graphical design of smartIflow models in Simulink/Simscape could look like.

4.8.2 Towards an efficient Modeling Process in Simulink

Basically, there are two challenges in realizing a process for graphical model composition in Simulink/Simscape. On the one hand, it is necessary to find an appropriate modeling formalism to represent smartIflow models in Simulink/Simscape. On the other hand, a converter is required that translates system models into the smartIflow language. Ideally, this converter is part of the simulation environment. There are several possibilities for graphical modeling of smartIflow systems in Simulink/Simscape. In the following, two quite different approaches are presented and evaluated afterward.

Full Modeling in Simulink/Simscape

One approach is to create smartIflow models from scratch in Simulink/Simscape. That means that aspects of a smartIflow model are represented by Simulink/Simscape components. The question is, which aspects of a smartIflow component model can be represented in Simulink, Simscape, and Stateflow. Are the blocks from the standard library sufficient? Let us begin by finding an appropriate block from Simulink/Simscape for the smartIflow components. A subsystem in Simulink is a special block that can contain other blocks including other subsystems. Thus, subsystems enable the building of hierarchical models. smartIflow also allows hierarchical modeling. It seems to be a good idea to represent each smartIflow component by a separate subsystem. The components can be placed in a custom library in Simulink. The advantage of this is, that a component only needs to be modeled once and multiple instances can be created by drag and drop (see Figure 4.17). The name of the subsystem in the library corresponds to the component type. The name of a library block instance (created by drag and drop) in a block diagram corresponds to the instance name of the component. In case a certain type is not yet available in the custom smartIflow library in Simulink, the type can be represented by a new subsystem, created directly in the Simulink block diagram where the system is modeled. The name of such a subsystem corresponds to the instance name of the component. The component type is undefined because Simulink subsystems are untyped.

Ports are represented by Simscape physical ports (`PMIOPort`). In smartIflow, ports are typed explicitly, while in Simscape the type of a port is defined implicitly by the connection or signal it transfers. Therefore, a mechanism is needed to assign a type to `PMIOPort`s. A possible solution could be a special block that is connected to such a port. This block (called `sifPortTyp`) is a simple dummy component that provides a mask where the port type can be defined. `sifPortTyp`s don't affect the behavior in Simscape. Note that each port should only be assigned one `sifPortTyp` block. Otherwise, it can lead to ambiguities, as the example in Figure 4.15 demonstrates. In this scenario, it is not clear which port is assigned to which type.

In theory, directed Simulink `Inport` and `Output` ports can also be used but with certain limitations. Basically, connections in smartIflow are solely bidirectional, i.e., information can flow in both directions. Optionally, ports can be restricted to let information flow in a certain direction. In Simscape, connection ports, and connections are nondirectional (quite similar to smartIflow). In Simulink, data solely flows over directional connections (in fact the connection structure is a directed graph). This kind of connection modeling is quite

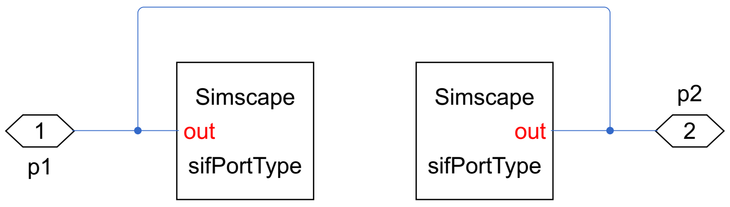


Figure 4.15: Critical situation with ambiguous port types in graphical modeling in Simulink

different from smartIflow with its bidirectional connections. Therefore, only Simscape connections and ports can be used in graphical modeling.

Default connections (i.e., connections that are active in any state) that do not publish any properties can be created as usual by interconnecting the component ports (see Figure 4.17) in Simulink. But what about the state-dependent behavior, transitions, and external events?

Sections `Transitions` and `Behavior` are without a doubt the most complex parts of a smartIflow component. Stateflow is a toolbox in Matlab for modeling state diagrams and seems to be the solution for representing the transitions and the behavior of a component. A possible approach could be exactly one Stateflow chart for each smartIflow component. In this chart both, the transitions and the behavior are defined [HLH15].

As the example in Figure 4.16 shows, the Stateflow chart consists of two superstates, namely `Transitions` and `Behavior`. The superstate `Transitions` is used to describe the variables, their potential values, and as the name implies, the transitions between values. In the Stateflow chart notation, substates with dashed borders indicate a parallel decomposition, while solid borders indicate exclusive state decomposition. The substates of superstate `Transitions` describe the variables of a component and their substates represent the potential values. The chart from the example defines the following variables (initial values are highlighted in bold):

- opm: [**Idle**, Operated]
- circuit1ReservoirLevel: [**Full**, Empty]
- circuit2ReservoirLevel: [**Full**, Empty]

Initial variable values are specified using default transitions, i.e., transitions with no source object. State transitions are labeled with the same logical expressions and features as in the smartIflow language (see Section 4.3.4). Nondeterministic transitions can be modeled by creating transitions that have the same source state and label and end at different destination states.

The state-dependent behavior of a component is defined in superstate `Behavior`. For each conditional behavior, a separate state is created. This state contains two substates, namely `Off` and `On`. The transition from `Off` to `On` is labeled with a logical expression that describes the condition under which the behavior is executed. The actual behavior

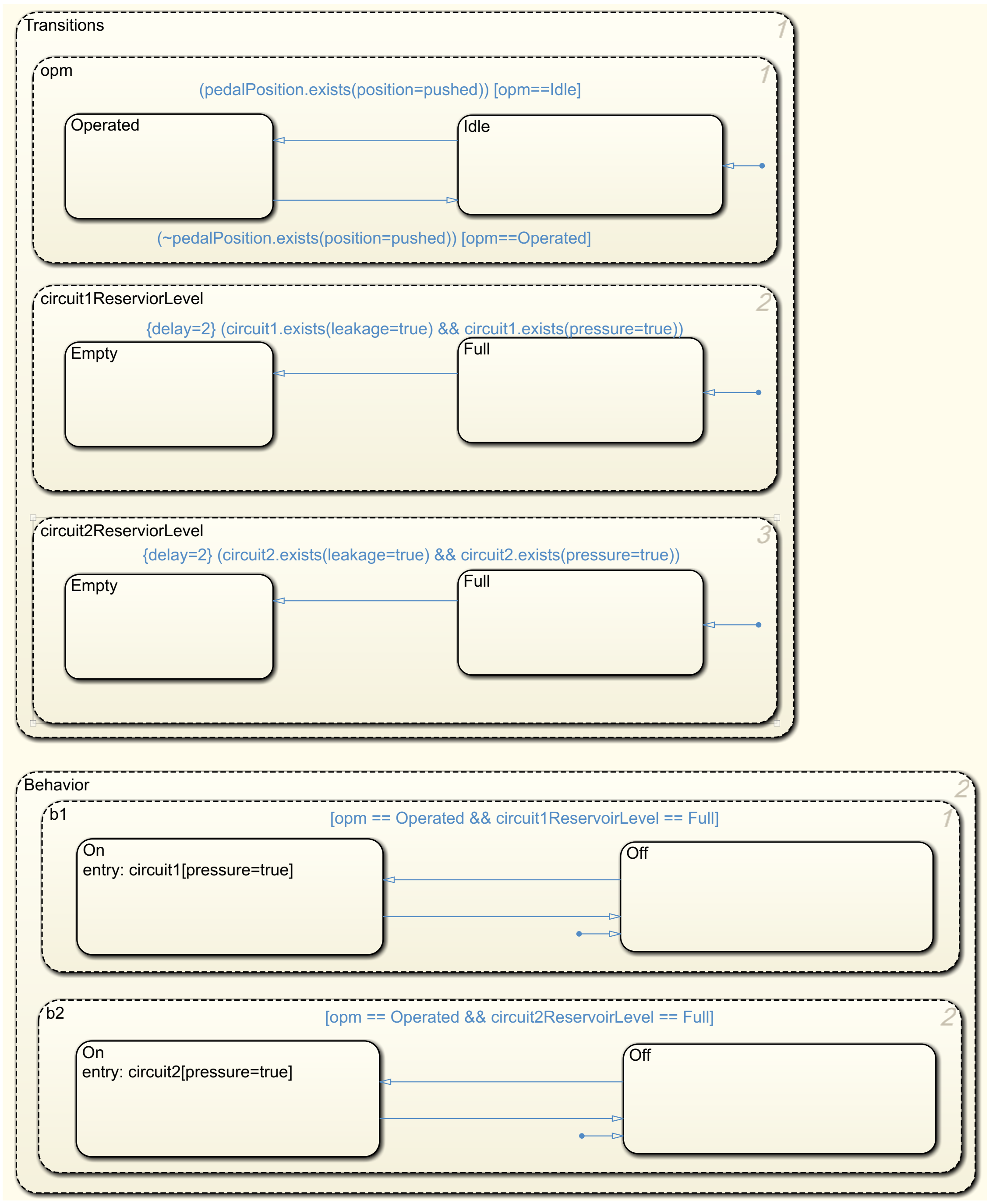

Transitions
1
opm
1
(pedalPosition.exists(position=pushed)) [opm==Idle]
Operated
Idle
(~pedalPosition.exists(position=pushed)) [opm==Operated]
circuit1ReserviorLevel
2
{delay=2} (circuit1.exists(leakage=true) && circuit1.exists(pressure=true))
Empty
Full
circuit2ReserviorLevel
3
{delay=2} (circuit2.exists(leakage=true) && circuit2.exists(pressure=true))
Empty
Full
Behavior
2
b1
1
[opm == Operated && circuit1ReservoirLevel == Full]
On
entry: circuit1[pressure=true]
Off
b2
2
[opm == Operated && circuit2ReservoirLevel == Full]
On
entry: circuit2[pressure=true]
Off

Figure 4.16: A possible graphical representation of transitions and behavior of the master cylinder from the diagonal car braking system (see Section 6.1) in Stateflow

is described in terms of *entry* actions in state `On` using a simplified syntax. For example, the expression `circuit1[pressure=true]` is the shortened form of `set(circuit1, pressure=true);`. The expression `p1=p2[leakage=true]` connects the ports `p1` and `p2` and publishes the property `leakage=true`. For default behavior, i.e., actions that are executed in any state, the condition from `Off` to `On` is omitted.

Of course, these models cannot be simulated in Simscape since the semantics of the models can only be partially interpreted by Simscape. In contrast, the converter understands the models quite well.

Lightweight Approach

Another approach is a half-automated translation process: Instead of modeling all aspects of a component solely in Simulink, use Simulink to model only the component interface (=ports) as well as the default connections. The components are stored in a library as in the first approach, but in addition, each component is assigned the raw smartIflow class. Components from the library can be instantiated as usual by drag and drop (see Figure 4.17). For a novel type that does not exist in the library yet, at least a component class with ports and subcomponents can be created. Consequently, the connection structure of arbitrary Simulink/Simscape models can be completely transferred to smartIflow.

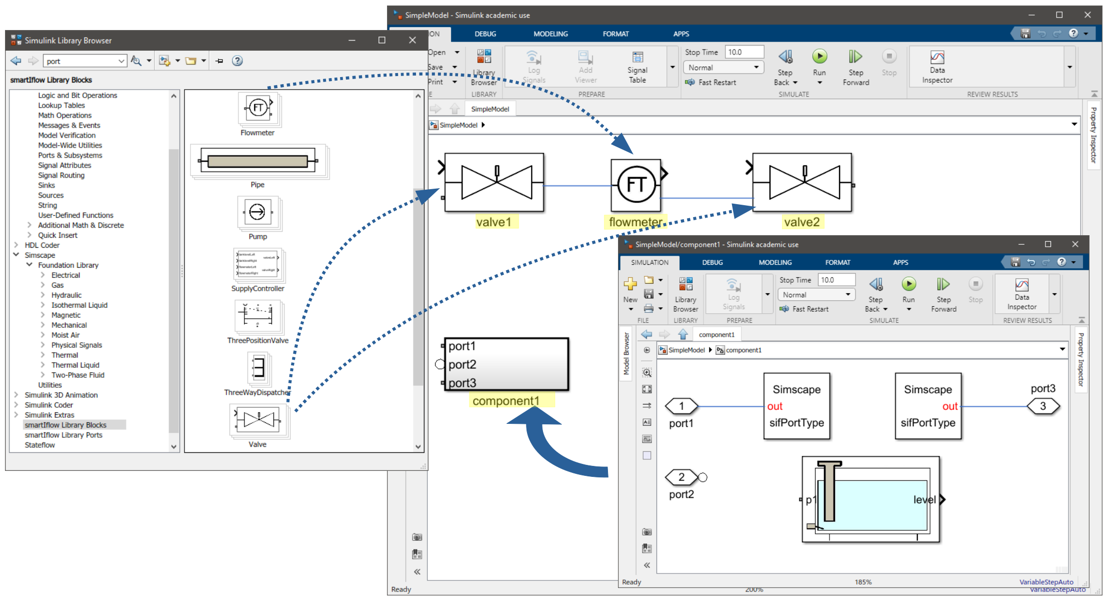


Figure 4.17: Graphical representation of smartIflow components in Simulink

Evaluation

The formalism of the first approach obviously does not follow the modeling guidelines for Stateflow charts. However, the additional syntax is mandatory for publishing properties, specifying delays, etc. In principle, behavior, and transitions modeled in a Stateflow chart have the same expressive power as defining them in the smartIflow language. This is good news because it proves that it is possible to compose models graphically in Simulink/Simscape that are semantically equal to the models in raw smartIflow classes. The drawback is that very special kinds of Simscape models have to be created. The component models need to follow some conventions (e.g., `sifPortTyp`s are used to define a port type). With larger systems, this kind of modeling is not practicable. The reason for this is the inconvenient modeling of behavior and transitions in Stateflow. While the definition of variables is quite intuitive, problems occur with the definition of actions (set/connection) and logical expressions as there is absolutely no support. The safety engineer gets feedback about the syntactic correctness of the expressions only when parsing the converted smartIflow classes. This results in a tedious and time-consuming process that is not applicable for use in industry. In addition to this, the graphical representation is by no means better readable than the textual representation.

With respect to the second approach, there are (at least) three scenarios where the Simulink integration improves modeling efficiency:

(1) In situations when modeling systems solely using blocks from the custom smartIflow library in Simulink/Simscape (and underlying raw smartIflow classes). Ideally, the translated models can be executed without any manual work.

(2) Existing Simulink/Simscape models can easily be converted into smartIflow classes. In this case, at least the connection structure can be transferred. Variables, behavior, transitions, and events must be defined manually.

(3) For modeling completely new systems, Simulink/Simscape can be used to specify the system architecture with very little effort. Figure 4.18 depicts this process. The system architecture can be represented just with subsystems, ports, and connections (1). After translation into smartIflow classes (2), variables, behaviors, transitions, and events can be added to the classes manually (3).

Table 4.3 summarizes the advantages and disadvantages of both approaches. Obviously, the lightweight approach seems to be a much better option. This is also confirmed by the experience of an industrial project in which also a lightweight approach has been developed. In the SysRO Project [Gir+20], technical systems were described with the SysML (focus on structural modeling) language and then analyzed using smartIflow. The next chapter describes, how these special models can be converted from Simulink/Simscape into the smartIflow language.

4.8.3 A first Prototypical Implementation and Experimental Validation

In principle, there are two possibilities for converting system models in Simulink to the smartIflow language: One way is to perform the conversion directly in Matlab. Matlab offers a comprehensive API (Application Programming Interface) that enables access to block parameters and properties (e.g., information about ports, layout, or the block type). The API can be used to traverse a system model, gather all relevant information, and generate smartIflow code out of it. Another way to perform the conversion is to use a separate tool that takes the system model in the Matlab file format as input. Matlab

Variant	**Full Modeling in Simulink/ Simscape**	**Lightweight Approach**
Advantages	+ New models can be created solely in Simulink/ Simscape	+ Simple and intuitive modeling process + Models are easy to read + For existing models, at least the connection structure can be transferred
Disadvantages	- Models cannot be simulated in Simulink/Simscape - Complex modeling formalism - Models are hard to read - No feedback about the correctness of expressions	- Synchronization of models (smartIflow $\leftrightarrow$ Simulink/Simscape) is quite complex

Table 4.3: Comparison of full modeling in Simulink/Simscape and the lightweight approach

stores models in the so-called *SLX* format[5]. Actually, an SLX file is just a compressed file that contains a set of XML files that describe the structure of a block diagram. In principle, it is possible to parse the XML files and generate smartIflow classes out of them. However, the format will likely change between the different Simulink versions due to new features. Therefore, it is not possible to create a reliable conversion tool that accepts every model regardless of its version. For this reason, the conversion should be done in Matlab.

Algorithm 12 shows a possible conversion implementation. The approach is quite simple: The hierarchical model structure is traversed, and at each level, all blocks are analyzed and a new smartIflow class is created. Depending on the existing block types at a level, either ports or subcomponents are added to the corresponding smartIflow class. Simulink/Simscape ports (`PMIOPort`, `Inport` and `Outport`) can be converted straightforward. For component instances from a custom library, the assigned smartIflow class is also copied. Additionally, the connection structure is analyzed and corresponding connections are added to the smartIflow class. To keep the layout of the model, the position information of each block is saved in an XML file.

The prototype is implemented in the Matlab language. The block diagram has to be translated and the output folder where the smartIflow classes are stored can be selected using a simple dialog (see (2) in Figure 4.18). The translation of a system consisting of 90 components takes approximately two seconds. Since translation time grows linear with the number of blocks in a model, computational effort is no limitation factor of this approach.

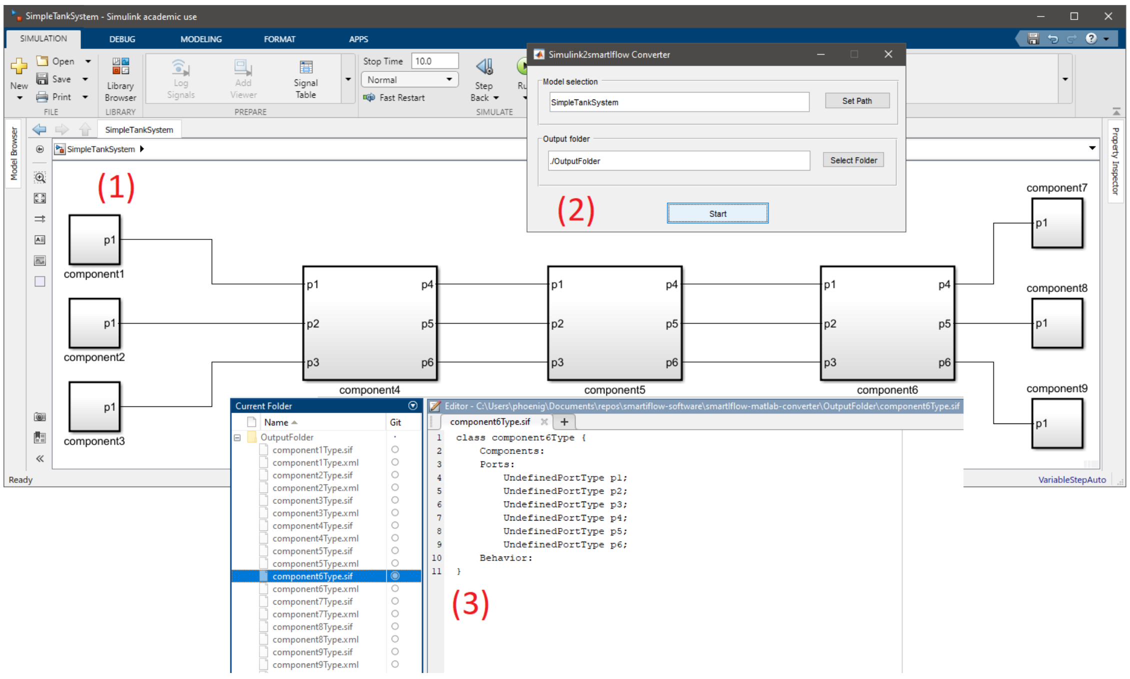
Our experiments with the prototype showed, that the approach has the potential to

[5]The SLX format has been standard since 2012 and has replaced the outdated MDL format.

Algorithm 12 Traversing Simulink models

```
function convert(block, outputFolder)
    sifClass = new smartIflow class instance;
    sifClass.typeName = block.name + 'Type';
    sifClass.instanceName = block.name;
    children = all blocks inside block;
    for each $block_i \in$ children do
        instanceName = get name of $block_i$;
        pos = get position of $block_i$;
        analyzeConnectionStructure($block_i$, sifClass);

        if $block_i$ is a port then
            convertPort($block_i$, sifClass, outputFolder);
        else if $block_i$ is referenced from library then
            comp = create empty component;
            comp.instanceName = instanceName;
            comp.typeName = get name of referenced block;
            sifClass.components = sifClass.components $\cup$ comp;
            comp.position = pos;

            // Copy raw smartIflow class if type is available
            if lookuptable contains comp.typeName then
                copy smartIflow class to outputFolder;
            end if
        else
            comp = create empty component;
            comp.instanceName = instanceName;
            comp.typeName = instanceName + 'Type';
            sifClass.components = sifClass.components $\cup$ comp;
            comp.position = pos;

            // Analyze blocks of the subsystem recursively
            if blockType == 'SubSystem' then
                convert($block_i$, outputFolder);
            end if
        end if
    end for
    writeSmartIflowClass(outputFolder, sifClass);
end function
```

increase the efficiency of modeling. Especially when modeling completely new systems, the approach is quite practicable and reduces effort significantly. Modeling systems solely using blocks from the custom smartIflow library in Simulink/ Simscape is applicable in practice only with certain restrictions. Even with a very comprehensive library of predefined blocks in Simulink there are often special component types (e.g., controller) required that are not available in the library yet. In this case, the safety engineer needs to model the component in the smartIflow language. Another drawback is, that the translation works only in one direction. Models can be translated into the smartIflow language but not the other way around. Therefore, changes in the system architecture in raw smartIflow classes do not affect the graphical representation. This results in inconsistencies that need to be fixed manually. Nevertheless, the approach provides a good starting point for safety engineers who had no experience with smartIflow so far. The approach also is very useful for the analysis of existing systems, since systems can be modeled faster and thus development costs are reduced.



Simulink2smartIflow Converter

Model selection: SimpleTankSystem [Set Path]

Output folder: ./OutputFolder [Select Folder]

[Start]

Editor - C:\Users\phoenig\Documents\repos\smartiflow-software\smartIflow-matlab-converter\OutputFolder\component6Type.sif

```
class component6Type {
    Components:
    Ports:
        UndefinedPortType p1;
        UndefinedPortType p2;
        UndefinedPortType p3;
        UndefinedPortType p4;
        UndefinedPortType p5;
        UndefinedPortType p6;
    Behavior:
}
```

Figure 4.18: Modeling a system architecture in Simulink/Simscape and translation to smartIflow classes

5 Formal Verification of Safety Requirements

Verification of safety requirements is one important task during the development of safety-critical systems. The first step in this analysis task is the specification of safety requirements. These requirements describe how the system should behave in general (e.g., expected behavior after a switch is pressed) or under special circumstances (e.g., behavior after a failure has happened). An example of a safety requirement could be “The cooling system shall keep the pressures of cooling liquid at any time”. After requirements specification, these requirements are verified against the model behavior. A specification is fulfilled by a design model if the system behaves as defined in the specification in every situation.

In principle, this task can be performed manually for smartIflow system model by checking whether all relevant execution paths of the transition system (obtained by unfolding) fulfill the specification. However, this is only feasible for very small systems. Despite mechanisms such as the referencing already explored states or the Event Trigger Specifications, the state space can grow very fast even of smaller models to several thousand states. A manual analysis of such results is extremely time-consuming, tedious, and therefore not practicable. Verification of safety requirements for real systems requires an automated method.

Many approaches to model-based safety analysis (e.g., Güdemann et al. [GO10], AltaRica 3.0 [Bat+13] or Joshi et al. [JWH]) use formal verification methods to automate the verification process. They utilize model checkers as they operate completely automatically, in contrast to deductive verification based on a theorem prover that requires expert knowledge. Model checking is also known as the ‘push-button’ method, meaning that it automatically checks whether a system fulfills a given specification. Model checkers require the system description and the specification in a formal language. As described in Section 2.4, there are two types of model-checking algorithms, namely symbolic model checking and explicit-state model checking. Symbolic model checkers are quite efficient, i.e., they can handle large state spaces. However, symbolic model checking is not compatible with the complicated implicitly defined transitions of smartIflow. Therefore, explicit-state model checking seems to be the solution. Explicit-state model checkers use a system description in a formal language to construct, as the name already implies, the transition system in an explicit representation. There already exist several explicit-state model checking tools (e.g., SPIN[Hol97] or PRISM[KNP11]). One approach to automate the verification of safety requirements of smartIflow system model could be to use an existing model-checking tool. The idea is to translate smartIflow system model into the input language of a model checker, define safety requirements in temporal logic, and let the model checker automatically verify the requirements against the model. Unfortunately, there is no input language that can represent the complex component behavior of smartIflow system model due to the following reasons:

- smartIflow allows to specify delays for internally triggered transition statements for reasoning about orders of magnitude of time. Existing model-checking tools do not support this type of time semantics.
- Existing model checking tools do not support mechanisms for state space limitation such as Event Trigger Specifications.

In addition to the reasons mentioned above, there is another strong argument to develop a new model checker specifically for smartIflow system model: One strength of model checking is the generation of counterexamples, in case the system model does not meet a specification. Of course, one counterexample is sufficient to disprove a specification. However, in safety analysis, the main question is not, whether a system can fail, but whether the probability of a failure is acceptable. Any system will fail if a certain number of failures occur in the system. Consider, for example, an auto emergency braking system available in many modern cars. Such systems utilize various sensors, such as radar sensors, to prevent a collision with an obstacle or to reduce the collision speed. One important safety requirement is “The emergency brake assist must only brake if there is an obstacle in front of the vehicle.”. Obviously, it cannot be guaranteed that this specification is fulfilled in any imaginable situation. The sensors may deliver falsified values or there could be an error in the software that is responsible for detecting obstacles. It cannot be guaranteed that the specification is always met, but the likelihood of it can be optimized. Therefore, it is not sufficient to deliver just one counterexample but rather to calculate all relevant counterexamples that lead to a violation of a specification. There already exist some approaches to tackle this problem (e.g., [Cla+02]), however existing tools still do not support multiple counterexample generation.

For this reason, an explicit-state model checker for smartIflow system model has been developed (the approach was first introduced in [HLH16][HLH17]) which is described in this chapter. To capture the timed behavior based on orders of magnitude of time in safety requirements, operators in CTL are extended as described in the next section. Finally, the calculation of minimal cutsets and generation of Fault Trees is explained.

5.1 Safety Requirements Specification

The safety requirements verified against a system model need to be specified in some formal language. Existing model-checking tools usually accept requirements specified in temporal logic such as CTL or LTL. As already described in Section 2.4.2, CTL does not a have higher expression power than LTL or vice versa, but their underlying concept is different. In other words, they are semantically incomparable. But which logic is more suitable for specifying safety requirements for technical systems? First, consider some typical specifications for real-world scenarios:

1. “There is always a possibility that the light can be turned on.”
2. “It must be possible to switch to manual mode at any time.”

3. “The airbag must be triggered only in case of a crash.”

The first example describes that there is always a possibility that a state can be reached (also known as reset property). This property cannot be expressed with LTL. LTL can only state that a state is always reached (e.g., $\mathsf{F}(Light == On)$). With branching in CTL it is possible to specify expressions that must be valid on at least one possible execution path. For example, the first requirement can be described by the following expression in CTL: $\mathsf{AG}(\mathsf{EF}(Light == On))$. Requirements in the form of the first example are quite important in real-world scenarios. Thus, CTL is used as a formal language to describe safety requirements for smartIflow system models. The remaining requirements can be expressed in both LTL and CTL.

Temporal logic like LTL or CTL abstracts from precise timing. For example, the expression $\mathsf{AF}\,\varphi$ states only that on all paths φ must be reached sometime in the future. “Somewhere in the future” could be a subsequent state or an arbitrary state along the reachable paths. Of course, this is not sufficient for real-world applications. In real-world applications, reactions to events are often expected within a guaranteed time.

Example 5.1.1 Diagonal Car Braking System
Consider the diagonal car braking system (see Section 6.1 for detail description) and the following property:

“Whenever the pedal is pressed, the brake must operate after less than 10 ms.”

Most aspects of the requirement could be expressed by the following CTL formula:

$$\mathsf{AG}(PedalPressed == \text{true} \rightarrow \mathsf{AF}(BrakeOperates == \text{true}))$$

$\mathsf{AF}(BrakeOperates == \text{true})$ means that the brake must operate somewhere in the future. This could be the next state or any future state. This is obviously too weak since a reaction is expected after at least 10 ms.

In this example, there are some interesting aspects that should be considered in CTL expressions:

- What is the maximum delay allowed before the reaction (braking effect) occurs?
- How long does the pedal have to be pressed until a reaction is possible?
- How long does the expected reaction (braking effect) have to last?

■

There already exists variants of CTL such as TCTL or RTCTL that can express timed properties. Properties to timed automata can be expressed with TCTL. TCTL uses clock constraints over the clocks of timed automata for reasoning about timing. TCTL cannot be used to specify requirements for smartIflow system models, because the timing in smartIflow is based on different orders of magnitude of time. RTCTL is also not suitable for specifying temporal requirements for smartIflow system models, as operators in RTCTL are just a helper macro for n next-operators in series. Since no suitable temporal logic for specifying safety properties to smartIflow system models exists, a new variant of CTL is required. CTL must be extended in such a way that safety requirements can also take into account the stability of states and the delays between state transitions.

5.2 Orders of Magnitude timed CTL (OMT-CTL)

This section implements *Contribution C5.*
Orders of Magnitude timed CTL (OMT-CTL, for short) is a variant of CTL aimed to express timed properties for smartIflow system models. To address temporal behavior in safety requirements, CTL is extended by two new concepts: The first concept is an additional predicate that can be attached to state formulae. This predicate is used to specify properties about the stability of states. It is a term of the form

$$\Phi_{\{stable\ \omega\ \mathrm{x}\}}$$

where Φ is a state formula, *stable* is the keyword for the stability value of a state, ω can be any relational operator ($<, <=, >, >=, !=, ==$), and x is a natural number. For example, $\mathsf{AG}(\mathrm{true}\{stable >= 3\})$ asserts that each state in the transition system must be stable with respect to three orders of magnitude above reference.

The second concept introduces a constraint to temporal operators (F, G, and U) that enables limiting the search depth. The so called *delay constraint* is a term of the form

$$\Theta_{\{delay\ \omega\ \mathrm{x}\}}$$

where Θ is a path-specific quantifier (F, G, or U), *delay* is the keyword for the delay of the transitions, ω can be any relational operator ($<, <=, >, >=, !=, ==$), and x is a natural number. For example, $\Phi\,\mathsf{U}\{delay < 3\}\Psi$ asserts that Ψ has to hold within a delay from at maximum two orders of magnitude above reference while only visiting Φ-states before reaching Ψ (all transitions on the path to Ψ must have a delay of order of magnitude value of two or less). Note that in OMT-CTL, as in TCTL, there is no timed variant of the next-step operator (X).

Definition 5.2.1 Syntax of OMT-CTL
Basically, the formulae in OMT-CTL have the same structure as CTL, RTCTL or TCTL expressions and are either state or path formulae. OMT-CTL state formulae are built upon a set AP of atomic propositions according to the following grammar:

$$\Phi ::= true \mid a \mid \Phi_{\{stable\ \omega\ \mathrm{x}\}} \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{E}\varphi \mid \mathsf{A}\varphi$$

where $a \in AP$ and φ is a path formula defined by $\varphi := \Phi\,\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi$. ■

Other propositional logic operators ($\vee, \rightarrow, \ldots$) can be obtained straightforward:

$$\begin{aligned}
\mathrm{true} &= \neg\mathrm{false} \\
\Phi \rightarrow \Psi &= \neg(\Phi \wedge \neg\Psi) \\
\Phi \leftrightarrow \Psi &= (\Phi \rightarrow \Psi) \wedge (\Psi \rightarrow \Phi) \\
\Phi \vee \Psi &= \neg(\neg\Phi \wedge \neg\Psi)
\end{aligned}$$

Order of magnitude timed variants of the Operators F and G can be obtained in the same way as in TCTL:

$$\begin{aligned}
\mathsf{EF}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi &= \mathsf{E}\,\mathrm{true}\,\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi \\
\mathsf{AF}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi &= \mathsf{A}\,\mathrm{true}\,\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi \\
\mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi &= \neg\,\mathsf{AF}_{\{delay\ \omega\ \mathrm{x}\}}\,\neg\Phi \\
\mathsf{AG}_{\{delay\ \omega\ \mathrm{x}\}}\,\Phi &= \neg\,\mathsf{EF}_{\{delay\ \omega\ \mathrm{x}\}}\,\neg\Phi
\end{aligned}$$

$\{stable\ \omega\ \mathrm{x}\}$ and $\{delay\ \omega\ \mathrm{x}\}$ are optional constraints that can be applied to state formulae respectively path-specific quantifiers. An OMT-CTL expression without any stable or delay constraint complies with regular CTL syntax and semantics.

Definition 5.2.2 Atomic Propositions in OMT-CTL
In OMT-CTL, atomic propositions are comparisons between variables and their values using the relational operators `==` and `!=`. Variables of components are accessed using the full qualified name. ■

Definition 5.2.3 Full Qualified Variable Name
A fully qualified variable name includes all component names separated by "." in the hierarchical sequence starting from the root component to the component that contains the desired variable and the name of the variable itself. ■

Example 5.2.1 OMT-CTL examples
Consider the following source code, which contains several OMT-CTL specifications:

```
stablityProperty1 := AG(true{stable >= 3});
stablityProperty2 := AG(true{stable >= 3} -> opm = Idle);
restartProperty := AG(EF{delay <= 2}(system.om == Restart));
correctBehavior :=
        AG((system1.om == Request){stable >= 2}
              -> AF{delay <= 3}((system2.om == Acknowledge){stable >= 2}));
failureDectection := AG(component1.fm == Failure -> AF(system.om == Repair));
systemOk := correctBehavior && restartProperty && failureDectection;
```

Each OMT-CTL specification is identified by a unique identifier. Therefore, specifications are uniquely referenceable, and in addition, sub-formulas can be reused (the rule 'Declare before use' holds). Formulae can be reused by referencing them using their identifier. Specification `stablityProperty1` defines that all states in a system are stable with respect to at least 3 orders of magnitude above reference. Specification `stablityProperty2` reuses `stablityProperty1` and states:

> "In all states on all paths that are stable with respect to 3 orders of magnitude above the reference, the system is in state idle."

The restart property defined in specification `restartProperty` expresses that from any state it is possible to perform a restart. The specification is not satisfied if, before the restart is performed, there is a state that is stable with respect to the order of magnitude 3 (or higher), even if all subsequent states have stable values that are smaller. The Property

> "Whenever a request holds for a duration of at least two orders of magnitude, an acknowledge is expected after a delay of 3 orders of magnitude at the latest, which holds for a duration of at least 2 orders of magnitude."

is expressed by the OMT-CTL formula `correctBehavior`. Specification `systemOk` is an example where existing requirements are reused. ■

Definition 5.2.4 Satisfaction Relation for OMT-CTL
Like regular CTL formulae OMT-CTL expressions are evaluated over the states and paths of transition systems $TS(\mathit{fssm}) = (S, I, AP, L, \longrightarrow)$. Let Paths(s) be the set of maximal path fragments π where $\pi[i]$ denotes the $(i+1)$th state of path π, and $\pi[0] = s$. For state $s = (\eta_v, \mathrm{state}_{ts_{\mathrm{prev}}}, \mathrm{state}_{ts_{\mathrm{del}}})$ in $TS(\mathit{fssm})$, $a \in AP$, OMT-CTL state formulae Φ and Ψ, and OMT-CTL path formula φ, the satisfaction relation $\models$ for state formulae is defined by

$$\begin{array}{lll}
s \models \mathrm{true} & & \\
s \models a & \text{iff} & a \in L(s) \\
s \models \Phi_{\{\mathit{stable}\ \omega\ \mathrm{x}\}} & \text{iff} & s \models \Phi \text{ and } \mathit{Stable}(s)\ \omega\ x \\
s \models \neg\Phi & \text{iff} & \text{not}\, s \models \Phi \\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \,\text{and}\, (s \models \Psi) \\
s \models \mathsf{E}\,\varphi & \text{iff} & \pi \models \varphi \,\text{for some}\, \pi \in \mathit{Paths}(s) \\
s \models \mathsf{A}\,\varphi & \text{iff} & \pi \models \varphi \,\text{for all}\, \pi \in \mathit{Paths}(s)
\end{array}$$

Remember that ω can be any relational operator, x is a natural number, and $\mathit{Stable}(s) = x$ iff s is marked stable with respect to OM value x.

Let $\pi = s_0 \xrightarrow{d_0}_{e_0} s_1 \xrightarrow{d_1}_{e_1} s_2 \xrightarrow{d_2}_{e_2} \ldots$ and $\mathit{delay}(s_i, s_j) = \mathit{max}(\{d_k \mid i \leq k < j\})$. The satisfaction relation $\models$ for OMT-CTL path formulae is defined by:

$$\begin{array}{lll}
\pi \models \mathsf{G}_{\{\mathit{delay}\ \omega\ \mathrm{x}\}}\,\Phi & \text{iff} & \forall j : (\mathit{delay}(\pi[0], \pi[j])\,\omega\,x) \;\rightarrow\; (\pi[j] \models \Phi) \\
\pi \models \Phi\,\mathsf{U}_{\{\mathit{delay}\ \omega\ \mathrm{x}\}}\,\Psi & \text{iff} & \exists j \geq 0 : (\mathit{delay}(\pi[0], \pi[j])\,\omega\,x) \quad \wedge \\
& & (\Phi[j] \models \Psi) \wedge (\forall 0 \leq k < j : \pi[k] \models \Phi)
\end{array}$$

■

Example 5.2.2 Examples of OMT-CTL formulae
Figure 5.1 visualizes three OMT-CTL expressions. Example 1 states that proposition p is always true on all paths within the specified time period. The time period is defined by paths whose delay values are less than the order of magnitude value 2. The second example defines that a state where proposition p is true is reached on all paths after a maximum delay of three orders of magnitude. Example 3 states that there is at least one path on which a q-state is reached after a delay of at maximum two orders of magnitude above reference, whereby only p-states are visited before q is reached. Although each path fulfills the property "p-states are visited before q is reached", the transition system does not fulfill the specification because the q-state is not reached within the specified time interval.

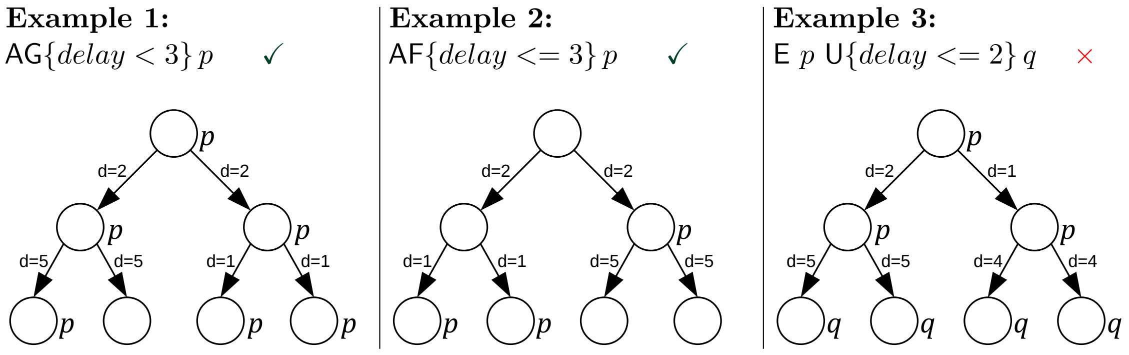

Example 1:
$\mathsf{AG}\{delay < 3\}\, p$ ✓
d=2
d=2
d=5
d=5
d=1
d=1
p
Example 2:
$\mathsf{AF}\{delay <= 3\}\, p$ ✓
d=2
d=2
d=1
d=1
d=5
d=5
p
Example 3:
$\mathsf{E}\ p\ \mathsf{U}\{delay <= 2\}\, q$ ×
d=2
d=1
d=5
d=5
d=4
d=4
p
q

Figure 5.1: Examples of OMT-CTL formulae

■

In CTL, all temporal modalities can be expressed using the basic operators EX, EF, and EG (=Existential Normal Form). For each OMT-CTL formula also exists an equivalent formula in ENF. Universal path quantifiers can be eliminated using the following duality laws:

Definition 5.2.5 Duality Laws for OMT-CTL

$$\mathsf{AF}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi) \equiv \neg\, \mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\neg\Phi)$$

$$\mathsf{EF}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi) \equiv \mathsf{E}((\mathrm{true})\ \mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\ \Phi)$$

$$\mathsf{AG}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi) \equiv \neg\, \mathsf{EF}_{\{delay\ \omega\ \mathrm{x}\}}(\neg\Phi)$$

$$\mathsf{A}(\Phi\ \mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\ \Psi) \equiv \neg\, \mathsf{E}(\neg\Psi\ \mathsf{U}(\neg\Phi \wedge \neg\Psi)) \wedge \neg\, \mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\neg\Psi)$$

$$\mathsf{AX}(\Phi) \equiv \neg\, \mathsf{EX}(\neg\Phi)$$

■

5.3 Explicit-State OMT-CTL Model Checking

This section implements *Contribution C4*.
The model checking problem for OMT-CTL is the same as the model checking problem for CTL: It is to verify for a OMT-CTL formula Φ and transition system TS whether $TS \models \Phi$. Therefore, it is obvious to use an existing model-checking algorithm. The explicit-state CTL model checker presented in [BK08] is quite efficient. The question whether $TS \models \Phi$ can be answered in linear time ($O((N + K) * |\Phi|)$, where $|\Phi|$ is number of subformulae of CTL formula Φ, N is the number of nodes and K is the number of transitions in TS). The algorithm is based on the calculation of satisfaction sets for each subformula Ψ of a CTL formula Φ. Remember, a satisfaction set $Sat(\phi)$ contains those states that satisfy ϕ. However, the use of this algorithm for the verification of smartIflow system models is associated with some considerable difficulties:

- During the calculation of the satisfaction sets, timing constraints and ETS must be considered. The calculation of $Sat(\mathsf{E}\,\phi\,\mathsf{U}\,\psi)$, $Sat(\mathsf{EG}\,\phi)$, $Sat(\mathsf{EX}\,\phi)$ is based on backward search. If the backward search discovers a path where the ETS or a timing constraint is violated, then the path must not be explored further.
- For the calculation of a satisfaction set of a subformula, the results of the computations of its children are used. For testing an ETS during backward search of an intermediate formula in addition to the events on the current path, the events that were triggered in the satisfaction sets of its children must also be considered. A possible solution could be as follows: For each element, s_i in satisfaction set $Sat(\Psi)$ for subformula Ψ, the relevant events that were triggered to satisfy subformula Ψ must be captured. Relevant events are those that occur in the ETS.
- During backward search, references and, depending on the ETS, cycles must be followed.

The probably greatest advantage of the model checking approach based on the calculation of satisfaction sets is its efficiency. However, it is not clear whether this algorithm can even be implemented for smartIflow system model. In addition, the approach does not support on-the-fly model checking, where only the paths relevant to model checking (e.g., to determine cut sets) are unfolded. For these reasons, a very simple but not particularly efficient model checker has been developed. This algorithm should satisfy the following properties:

- In case of specification violation, all relevant counterexamples should be generated.
- The model checker may only check the paths in the transition system that are allowed with respect to the ETS. Thus, search space can be limited to relevant paths (e.g., at a maximum of two failure events).
- While exploring new paths, the timing requirements of the OMT-CTL formulae must be considered.

5.3.1 Basic Algorithm

The proposed model checking algorithm accepts OMT-CTL formulae in ENF, i.e., OMT-CTL formulae consisting only of the basic operators EG, EU, and EX. Each OMT-CTL formula can be transformed automatically into an equivalent ENF formula using the duality laws from Definition 5.2.5. For example, the OMT-CTL formula

$$\mathsf{AG}(\mathsf{AF}\{delay < 3\}(om == Ok))$$

is automatically transformed to

$$!(\mathsf{E}(\text{true}\,\mathsf{U}!(!(\mathsf{EG}\{delay < 3\}!(om == Ok)))))$$

As OMT-CTL formulae are expected in ENF, only algorithms for $\mathsf{E}\,\Phi_1\,\mathsf{U}\,\Phi_2$, $\mathsf{EG}\,\Psi$, $\mathsf{EX}\,\Psi$, and Φ are required.

After the transformation of OMT-CTL formula Φ to ENF, the question of whether $TS \models \Phi$ can be solved. Algorithm 13 sketches the basic idea of the model-checking procedure. The recursive algorithm is based on an incremental top-down traversal of the parse tree of the OMT-CTL formula Φ. As OMT-CTL formulae are expected in ENF, labels of inner nodes of the parse tree are $\neg, \wedge, \mathsf{EG}, \mathsf{EU}$, or EX. Leaves can be atomic propositions $a \in AP$ or the constant true. For complying ETS and cycle detection (cycles will be discussed later) while traversing the transition system, the state transitions visited must be remembered. The list history (the order is important) consists of a set of state transitions that describe the path towards the state where Φ is verified. The history must necessarily be composed of state transitions since each state transition is assigned information about the event (e.g., type of the externally triggered transition statement) that is essential for testing the ETS. Initially, history contains an artificial state transition leading to the initial state of the transition system (we expect that there is only one initial state). Function verify analyzes the formula Φ and evaluates subformulae by recursively calling function verify. If Φ is of the form $\mathsf{E}\,\Phi_1\,\mathsf{U}\,\Phi_2$, $\mathsf{EG}\,\Psi$, or $\mathsf{EX}\,\Psi$, the expression is evaluated by the corresponding functions verifyEU, verifyEG and verifyEX. The result of each function (verify, verifyEU, verifyEG, and verifyEX) is a tuple of the form (result, proofs). The result is true iff $TS \models \Phi$. The set proofs consists of paths that prove the result. If result is true, proofs are witnesses and if result is false, proofs are counterexamples. Note that a negation will invert the result, i.e., witnesses turn to counterexamples and counterexamples turn to witnesses.

Algorithm 13 Function for computation of verification results

Input: Finite transition system TS with state set S, OMT-CTL formula Φ in ENF, an ETS, and history describing the trace to the state where Φ is verified.

Output: The tuple (result, proofs) where result = true iff $TS \models \Phi$ in state lastState(history) and proofs is a set of counterexamples/witnesses (depending on result).

```
1:  function verify($\Phi$, history, $TS$, ets)
2:      switch $\Phi$ do
3:          case true :
4:                  return (true, $\emptyset$);
5:          case a :
6:                  if $a \in L$(lastState(history)) then
7:                      return (true, $\emptyset$);
8:                  else
9:                      return (false, $\emptyset$);
10:                 end if
11:         case $\Phi_1 \wedge \Phi_2$ :
12:                 (result$_{\Phi_1}$, proofs$_{\Phi_1}$) = verify($\Phi_1$, history, $TS$, ets);
13:                 (result$_{\Phi_2}$, proofs$_{\Phi_2}$) = verify($\Phi_2$, history, $TS$, ets);
14:                 return (result$_{\Phi_1}$ $\wedge$ result$_{\Phi_2}$, proofs$_{\Phi_1}$ $\cup$ proofs$_{\Phi_2}$);
15:         case $\neg\Psi$ :
16:                 (result, proofs) = verify($\Psi$, history, $TS$, ets);
17:                 return ($\neg$result, proofs);
18:         case $\Psi_{\{stable\ \omega\ \mathrm{x}\}}$ :
19:                 if lastState(history) $\omega$ $x$ then
20:                     return verify($\Psi$, history, $TS$, ets);
21:                 else
22:                     return (false, $\emptyset$);
23:                 end if
24:         case $\mathsf{EX}(\Psi)$ :
25:                 return verifyEX($\Psi$, history, $TS$, ets);
26:         case $\mathsf{E}(\Phi_1\ \mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi_2))$ :
27:                 return verifyEU($\Phi_1$, $\Phi_2$, $\omega$, $x$, history, $TS$, ets);
28:         case $\mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\Psi)$ :
29:                 return verifyEG($\Psi$, $\omega$, $x$, history, $TS$, ets);
30: end function
```

5.3.2 The Operator EG

The basic idea behind functions verifyEU, verifyEG, and verifyEX is quite similar: The functions accept existential statements ($\mathsf{E}\,\Phi_1\,\mathsf{U}\,\Phi_2$, $\mathsf{EG}\,\Psi$, or $\mathsf{EX}\,\Psi$) for which they search for all relevant witnesses. In the following, the function verifyEG is described in more detail. Consider the OMT-CTL formula $\mathsf{EG}(\Phi)$, where Φ can be an arbitrary OMT-CTL formula in ENF. The expression states that there is at least one path in the transition system on which Φ holds on the entire path. To check whether such paths exist, the transition system is traversed as described in Algorithm 14.

Algorithm 14 Function for the computation of the verification result for $\mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi)$ based on depth-first search

Input: OMT-CTL formula Φ in ENF, the delay constraint of operator EG, history describing the trace to the state where $\mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi)$ is verified, the finite transition system TS, and an ETS.

Output: The tuple (result, proofs) where result = true iff $TS \models \mathsf{EG}_{\{delay\ \omega\ \mathrm{x}\}}(\Phi)$ in state lastState(history), and proofs is a set of witnesses.

```
1:  function verifyEG($\Phi, \omega, x$, history, $TS$, ets)
2:      stack = create empty stack;
3:      stack.push(lastState(history));
4:      witnesses = {};
5:      while stack $\neq \emptyset$ do
6:          $\mathrm{path_{curr}}$ = stack.pop();
7:          $\mathrm{path_{curr_{full}}}$ = connect(history, $\mathrm{path_{curr}}$);
8:          (result, proofs) = verify($\Phi$, $\mathrm{path_{curr_{full}}}$, $TS$, ets);
9:          if result == true then
10:             validPathFound = false;
11:             for each $\mathrm{trans_{next}} \in$ getRelevantTransitions($\mathrm{path_{curr_{full}}}, \omega, x$, ets) do
12:                 $\mathrm{path_{next}}$ = connect($\mathrm{path_{curr}}$, $\mathrm{trans_{next}}$);
13:                 if $\neg$isCyclic($\mathrm{path_{curr}}$, ets, $\mathrm{trans_{next}}$) then
14:                     stack.push($\mathrm{path_{next}}$);
15:                 else
16:                     witnesses = witnesses $\cup$ $\mathrm{path_{next}}$;
17:                 end if
18:                 validPathFound = true;
19:             end for
20:             if ($\neg$validPathFound) then
21:                 witnesses = witnesses $\cup$ $\mathrm{path_{curr}}$;
22:             end if
23:         end if
24:     end while
25:     return ( witnesses $\neq \emptyset$, witnesses);
26: end function
```

The idea of the algorithm is quite simple: The starting point is the last state that is reached by history. From this state, the transition system is traversed and it is searched for paths on which Φ holds globally. A stack is used to manage all paths that have not been completely explored yet. The function is terminated as soon as there is no more path on the stack. Initially, the last state reached by path history is added to the stack. This is the state from which witnesses for $\mathsf{EG}(\Phi)$ are searched. $\mathrm{path_{curr}}$ contains the path that is currently being examined (starting from the state where $\mathsf{EG}(\Phi)$ is verified). $\text{path}_{\text{curr}_{\text{full}}}$ contains the entire path, i.e., history plus $\text{path}_{\text{curr}}$. For each state, it is checked whether Φ holds. This is achieved by a recursive call of function verify (line 8). Only if the result is true, the subsequent states are explored. Otherwise, the current path is skipped, as it cannot lead to a witness (the specification is already violated on the path). Function getRelevantTransitions(.) determines the admissible subsequent transitions with respect to both, the ETS and delay constraint. The ETS and delay constraint act as filters, i.e., only paths that comply with the ETS respectively delay constraint are considered during verification. If a transition leads to a true cycle (tested by function isCyclic), the resulting path is added to the witnesses (see line 16). Otherwise, the resulting path ($\text{path}_{\text{next}}$) will be added to the stack. If at least one subsequent transition exists (regardless of whether this transition leads to a cycle or not), the flag validPathFound is set to true. If there is no valid subsequent transition (validPathFound=false), the current path is a witness since Φ holds globally on it (see lines 20-22). Unlike most available model checkers that stop after the first witness is found, our approach keeps searching to find all relevant available witnesses. Finally, the result is returned. The expression $\mathsf{EG}\,\Phi$ is fulfilled once at least one witness has been found.

5.3.3 The Until and Next Operator

The function verifyEX (see Algorithm 15) for expressions of the form $\mathsf{EX}\,\Phi$ and the function verifyEU (see Algorithm 16) for expressions of the form $\mathsf{E}\,\Phi\,\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Psi$ follow the same principle as verifyEG. verifyEU searches for all paths on which Φ holds until the first occurrence of Ψ, and verifyEX searches for all transitions on which Φ has to hold at the next state.

Algorithm 15 Function for the computation of the verification result for $\mathsf{EX}(\Phi)$

Input: Finite transition system TS, OMT-CTL formula $\mathsf{EX}(\Phi)$ in ENF, an ETS and history describing the trace to the state where $\mathsf{EX}(\Phi)$ is verified.
Output: The tuple (result, proofs) where result = true iff $TS \models \mathsf{EX}(\Phi)$ in state lastState(history) and proofs is a set of witnesses.

```
1:  function verifyEX($\Phi$, history, $TS$, ets)
2:      witnesses = {};
3:      for each $\text{trans}_{\text{next}} \in$ getRelevantTransitions(history, ets) do
4:          $\text{path}_{\text{next}}$ = connect(history, $\text{trans}_{\text{next}}$);
5:          (result, proofs) = verify($\Phi$, $\text{path}_{\text{next}}$, $TS$);
6:          if result == true then
7:              witnesses = witnesses $\cup$ $\text{trans}_{\text{next}}$;
8:          end if
9:      end for
10:     return ( witnesses $\neq \emptyset$, witnesses);
11: end function
```

Algorithm 16 Function for the computation of the verification result for $\mathsf{E}\,\Phi(\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Psi)$ based on depth-first search

Input: OMT-CTL formulae Φ and Ψ in ENF, the delay constraint of operator EU, history describing the trace to the state where $\mathsf{E}\,\Phi(\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Psi)$ is verified, the finite transition system TS, and an ETS.

Output: The tuple (result, proofs) where result = true iff $TS \models \mathsf{E}\,\Phi(\mathsf{U}_{\{delay\ \omega\ \mathrm{x}\}}\,\Psi)$ in state lastState(history) and proofs is a set of witnesses.

```
 1: function verifyEU($\Phi, \Psi, \omega, x$, history, $TS$, ets)
 2:     stack = create empty stack;
 3:     stack.push(lastState(history));
 4:     witnesses = {};
 5:     while stack $\neq \emptyset$ do
 6:         $\mathrm{path_{curr}}$ = stack.pop();
 7:         $\mathrm{path_{curr_{full}}}$ = connect(history, $\mathrm{path_{curr}}$);
 8:         ($\mathrm{result}_\Psi, \mathrm{proofs}_\Psi$) = verify($\Psi$, $\mathrm{path_{curr_{full}}}$, $TS$, ets);
 9:         if $\mathrm{result}_\Psi$ == true then
10:             if $\mathrm{proofs}_\Psi \neq \emptyset$ then
11:                 for each $p_i \in \mathrm{proofs}_\Psi$ do
12:                     witnesses = witnesses $\cup$ connect($\mathrm{path_{curr}}, p_i$);
13:                 end for
14:             else
15:                 witnesses = witnesses $\cup$ $\mathrm{path_{curr}}$;
16:             end if
17:         else
18:             ($\mathrm{result}_\Phi, \mathrm{proofs}_\Phi$) = verify($\Phi$, $\mathrm{path_{curr_{full}}}$, $TS$, ets);
19:             if $\mathrm{result}_\Phi$ == true then
20:                 for each $\mathrm{trans_{next}} \in$ getRelevantTransitions($\mathrm{path_{curr_{full}}}, \omega, x$, ets) do
21:                     $\mathrm{path_{next}}$ = connect($\mathrm{path_{curr}}, \mathrm{trans_{next}}$);
22:                     if $\neg$isCyclic($\mathrm{path_{next}}$, ets, $\mathrm{trans_{next}}$) then
23:                         stack.push($\mathrm{path_{next}}$);
24:                     end if
25:                 end for
26:             end if
27:         end if
28:     end while
29:     return (witnesses $\neq \emptyset$, witnesses);
30: end function
```

5.3.4 Cycle Detection and Relevant Counterexamples

The transition systems obtained by unfolding will most likely be cyclic. Sooner or later such cycles will be encountered during verification. A cycle occurring during verification means that a state has been reached that already has been verified at some time previously. To be more concrete: If we encounter a cycle when verifying an expression of the form $\mathsf{EG}\,\Phi$, we have found a path on which Φ holds globally. However, such a path is not automatically a witness, as Example 5.3.1 will demonstrate.

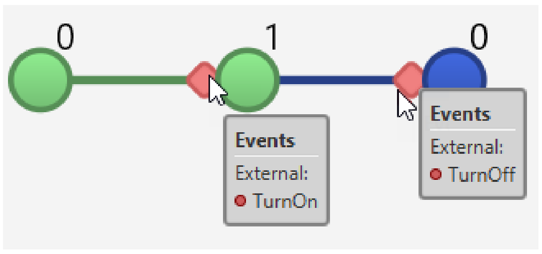
Example 5.3.1 Handling cycles during verification
Consider a simple electrical switch with the two states `On` and `Off`. This component can be modeled in smartIflow by the following component class:

```
class Switch {
    Variables:
        Enum[On, Off] state = Off;

    Events:
        TurnOn{type=operation};
        TurnOff{type=operation};

    EventHandlers:
        when(TurnOn)[state == Off]
            state = On;
        when(TurnOff)[state == On]
            state = Off;
}
```

Changes in the switching state are caused by the external events `TurnOn` respectively `TurnOff`. The following ETS allows only paths with at maximum four events of type `operation`:

```
max4Operations := path.filter(type=operation).count() <= 4;
```

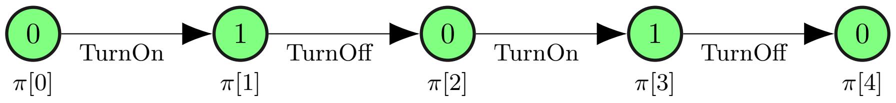
The unfolded transition system is extremely simple:



As the system model reflects only the nominal behavior of a switch, the transition system consists only of two states. In the initial state (state 0), the switch is off (`state=Off`). Triggering the external event `TurnOn` leads to a second state (state 1), where the switch is on (`state=On`). Triggering the external event `TurnOff` in state 1 leads back to the initial state. Consider the following CTL formula, which expresses that the switch can be turned on at any time:

```
SwitchCanBeTurnedOn := AG(EF state == On);
```

At first glance, one might think that the formula `SwitchCanBeTurnedOn` must be satisfied without any doubt. However, this is not the case, as the following path π[1] illustrates:



The cyclic path contains four events of type `operation` and it therefore is an admissible path with respect to ETS `max4Operations`. However, this path does not fulfill CTL formula `SwitchCanBeTurnedOn` as in $\pi[4]$ the switch is off and cannot be turned on again. This is caused by the ETS which does not allow further events to be triggered in $\pi[4]$ (the maximum number of events of type `operation` already is reached). ■

The example shows, that the exploration of a path during the verification must not be stopped immediately when a cycle is detected. In the example, stopping the exploration of the path after the first cycle has occurred would leave the violation of the specification undetected. However, the verification does not have to follow all cycles generally. In the example, the violation of the CTL formula is not caused by the model itself but by the ETS which prevents the execution of external events (if the path is traversed further, the CTL formula is fulfilled, but at the same time the ETS is violated). Cycles on which only internal events occur do not influence the ETS and therefore do not need to be followed. Cycles on which external events occur, but whose event types do not occur in the ETS, also do not have to be followed. If we had used the ETS

```
max2Failures := path.filter(type=failure).count() <= 2;
```

in Example 5.3.1, the specification would have been fulfilled. Therefore, cycles only must be checked if at least one external event occurs on the cycle whose type occurs in the ETS. Function isCyclic (see Algorithm 17) determines whether for $\text{path}_{\text{curr}}$ the transition $\text{trans}_{\text{next}}$ leads to a cycle. Function isCyclic requires the ETS used for verification to check whether external events occur on the cycle, whose event types occur in the ETS.

Initially, it is checked whether $\text{trans}_{\text{next}}$ leads to a cycle at all (regardless of whether external events occur or not). This is done checking whether $\text{trans}_{\text{next}}$ already exists on $\text{path}_{\text{curr}}$ ($\text{trans}_{\text{next}} \in \text{path}_{\text{curr}}$). Then, for each event type that occurs in the ETS, we count how many events of that type have occurred in the path up to the first and last transition of the cycle. The result is two so-called event counters. Basically, an event counter is a vector where each component represents the number of events of a certain type. If both vectors (vectors at the first and last transition of the cycle) are equal, the cycle certainly does not affect the satisfiability of the ETS. Thus, the cycle does not have to be followed during verification.

[1]Remember: $\pi[i]$ denotes the $(i+1)th$ state of path π

Algorithm 17 Function for cycle detection for verification

Input: $\text{path}_{\text{curr}}$ represented by a set of transtions, transition $\text{trans}_{\text{next}}$ and an ETS
Output: true iff $\text{trans}_{\text{next}}$ leads cycle which has no influence on the satisfiability of the ETS

```
function isCyclic($\text{path}_{\text{curr}}, \text{ets}, \text{trans}_{\text{next}}$)
    if $\text{path}_{\text{curr}}$.contains($\text{trans}_{\text{next}}$) then
        $\text{path}_{\text{next}}$ = connect($\text{path}_{\text{curr}}, \text{trans}_{\text{next}}$);
        indexCycleFirst = $\text{path}_{\text{curr}}$.lastIndexOf($\text{trans}_{\text{next}}$);
        indexCycleLast = $\text{path}_{\text{next}}$.size() - 1;
        eventCountersCycleFirst = getEventCounters($\text{path}_{\text{next}}$, ets, indexCycleFirst);
        eventCountersCycleLast = getEventCounters($\text{path}_{\text{next}}$, ets, indexCycleLast);
        if eventCountersCycleFirst == eventCountersCycleLast then
            return true;
        else
            return false;
        end if
    else
        return false;
    end if
end function
```

Example 5.3.2 Event counters
The event counters for Example 5.3.1 are quite simple. Since only one event type (`type=operation`) occurs in the ETS, the event counters consist of just one component. At each transition in the path, an event of `type=operation` is executed. As a result, the counter values along the path increase successively.

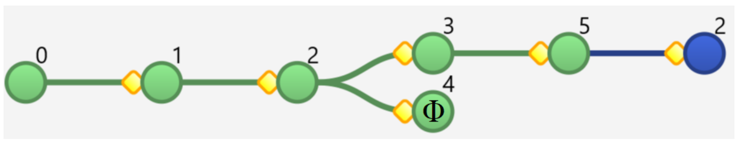
State on path	$\pi[0]$	$\pi[1]$	$\pi[2]$	$\pi[3]$	$\pi[4]$
Event counters	(0)	(1)	(2)	(3)	(4)

Since the event counters at the states where the cycles start and end ($\pi[0], \pi[2]$ and $\pi[2], \pi[4]$) have different values, it is mandatory to follow the cycles in this example. ■

Definition 5.3.1 Relevant counterexamples
Let Φ be a specification in OMT-CTL and CE_{full} is the set of all counterexamples obtained by verification of Φ. $CE_{\text{irrel}} \subset CE_{\text{full}}$ is the set of irrelevant counterexamples if each counterexample $ce_{\text{ext}} \in CE_{\text{irrel}}$ is a variant of another counterexample $ce \in CE_{\text{full}}$ extended by a cyclic subpath. The extensions of the paths do not lead to any new knowledge about the reason for the violation of the specification. Thus, these paths are irrelevant. Finally, the set of relevant counterexamples (CE_{rel}) is defined by: $CE_{\text{rel}} = CE_{\text{full}} \setminus CE_{\text{irrel}}$ ■

The proposed cycle detection ensures that all **relevant counterexamples** are detected. The following example demonstrates which counterexamples are relevant.

Example 5.3.3 Relevant counterexamples
Consider the following transition system and the specification !($\mathsf{EF}(\Phi)$):



The specification !(EF(Φ)) is obviously not fulfilled since there are paths leading to a Φ state:

$$\begin{tikzcd}
\boxed{Path_1} & 0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & \underset{\Phi}{4} & & & & \\
\boxed{Path_2} & 0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 5 \arrow[r] & 2 \arrow[r] & \underset{\Phi}{4} & \\
\boxed{Path_3} & 0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 5 \arrow[r] & 2 \arrow[r] & 3 \arrow[r, phantom, "\cdots"] & \underset{\Phi}{4}
\end{tikzcd}$$

...

In fact, there are infinite paths that disprove the specification. Consequently, all of these paths are also counterexamples. However, only the first counterexample ($Path_1$) is a relevant counterexample. $Path_1$ is a path that shows how a Φ state is reached. The other paths ($Path_2$, $Path_3$, ... $Path_n$) also show how this state is reached, but not directly but by a cyclic extension ($2 \rightarrow 3 \rightarrow 5 \rightarrow 2$). Thus, these paths do not contain any new knowledge and therefore only $Path_1$ is a relevant path. ■

To summarize, following cycles can be stopped if...

- ...the event-counters at the beginning and at the end of the cycle are equal
- ...a counterexample/witness has been found
- ...the Event Trigger Specification or timing constraint is violated

5.4 Minimal Cutsets and Fault Tree Generation

The verification of real-world systems will likely produce a lot of counterexamples. The number of counterexamples depends first on the model and the specification, of course, but the ETS also has a major impact. An ETS allowing multiple failure events on a path will probably lead to more counterexamples than allowing nominal behavior only. The resulting counterexamples are often very similar, as only the order of events on the paths differs.

Example 5.4.1 Relevant information in counterexamples
Consider the following system model that consists of variable (`var`), three external events (`ActivateFailure`, `ActivateMode1` and `ActivateMode2`), and according event handlers to react to the events:

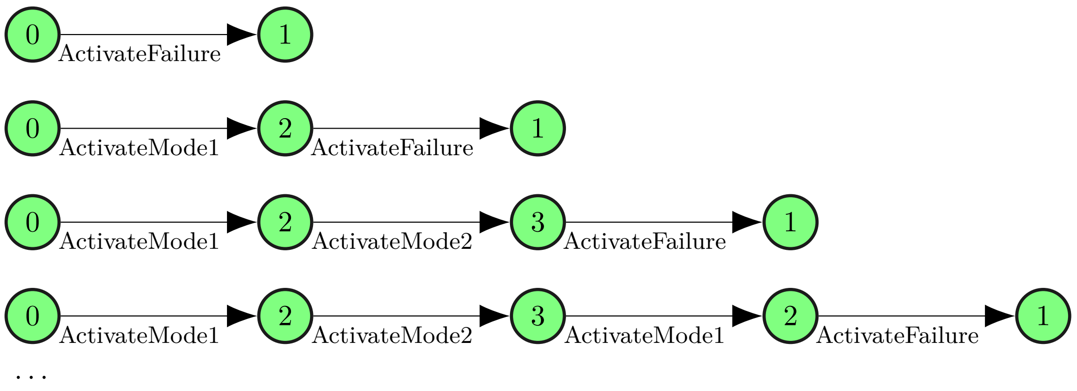
```
class Demonstrator {
    Variables:
        Enum[Init, Mode1, Mode2, Failure] var = Init;

    Events:
        ActivateFailure{type=failure};
        ActivateMode1{type=operation};
        ActivateMode2{type=operation};

    EventHandlers:
        when(ActivateMode1)[var != Mode1 && var != Failure] {
            var = Mode1;
        }
        when(ActivateMode2)[var != Mode2 && var != Failure] {
            var = Mode2;
        }
        when(ActivateFailure)[var != Failure] {
            var = Failure;
        }
}
```

Initially, the system is in state `Init`. In state `Init`, it is possible to switch to state `Mode1` or `Mode2`. A change between `Mode1` and `Mode2` is possible at any time. The failure mode `var=Failure` can also be activated at any time. However, as soon as the system is in state `Failure`, it is no longer possible to switch back to `Mode1` or `Mode2`.

The following OMT-CTL formula states, that the failure mode will never be activated (on all paths there must be no state in which `var=Failure`).

```
Spec1 := AG(var != Failure);
```

Since the failure mode can be activated at any time, there will be some paths where this condition does not hold. Using an ETS allowing at maximum four events of type `operation`, the model checker delivers 9 counterexamples which disprove specification `Spec1`[2]. Consider the following paths that are part of the counterexamples:



The counterexamples are quite different. There are both short and long paths in the counterexamples. Generally, the counterexamples show **all** possible paths that lead at the end to the state where `var=Failure`. However, the event `ActivateFailure` is solely responsible for the violation of the specification. This message is essential for the safety

[2]Because of the ETS, cycles must be followed during verification (see previous section) and therefore counterexamples can be cyclic.

engineers. It's nice to have the list of all relevant counterexamples, but what matters are the minimal cutsets, i.e., the sets of critical events that lead to the violation of the specification. ■

Remember: A cutset describes a set of basic events (failure events) that together lead to the violation of a specification. A minimal cutset consists of the smallest set of basic events that together lead to the specification violation.

Algorithm 18 Algorithm for the generation of minimal cutsets

```
Input:  Set of counterexamples
Output: Set of minimal cutsets
function generateMinimalCutsets(counterexamples_full)
    // First determine all cutsets...
    cutsetMap= create empty map;
    for each $ce_i \in$ counterexamples_full do
        cutset_{$ce_i$}= filter($ce_i$); // Get all failure events of $ce_i$
        if cutsetMap.contains(cutset_{$ce_i$}) then
            cutsetMap.get(cutset_{$ce_i$}).add($ce_i$);
        else
            cutsetMap.addEntry(cutset_{$ce_i$}, {$ce_i$});
        end if
    end for

    // ...then determine minimal cutsets
    noMinimalCutsets= create empty set;
    for each ( cutset_1, counterexamples_1) $\in$ cutsetMap.entries() do
        for each ( cutset_2, counterexamples_2) $\in$ cutsetMap.entries() do
            if cutset_1 $\subset$ cutset_2 then
                // cutset_2 is definitely no minimal cutset
                noMinimalCutsets.add(cutset_2);
                cutset_1.addCounterexamples(counterexamples_2);
            end if
        end for
    end for
    cutsetMap.removeAll(noMinimalCutsets);
    return cutsetMap;
end function
```

Algorithm 18 shows a possible implementation for the calculation of the minimal cutsets based on a set of counterexamples. First, all cutsets are determined. A cutset is a set of failure events (external events to which the feature `type=failure` is assigned) that cause the specification violation. These cutsets are organized in a map (cutsetMap) that maps cutsets to corresponding counterexamples. In fact, it would be enough to remember only the cutsets. However, it may be interesting for safety engineers to view the corresponding counterexamples for a cutset. For each counterexample, the corresponding cutset is calculated using function filter. This function filters events in a counterexample to include only failure events (see line 5). Finally, cutsetMap contains all cutsets and its corresponding counterexamples. After that, the minimal cutsets are determined by comparing all cutsets with each other. If cutset$_1$ is a subset of cutset$_2$, cutset$_2$ is definitely not a minimal cutset. Therefore, cutset$_2$ is added to set noMinimalCutsets. On the other hand, cutset$_1$ is not automatically a minimal cutset, because there may exist cutsets which are subsets of cutset$_1$. Finally, the set of minimal cutsets is defined by all cutsets that are in cutsetMap but not in noMinimalCutsets.

The minimal cutsets from the counterexamples of a specification violation can be represented as a Fault Tree. The events of a minimal cutset are basic events and linked using an AND gate (the specification is only violated if all events occur). The outputs of these AND gates are linked using an OR gate and the top event is the specification (=undesired state).

Example 5.4.2 Representation of minimal cutsets as Fault Tree
Consider the following minimal cutsets, obtained by verification of specification TankOverflowShallNotOccur:

1. sensorE.ActivateDisconnected, sensorF.ActivateDisconnected
2. valve.ActivateStuckClosed
3. sensorE.ActivateStuckFalse, sensorF.ActivateDisconnected
4. sensorF.ActivateStuckFalse

The corresponding Fault Tree for these minimal cutsets is shown in Figure 5.2. We used the tool *Abre Analyste*[3] for rendering the Fault Tree. Using features, probabilities of occurrence can be assigned to the external events. These values can be used by the FTA tools for the calculation of the probability of the undesirable event.

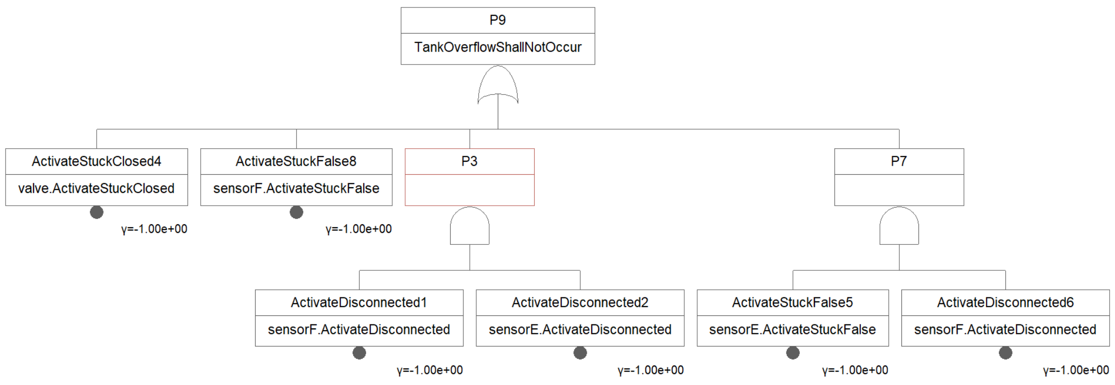


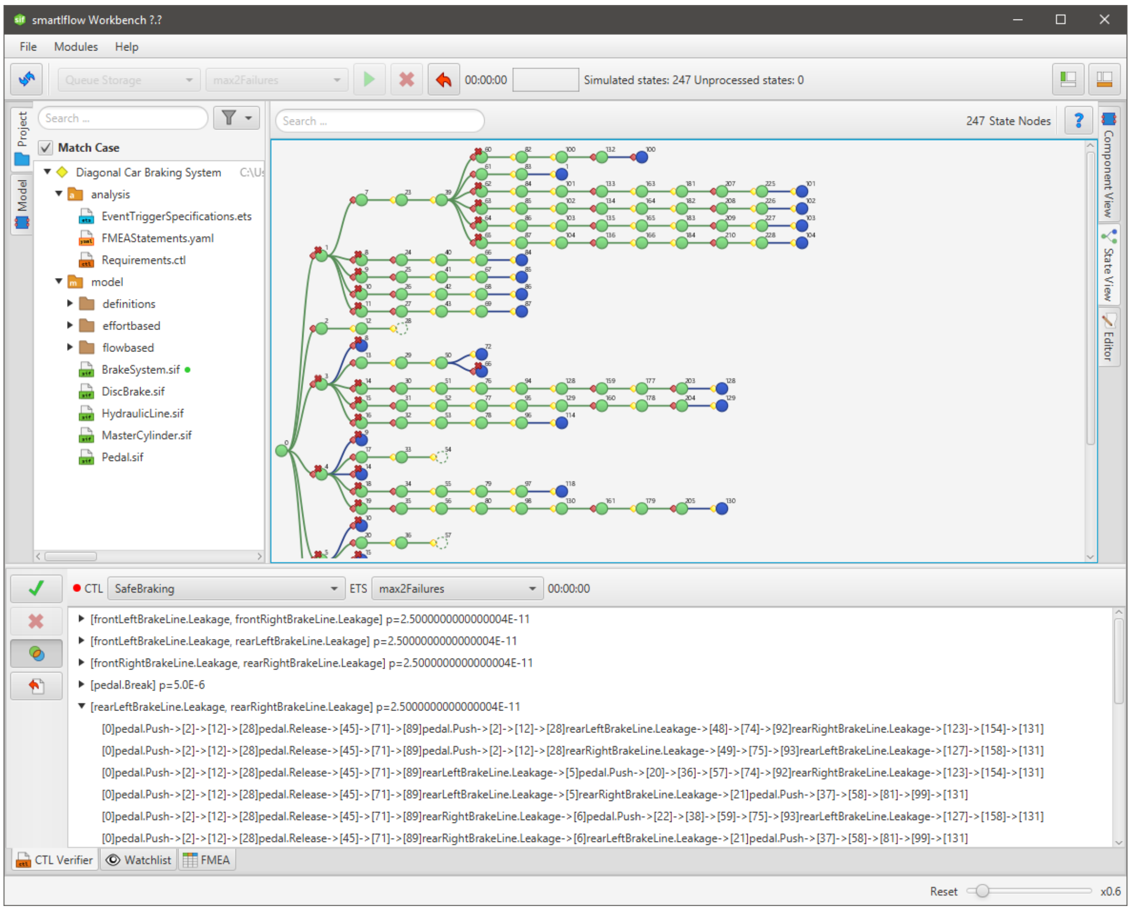
Figure 5.2: Minimal cutsets represented as Fault Tree

■

[3]https://www.arbre-analyste.fr/en.html

6 Case Studies

All concepts presented in this thesis and several others are implemented in the smartIflow Workbench[1]. The smartIflow Workbench is an integrated development environment (IDE) that supports the creation, unfolding, analysis, and verification of smartIflow system models. An integrated editor with instant feedback about syntax errors assists in the development of models. Different views such as a change history or watch list can be used to debug models. These views also help to understand how variable values, and properties at ports or connections evolve on a path in the transition system.



```
[frontLeftBrakeLine.Leakage, frontRightBrakeLine.Leakage] p=2.5000000000000004E-11
[frontLeftBrakeLine.Leakage, rearLeftBrakeLine.Leakage] p=2.5000000000000004E-11
[frontRightBrakeLine.Leakage, rearRightBrakeLine.Leakage] p=2.5000000000000004E-11
[pedal.Break] p=5.0E-6
[rearLeftBrakeLine.Leakage, rearRightBrakeLine.Leakage] p=2.5000000000000004E-11
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]pedal.Push->[2]->[12]->[28]rearLeftBrakeLine.Leakage->[48]->[74]->[92]rearRightBrakeLine.Leakage->[123]->[154]->[131]
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]pedal.Push->[2]->[12]->[28]rearRightBrakeLine.Leakage->[49]->[75]->[93]rearLeftBrakeLine.Leakage->[127]->[158]->[131]
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]rearLeftBrakeLine.Leakage->[5]pedal.Push->[20]->[36]->[57]->[74]->[92]rearRightBrakeLine.Leakage->[123]->[154]->[131]
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]rearLeftBrakeLine.Leakage->[5]rearRightBrakeLine.Leakage->[21]pedal.Push->[37]->[58]->[81]->[99]->[131]
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]rearRightBrakeLine.Leakage->[6]pedal.Push->[22]->[38]->[59]->[75]->[93]rearLeftBrakeLine.Leakage->[127]->[158]->[131]
  [0]pedal.Push->[2]->[12]->[28]pedal.Release->[45]->[71]->[89]rearRightBrakeLine.Leakage->[6]rearLeftBrakeLine.Leakage->[21]pedal.Push->[37]->[58]->[81]->[99]->[131]
```

Figure 6.1: smartIflow Workbench showing the state space and verification result with cut sets and counterexamples

[1]An evaluation version of smartIflow Workbench is available for free download under `https://smartiflow.bitbucket.io/`

The state view (see the upper part of Figure 6.1) visualizes the state space in a tree structure where dark blue nodes represent references to other nodes in the tree. Yellow diamonds indicate internal events and red diamonds indicate external events. Nodes with red stars and diamonds indicate an external failure event. A separate view shows the results of the verification (see the lower part of Figure 6.1) with minimal cut sets and counterexamples. In addition, traces of counterexamples can be highlighted in the tree structure. smartIflow Workbench can export the minimal cut sets into different formats, among others Open-PSA (Open Probabilistic Safety Assessment[2]) is supported. The Open-PSA initiative defined a standard format for safety-related models that is supported by many Fault Tree tools (e.g., Arbre Analyste[3] or FaultCat[4]). The smartIflow Workbench is implemented in Java and thanks to the Java Platform Module System (JPMS) introduced in version 9, new features can be easily integrated, even during runtime.

The following experiments have been performed on a standard computer with an Intel Core i7-9700K, 3.6GHz Octa Core, and 64 GB RAM. Arbre Analyste was used to visualize the verification results as a Fault Tree. Note that in the code snippets import and package statements are omitted.

6.1 The Diagonal Car Braking System

Nowadays, most cars are equipped with a hydraulic brake system based on two independent brake circuits. These systems are also known as dual-circuit braking systems. There are either circuits for the front and rear axle or a diagonal division (X split) is made. The latter one is examined in this case study. Two independent brake circuits enable safe braking of the vehicle even if one of the brake circuits fails.

Figure 6.2 depicts the structure of a diagonal car braking system. The system consists of two independent hydraulic circuits in which each front brake is connected to the opposed rear brake. The master cylinder converts non-hydraulic pressure from the pedal into hydraulic pressure and a vacuum booster assists braking effort. Each circuit has its own reservoir filled with hydraulic fluid. Hydraulic pressure is conducted over hydraulic lines and thus force is applied to the calipers at the disc brakes.

A hazardous situation can be described as follows: “No braking effect after pressing the brake pedal.” How likely is it, that this hazard will occur, and how many components must fail? Is there a single point of failure? In order to answer these questions, this system is modeled using smartIflow Workbench. In principles, there are many ways to model the system. The steps to create a quite simple system model based on properties are explained hereafter.

6.1.1 Towards a simple System Model based on Properties

Let us start with the identification of necessary components and their responsibilities. For a simple system model at least the following components are required:

[2]The format is described in detail under `https://open-psa.github.io/mef/`

[3]`https://www.arbre-analyste.fr/en.html`

[4]`http://www.iu.hio.no/FaultCat`

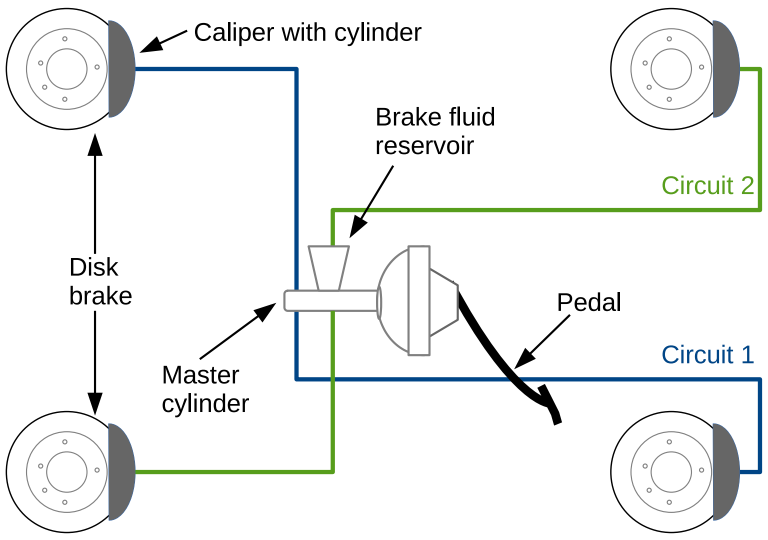


Figure 6.2: The diagonal car braking system

- Disk brake: The brake operates if pressure is available.
- Hydraulic line: It conveys pressure from the master cylinder to the disk brakes. The hydraulic line can start leaking (e.g., caused by corrosion, material fatigue, or damaged pipe joints).
- Master cylinder: Converts non-hydraulic pressure from the pedal into hydraulic pressure. A leakage in a hydraulic line causes the reservoirs to drain.
- Pedal: Can be pressed or released. If the pedal is pressed, pressure is conveyed to the master cylinder. Extremely unlikely but still possible is, that the pedal could break due to material fatigue.
- Diagonal car braking system: Main component where all sub-components are interconnected.

The next step is to define the connection points (ports), variables, and events. In principle, two port types are necessary, namely `Mechanical` and `Hydraulical`. The former is required for the connection between the pedal and the master cylinder. `Hydraulical` is used for ports of master cylinder, hydraulic line, and disc brake.

External events are used for pedal operations (press, release) as well as for the activation of the failure modes.

The variables and their domain to define the operational mode respectively the failure mode of a component are:

- **DiskBrake**
  - opm: [Idle, Braking]
- **HydraulicLine**

  - fm: [Ok, Leakage]
- **Pedal**
  - fm: [Ok, Broken]
  - opm: [Idle, Pressed]
- **MasterCylinder**
  - fm: [Ok, Delayed]
  - opm: [Idle, Operated]
  - circuit1ReservoirLevel: [Full, Empty]
  - circuit2ReservoirLevel: [Full, Empty]

For the sake of simplicity, it is assumed, that the disk brakes do not fail and the master cylinder only has a failure mode with delayed reaction. In reality, the disk brakes and the master cylinder can fail in many more different ways.

The hydraulic fluid can be implemented in two different ways. Either the qualitative energy flow analysis concept is applied or the hydraulic pressure is abstracted by use of properties. The former one has the advantage, that distinctions between pressure levels resulting from different pedal positions can be made. However, to answer the preceding questions, it is sufficient to know whether pressure at the disk brakes is present or not. Beyond that, the additional pressure generated by the vacuum booster can be neglected, too. Therefore, it is sufficient to model the pressure in the pipes using a property. The idea is to publish the property `pressure=true` if the pedal is pressed. The property is propagated from the master cylinder over the hydraulic line to the disk brakes. The behavior of the master cylinder can be expressed in smartIflow as follows:

```
Behavior:
    if (opm == Operated && circuit1ReservoirLevel == Full)
        set(circuit1, {pressure=true});
    if (opm == Operated && circuit2ReservoirLevel == Full)
        set(circuit2, {pressure=true});
```

The existence of property `pressure=true` at the disk brakes indicates that pressure is present and therefore the brake operates. Conversely, if there is no property (i.e., there is no pressure), the brake does not operate. The corresponding transition statements for the disc brake can be defined as follows:

```
Transitions:
    when (cylinder.exists(pressure=true)) [opm==Idle]
        opm = Braking;
    when (!cylinder.exists(pressure=true)) [opm==Braking]
        opm = Idle;
```

The leakage in the hydraulic line can be realized simply by publishing the property `leakage=true`:

```
Behavior:
    connect(p1, p2);
    if (fm == Leakage)
        set(p1, {leakage=true});
```

The property is propagated to both, the disk brake and to the master cylinder, but only the master cylinder will react to it. If the master cylinder receives the property `leakage=true` at one of the circuits, the corresponding reservoir drains:

```
Transitions:
    when (circuit1.exists(leakage=true) && circuit1.exists(pressure=true))
        circuit1ReservoirLevel = Empty;
    when (circuit2.exists(leakage=true) && circuit2.exists(pressure=true))
        circuit2ReservoirLevel = Empty;
```

6.1.2 Verification Results

The time for unfolding the system model without a limiting Event Trigger Specification (ETS) is negligible (< 100 milliseconds) and the result is 348 different reachable states. The safety requirement “Whenever the pedal is pressed, at least one brake on each side must operate.” can be specified in OMT-CTL as follows:

```
brakeOnEachSideActive :=
    (frontLeftBrake.opm == Braking || rearLeftBrake.opm == Braking) &&
    (frontRightBrake.opm == Braking || rearRightBrake.opm == Braking);

SafeBraking:=
    AG(pedal.opm == Pushed -> AF brakeOnEachSideActive);
```

For the sake of clarity, the requirement is separated into two parts. The expression `SafeBraking` requires that all states on all paths must fulfill the following implication: Whenever the pedal is pushed, at least one brake on each side must operate.

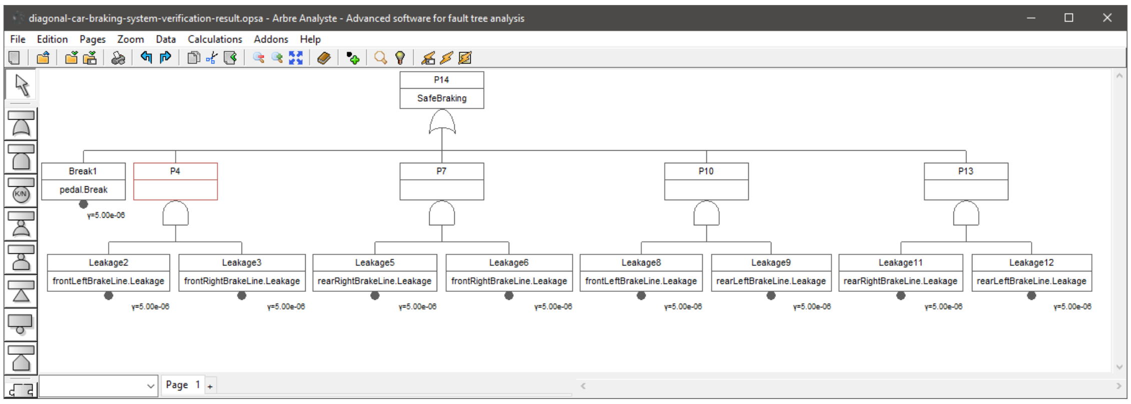

diagonal-car-braking-system-verification-result.opsa - Arbre Analyste - Advanced software for fault tree analysis
File Edition Pages Zoom Data Calculations Addons Help
P14
SafeBraking
Break1
pedal.Break
γ=5.00e-06
P4
P7
P10
P13
Leakage2
frontLeftBrakeLine.Leakage
Leakage3
frontRightBrakeLine.Leakage
Leakage5
rearRightBrakeLine.Leakage
Leakage6
frontRightBrakeLine.Leakage
Leakage8
frontLeftBrakeLine.Leakage
Leakage9
rearLeftBrakeLine.Leakage
Leakage11
rearRightBrakeLine.Leakage
Leakage12
rearLeftBrakeLine.Leakage
γ=5.00e-06
Page 1

Figure 6.3: Fault Tree for top event `SafeBraking`

As expected, some counterexamples disprove this specification. Overall, the verification generated 50277 counterexamples within 2025 milliseconds[5]. The results show that the break of the pedal alone leads to a violation of the specification. This is plausible because

[5]Without any limiting ETS

with a broken pedal, no pressure is generated at the master cylinder and therefore the brakes cannot operate anymore. The good news is, that this event is extremely rare but nevertheless, special attention should be paid during the development and production of this part. All remaining counterexamples consist of arbitrary combinations of leakage in each circuit. A leakage in each circuit causes both to leak and therefore no more pressure can be generated when pressing the pedal. Figure 6.3 shows the complete Fault Tree for the event `SafeBraking`.

6.2 Airbag System

Even though an airbag system is not mandatory, almost all new vehicles are usually equipped with one, especially in small cars. In case of a crash, the airbag is deployed automatically within a few milliseconds. This prevents the passengers from colliding with hard parts of the interior such as the dashboard or steering wheel. Figure 6.4 shows in simplified form the architecture of an airbag system as described in [Alj+09]. There are three types of subsystems, namely sensors, evaluation, and actuators. The impact in a crash situation is detected by acceleration sensors. Two redundant microcontrollers evaluate the sensor signals and decide whether the airbag needs to be triggered or not. The microcontroller sends the decision to each other and only if both come to the same decision, the airbag is actually triggered. Airbag activation consists of a FET (Field Effect Transistor) that delivers the power for the FASIC (Firing Application Specific Integrated Circuit) to ignite the airbag.

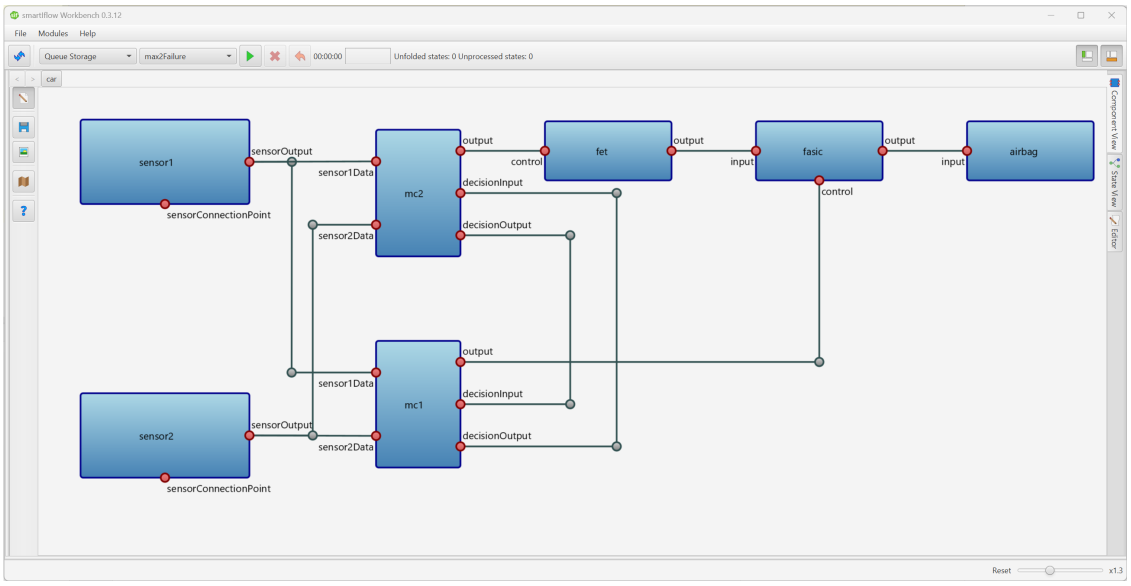


Figure 6.4: Schematic structure of an airbag system

The following failures can occur in the system:

- The acceleration sensor delivers falsified values. For example, the sensor generates a value that indicates high acceleration even though the vehicle does not move at all or moves at constant speed.

- A microcontroller sends a fire command even though both sensors detect a low acceleration. The other case is also possible, where both sensors detect a crash situation, but the microcontroller does not send a fire command. Incorrect fire commands also occur when both sensors deliver inconsistent values (e.g., sensor1: acceleration=normal; sensor2: acceleration=high).
- FASIC or FET suppress a fire command or inadvertently activate their output pin.

The airbag system is designed to protect the life of passengers in case of a crash. However, the system can also lead to critical situations in case of a malfunction. Basically, there are two important safety requirements for the airbag system:

1. The airbag must always be triggered in case of an accident. If not, the passengers are not protected from the impact.
2. The airbag must not be deployed unintentionally. If the airbag is triggered unintentionally while driving, the driver can lose control of the vehicle and may cause an accident.

6.2.1 Modeling the Control Logic in smartIflow

Component types are needed for sensors, microcontroller, FET, FASIC, and the airbag. Finally, the whole system itself is described in class `Car`.

The flexible property concept has also proven to be very helpful in modeling this system. Sensor signals as well as communication between microcontroller and airbag deployment can be realized using properties. For the sensors, it is sufficient to differentiate between high and normal acceleration. These values are evaluated by the microcontroller in the following way:

```
class Microcontroller {
  Ports:
      Logical decisionInput;
      Logical decisionOutput;

      Logical output;

      Logical sensor1Data;
      Logical sensor2Data;

    Variables:
       Enum[Fire, Idle] decisionThis = Idle;
       Enum[Fire, Idle] decisionOther = Idle;

       Enum[Normal, FailureDetected] om = Normal;
       Enum[Ok, StuckOnFire, StuckOnIdle] fm = Ok;

    Behavior:
```

```
19         if (decisionThis == Fire && om == Normal && fm == Ok)
20             set(decisionOutput, {airbag=fire});
21 
22         if (decisionThis == Fire && decisionOther == Fire
23                                       && fm == Ok && om == Normal)
24             set(output, {airbag=fire});
25 
26         if (fm == StuckOnFire) {
27             set(output, {airbag=fire});
28             set(decisionOutput, {airbag=fire});
29         }
30 
31     Transitions:
32         /* Evaluate sensor signal */
33         when (sensor1Data.exists(acceleration=high)
34                   && sensor2Data.exists(acceleration=high) && fm == Ok)
35               decisionThis = Fire;
36 
37         /* Evaluate signal from other microcontroller */
38         when (decisionInput.exists(airbag=fire) && fm == Ok)
39               decisionOther = Fire;
40 
41         /* Error detection */
42         when (sensor1Data.exists(deviation=true)
43             || sensor2Data.exists(deviation=true)
44             || decisionThis == Fire && decisionOther == Idle
45             || decisionThis == Idle && decisionOther == Fire) [fm == Ok] {
46             om = FailureDetected;
47         }
48 
49     EventHandlers:
50         when (ActivateStuckOnFire) [fm == Ok] {
51             fm = StuckOnFire;
52             decisionThis = Fire;
53         }
54 
55         when (ActivateStuckOnIdle) [fm == Ok] {
56             fm = StuckOnIdle;
57             decisionThis = Idle;
58         }
59 }
```

Each microcontroller stores its fire decision and the decision of the other microcontroller in a variable. The helper variables (`decisionThis` and `decisionOther`) are required since smartIflow does not allow sending messages as a direct reaction to events. Only if both microcontrollers come to the same decision, a fire command is sent to FASIC and FET. In case one microcontroller detects that its decision differs from the decision of the other microcontroller, the component changes its operation mode to `FailureDetected` (see lines 42-47). This causes the airbag indicator bulb to light up, alerting the driver that a workshop should be visited as soon as possible.

A fire command modeled with the property `airbag=fire` enables the FET to deliver power to the FASIC to deploy the airbag. The following transition statement changes the state of FET to `Close` when receiving the fire command:

```
when (control.exists(airbag=fire)) [fm == Ok]
    om = Close;
```

The fire command in FASIC transmits the electrical power to the airbag which is then ignited.

An open question is, how a crash situation can be modeled and how the sensors get knowledge about the crash. A simple and intuitive solution is to add an external event `Crash` in component `Car`. This event initiates a crash situation and propagates the property `acceleration=high` via a connection to the sensors. In a sense, the connection represents the mechanical link between the sensor and the car body. The proposed mechanism for crash activation in component `Car` can be implemented in smartIflow as follows:

```
//...
    Behavior:
    //...
        if (om == Ok || om == Crashed) {
            set(sensor1.sensorConnectionPoint, {acceleration=normal});
            set(sensor2.sensorConnectionPoint, {acceleration=normal});
        } else if (om == Crash) {
            set(sensor1.sensorConnectionPoint, {acceleration=high});
            set(sensor2.sensorConnectionPoint, {acceleration=high});
        }
    Transitions:
        when (om == Crash)
            om = Crashed;

    EventHandlers:
        when (Crash) [om == Ok]
            om = Crash;
}
```

The sensor just passes the property to the microcontroller where the actual processing takes place. As usual, component failures are activated by external events:

- Microcontroller: `ActivateStuckOnFire, ActivateStuckOnIdle`
- FASIC: `ActivateInadvertentlyFailure, ActivateSupressFireCommand`
- FET: `ActivateInadvertentlyCloseFailure, ActivateStuckOpenedFailure`
- AccelerationSensor: `ActivateStuckHighFailure, ActivateMechanicalFailure, ActivateWrongAmplitude`

6.2.2 Verification Results

Unfolding takes 161 milliseconds and results in 694 different states (using an ETS allowing a maximum of two failure events on a path). The requirement specifications for no inadvertent airbag activation and correct activation in case of a crash can be formulated in OMT-CTL as follows:

```
CorrectAirbagActivation := AG(om==Crash -> AF(airbag.om==Fired));
NoInadvertentAirbagActivation := AG(om==Ok -> airbag.om!=Fired);
```

The model checker generates counterexamples for both requirements. Figure 6.5 shows the Fault Tree for top event `NoInadvertentAirbagActivation`.

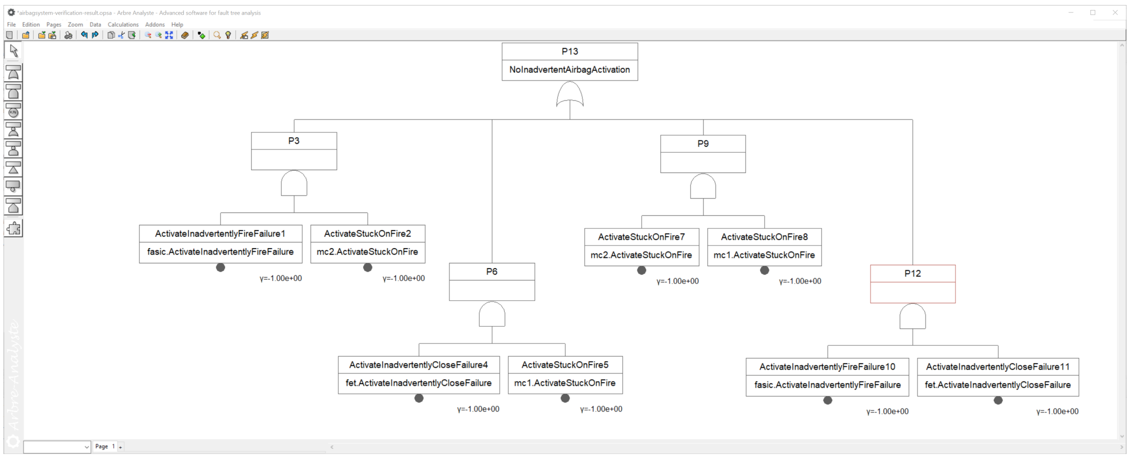

*airbagsystem-verification-result.opsa - Arbre Analyste - Advanced software for fault tree analysis
File
Edition
Pages
Zoom
Data
Calculations
Addons
Help
P13
NoInadvertentAirbagActivation
P3
P9
ActivateInadvertentlyFireFailure1
fasic.ActivateInadvertentlyFireFailure
γ=-1.00e+00
ActivateStuckOnFire2
mc2.ActivateStuckOnFire
γ=-1.00e+00
ActivateStuckOnFire7
mc2.ActivateStuckOnFire
γ=-1.00e+00
ActivateStuckOnFire8
mc1.ActivateStuckOnFire
γ=-1.00e+00
P6
P12
ActivateInadvertentlyCloseFailure4
fet.ActivateInadvertentlyCloseFailure
γ=-1.00e+00
ActivateStuckOnFire5
mc1.ActivateStuckOnFire
γ=-1.00e+00
ActivateInadvertentlyFireFailure10
fasic.ActivateInadvertentlyFireFailure
γ=-1.00e+00
ActivateInadvertentlyCloseFailure11
fet.ActivateInadvertentlyCloseFailure
γ=-1.00e+00
Arbre-Analyste
Page 1

Figure 6.5: Fault Tree for top event `NoInadvertentAirbagActivation`

The good news is, that single faults cannot lead to inadvertent airbag deployment. A combination of two component failures is required to ignite the airbag in a non-crash situation.

However, if any component fails, there is no guarantee, that the airbag will be deployed in case of an accident. This is reasonable, as the redundancy in the airbag system is designed to prevent the airbag from being triggered unintentionally. Consequently, the failure of a component can prevent the airbag from being triggered in the event of a crash. However, most airbag systems automatically detect faults in the system and indicate this to the driver in the instrument cluster.

6.3 Radio-Based Railroad Crossing

There are different ways to realize railway crossing protection systems. Basically, a distinction is made between signal-controlled and train-controlled types. In signal-controlled variants, a control center supervises the state of gates and signals. For the train-controlled version (see Figure 6.6) redundant sensors (inductive loops) that can detect the presence of a train are installed before and after the crossing.

If the train-approaching sensor recognizes the train, traffic lights are switched on and gates are closed. The gates are reopened when the train-leaving sensor detects the train. A signal indicates the conductor whether the crossing is safe (gates must be closed). The distance of the signal to the railway crossing is chosen in such a way, that the train can be stopped before the railway crossing in case of an emergency brake. This implementation is

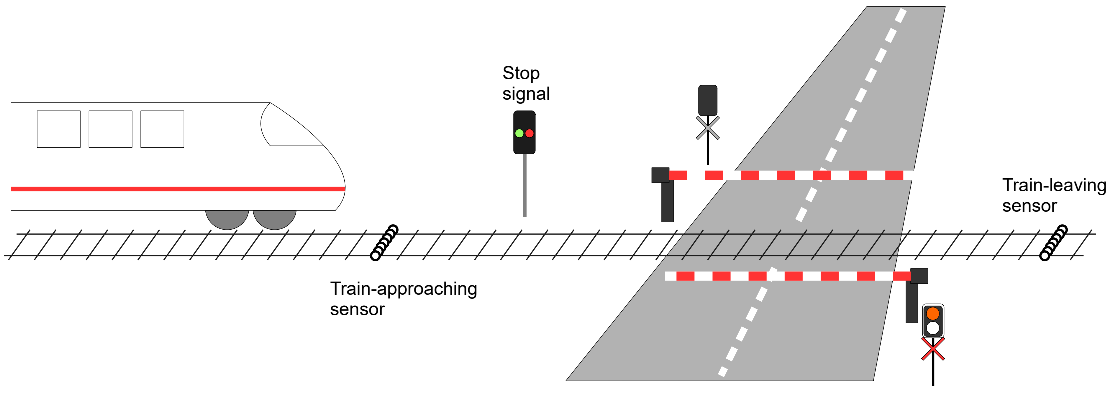


Figure 6.6: Traditional design of a railroad crossing system based on sensors and signals

quite expensive and leads to long waiting times for cars. As soon as the train-approaching sensor detects a train, the gates are immediately closed, regardless of the speed of the train. This causes long waiting times when a slow train passes the crossing. The slower the train, the longer the cars have to wait. Actually, these long waiting times are not necessary, as the braking distance decreases with slower speeds, and therefore the gates can be closed later.

A new variant of the railroad crossing system based on radio communication attempts to overcome these problems (see Figure 6.7) [ORS05]. The strategy is as follows: The train knows its position and speed (using GPS sensors) and the position of the crossing. If the train knows the time needed to close the gates, it can determine the perfect moment for closing the gates. The ideal time to close the gates is when the waiting time for cars is as short as possible, but at the same time, there is still enough time for emergency braking.

When approaching the gate, the train sends a close request to the crossing. Once the crossing receives this signal, the gates are closed. To ensure that the gates are closed, the train sends a status-request to the crossing. A position sensor installed at the gates provides information about the actual state (opened/closed). The crossing utilizes this sensor information to respond to the status-request. In case the gates are not closed, an emergency break is performed. Otherwise, the train passes the crossing and the train-leaving sensor causes the crossing to open the gates. The gates are automatically reopened after some time if no further train actions occur in the meantime (e.g., the train-leaving sensor has detected no train).

A significant advantage of this variant is, that the railroad crossing protection system is activated individually by each train depending on its actual speed. In theory, this results in a shorter waiting time for cars.

Of course, component failures can still occur:

- The gate-closed sensor delivers an erroneous value that does not reflect the actual state of the gate.
- The train-leaving sensor detects a train even though there is no train.

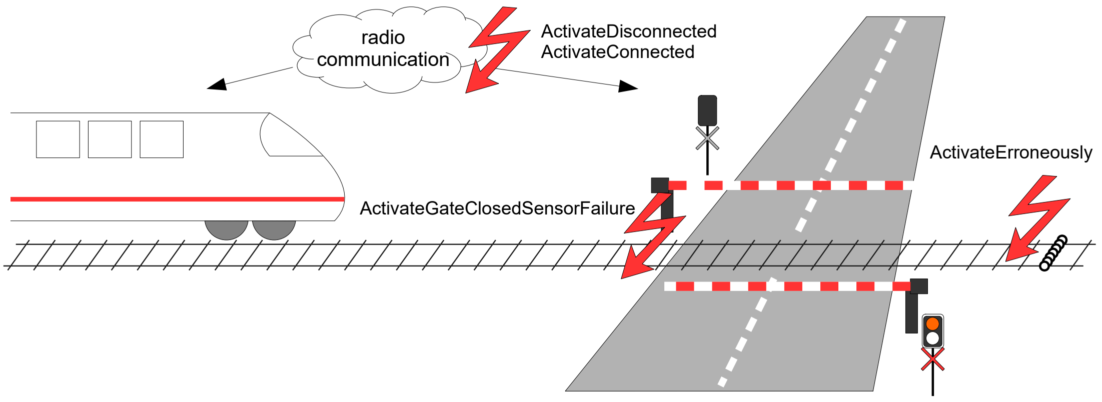


Figure 6.7: Components and potential failure events in the railroad crossing model

- Radio communication may be temporarily unavailable.

The train may only pass the crossing when the gates are closed. The question is whether this planned strategy is safe. Is it possible that a temporary network failure leads to disaster?

6.3.1 Modeling using different Orders of Magnitude of Time

Let us start with the identification of relevant components[6]. Obviously, component types for the main parts of the system are required (`Train` and `Crossing`). Beyond that, types for the radio network and the position sensor are necessary (`RadioNetwork` and `PosSensor`). Both components can fail and therefore behavior needs to be specified. A separate type for the gates is not necessary, since a variable with information about the current state of the gate (opened/closed) in type `Crossing` is sufficient. In class `Main` the whole system is described, i.e., all component types are instantiated and connections are established:

```
class Main {
    Components:
        Train train;
        Crossing crossing;
        RadioNetwork network;
        PosSensor trainLeavingSensor;
    Behavior:
        connect(train.radioConnection, network.p1);
        connect(crossing.radioConnection, network.p2);
        connect(train.isPassingSignal, trainLeavingSensor.in);
        connect(crossing.isTrainPassingSignal, trainLeavingSensor.out);
}
```

Concerning the train, variables for position and speed are required. For speed, three different levels are sufficient: `Moving`, `SlowingDown`, and `Stopped`. With respect to the

[6]The smartIflow system model was first introduced in [LHM18]

position a differentiation between zones `Far`, `Approaching`, `NearEmergencyBrakingZone`, `InDangerZone`, and `Passed` is made. Besides the variables `speed` and `position` two helper variables (`requestingClose` and `requestingStatus`) are necessary. The helper variables are helpful in two aspects: ① They help to limit the number of messages. Only if such a variable is true, the corresponding message is sent. ② "smartIflow does not allow to send messages as direct reaction to events. Instead, messages are state-dependent." [LHM18] Once an event has occurred, the appropriate value is assigned to the helper variable, and in the state-dependent behavior, the corresponding messages are sent.

In the systems shown so far, delays in the transition statement for the control logic have been neglected (the default value for delay is 1). For the radio-controlled railroad crossing system, specific delays are mandatory as the following example illustrates: As already described in the previous section, gates reopen after a specified time without train interactions. This logic can be easily described with the following transition statement:

```
when {delay=5} (gateState==Closed) [gateState==Closed]
    gateState = Opened;
```

Once the state of the gate is `Closed`, the transition statement is triggered. However, the transition is not executed directly, but with a delay of five orders of magnitude above reference. Before this transition is executed, all active transition statements with a delay of less than magnitude five are executed. Guard conditions are permanently rechecked until they are finally executed. In this case, the guard condition ensures that the transition is only executed when the gate is in state `Closed`. When the transition is executed, the state of the gate is changed from `Closed` to `Opened`. Without different delays, the result would be that the gates would reopen immediately after closing. Obviously, this is not the intended behavior. Overall, quite different orders of magnitude for delays are required for this system. For example, moving from one zone to another takes much more time than communication between trains and crossing via a radio network. The waiting time for the response to a sent message also must be longer than the delay during sending a message. How many different orders of magnitude are needed in total? Table 6.1 describes the set of orders of magnitude of time used in the system model.

Relevant transitions	Delay
Setting helper variables for sending messages via radio network	OMT=1
Waiting time for the response to a sent message	OMT=2
Changing the state of the gate (`Opened`, `Closing`, `Closed`) or changing the speed of the train (e.g., from state `Moving` to `SlowingDown`)	OMT=3
Train position changes (e.g., from `Far` to `Approaching`)	OMT=4
Reopen gates if the train-leaving sensor detected no train in the meantime (e.g., due to a defective sensor)	OMT=5

Table 6.1: Different orders of magnitude of time in radio-controlled railroad crossing system

The movement of the train from one zone to the next is realized by internal events. This is reasonable as the train moves independently (without external influence) depending on the current speed. Nevertheless, a solution with external events is also possible. The communication between trains and crossing over the radio network is realized again with the help of properties. The component class definition for the train is sketched below:

```
 1 class Train {
 2     Ports:
 3         Bus radioConnection;
 4         BooleanSignal{dir=output} isPassingSignal;
 5 
 6     Variables:
 7         Enum[Far, Approaching, NearEmergencyBrakingZone, InDangerZone, Passed]
               position = Far;
 8         Enum[Moving, SlowingDown, Stopped] speed;
 9         Boolean requestingClose;
10         Boolean requestingStatus;
11         Boolean isPassing;
12 
13      Behavior:
14         if (requestingClose == True)
15             set(radioConnection, {closeRequest=True});
16 
17         if (requestingStatus == True)
18             set(radioConnection, {statusRequest=True});
19 
20         if (isPassing == True)
21             set(isPassingSignal, {value=True});
22 
23     Transitions:
24         when {delay=4, name=StartingApproach} (position == Far) [speed !=
               Stopped]
25             position = Approaching;
26         // ...
27         when {delay=1} (position == NearEmergencyBrakingZone)
28             requestingStatus = True;
29 
30         when {delay=1} (requestingStatus == True)
31             requestingStatus = False;
32         // ...
33         when {delay=2} (position == NearEmergencyBrakingZone)
34                               [!radioConnection.exists(releaseMsg=True)]
35             speed = SlowingDown;
36         // ...
37         when {delay=3} (speed == SlowingDown)
38             speed = Stopped;
39         // ...
40 }
```

Especially interesting is the transition which initiates emergency braking if the train has not received a release message (lines 33-35). The transition is already triggered when the position `NearEmergencyBrakingZone` is reached. As soon as the release message arrives, the guard condition is no longer fulfilled and therefore the transition is canceled. Otherwise (no release message arrived) the transition is executed.

The radio network is realized by a quite simple component with two ports that forwards messages. In case of an error, the communication is interrupted. The failure or reactivation of the network is realized using external events. To limit the state space, besides the feature `type=failure` to limit the number of independent failure events on each relevant path, the stable condition `stable=2` is added. This limits the failure activation to states that are stable to an order of magnitude value of two or higher. In addition, the guard condition must also be fulfilled. The listing below shows a possible implementation of the radio network in the smartIflow language.

```
class RadioNetwork {
    Ports:
        Bus p1;
        Bus p2;

    Variables:
        Enum[Ok, Disconnected] fm;

    Events:
        ActivateDisconnected{type=failure, p=1e-3};
        ActivateConnected{type=repair, p=1};

    Behavior:
        if (fm == Ok)
            connect(p1, p2);

    EventHandlers:
       when {stable=2} (ActivateDisconnected) [fm == Ok]
            fm = Disconnected;

        when {stable=2} (ActivateConnected) [fm == Disconnected]
            fm = Ok;
}
```

Class `Crossing` and `PosSensor` are implemented similarly.

6.3.2 Verification Results

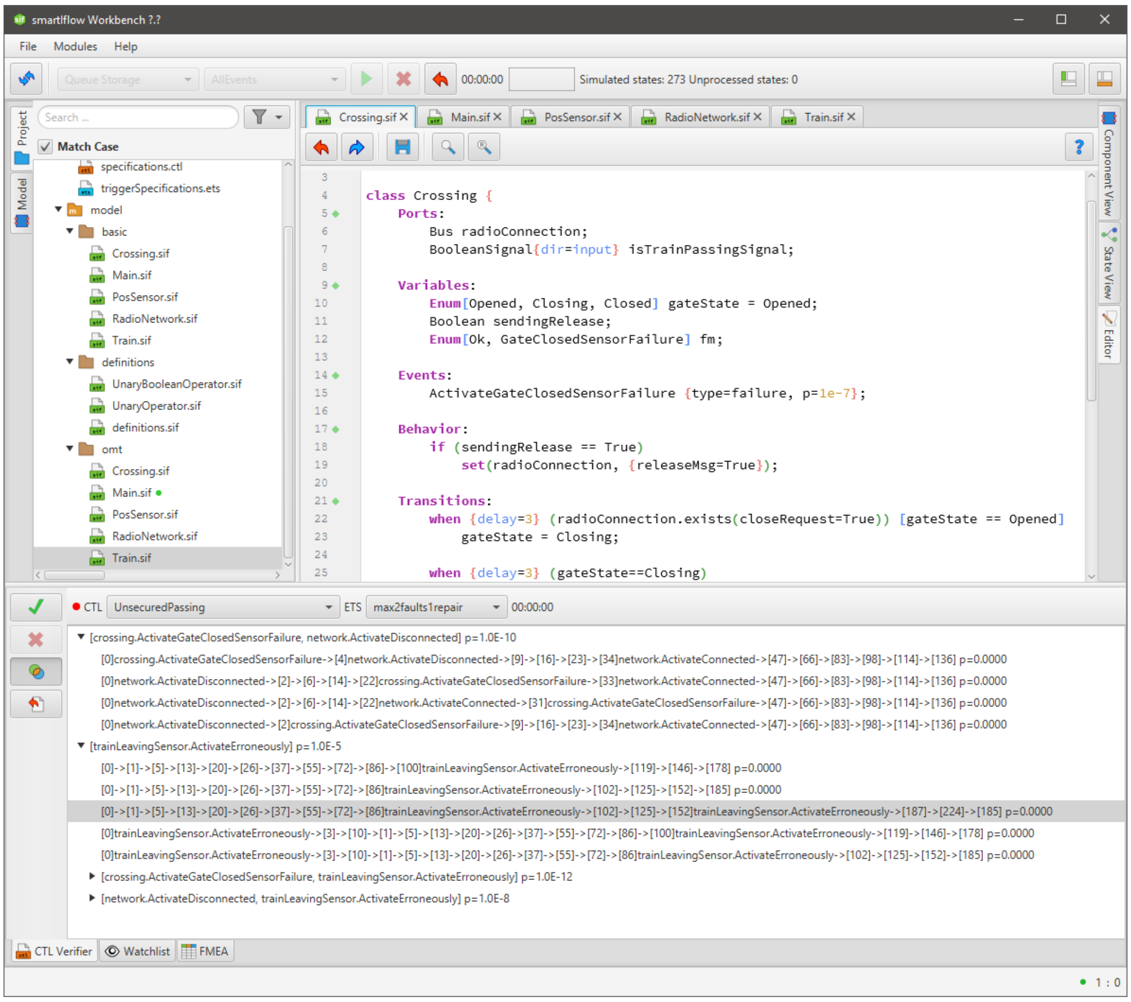
Unfolding the system model takes 55 milliseconds and results in a transition system with 270 different reachable states[7]. The requirement “Whenever the train passes the crossing, the gates must be closed” can be expressed in OMT-CTL as follows:

```
UnsecuredPassing :=
     AG(train.position == InDangerZone -> crossing.gateState == Closed);
```

Verification of the requirement takes 11 milliseconds and delivers 69 different counterexamples (see Figure 6.8). The good news is, that a temporary loss of radio connection cannot lead to an unsecured train passing. Most of the counterexamples result from an

[7]An ETS that allows a maximum of two failure events and one repair event on a path was applied.

erroneous activation of the train leaving sensor (event `ActivateErroneously` in class `PosSensor`). There is a reasonable explanation for this: The train sends a close request which causes the crossing to close the gates. The train received the release message and continued moving. However, shortly afterward, the train leaving sensor is activated erroneously which causes the gates to open. Now the train is passing the crossing even though the gates are open. The remaining counterexamples consist of a specific combination of temporary loss of network and a gate sensor failure: Due to the temporary loss of the radio connection, the close request will never reach the crossing and therefore the gates remain open. Soon after that, the radio connection is available again. The train sends the status request to the crossing which is answered incorrectly due to a sensor failure. Therefore, gates remain open and the train passes the crossing.



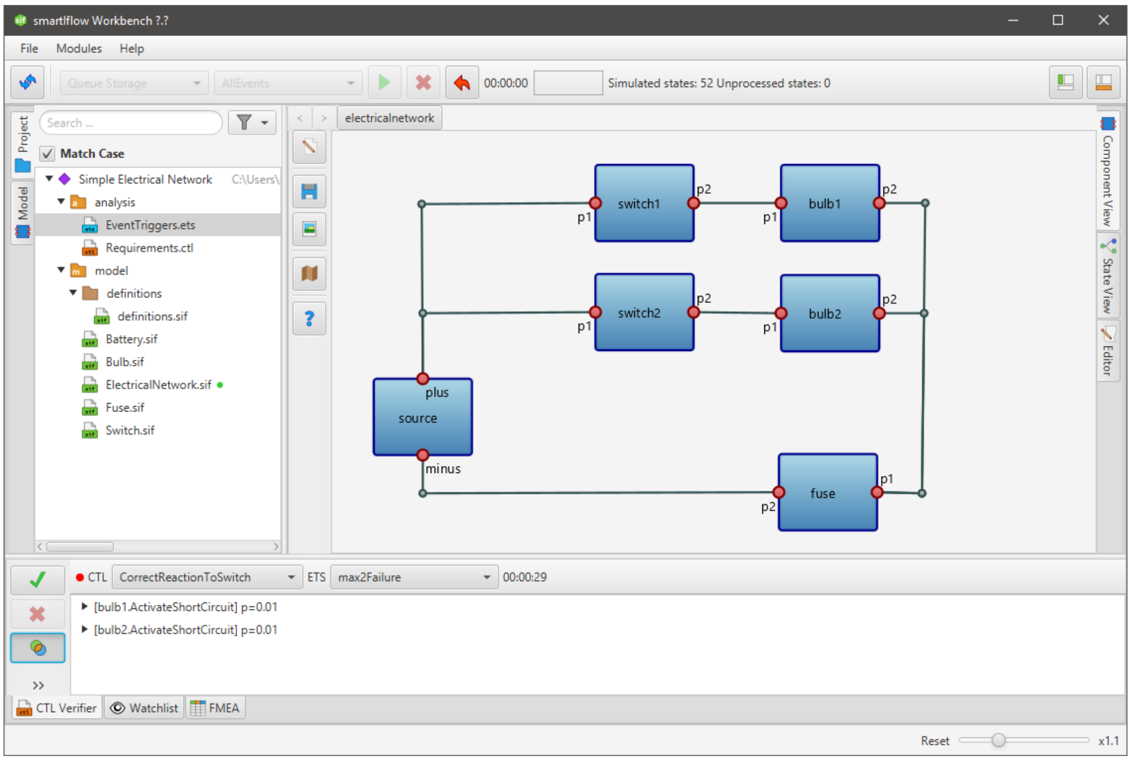
```
class Crossing {
    Ports:
        Bus radioConnection;
        BooleanSignal{dir=input} isTrainPassingSignal;

    Variables:
        Enum[Opened, Closing, Closed] gateState = Opened;
        Boolean sendingRelease;
        Enum[Ok, GateClosedSensorFailure] fm;

    Events:
        ActivateGateClosedSensorFailure {type=failure, p=1e-7};

    Behavior:
        if (sendingRelease == True)
            set(radioConnection, {releaseMsg=True});

    Transitions:
        when {delay=3} (radioConnection.exists(closeRequest=True)) [gateState == Opened]
            gateState = Closing;

        when {delay=3} (gateState==Closing)
```

```
▼ [crossing.ActivateGateClosedSensorFailure, network.ActivateDisconnected] p=1.0E-10
    [0]crossing.ActivateGateClosedSensorFailure->[4]network.ActivateDisconnected->[9]->[16]->[23]->[34]network.ActivateConnected->[47]->[66]->[83]->[98]->[114]->[136] p=0.0000
    [0]network.ActivateDisconnected->[2]->[6]->[14]->[22]crossing.ActivateGateClosedSensorFailure->[33]network.ActivateConnected->[47]->[66]->[83]->[98]->[114]->[136] p=0.0000
    [0]network.ActivateDisconnected->[2]->[6]->[14]->[22]network.ActivateConnected->[31]crossing.ActivateGateClosedSensorFailure->[47]->[66]->[83]->[98]->[114]->[136] p=0.0000
    [0]network.ActivateDisconnected->[2]crossing.ActivateGateClosedSensorFailure->[9]->[16]->[23]->[34]network.ActivateConnected->[47]->[66]->[83]->[98]->[114]->[136] p=0.0000
▼ [trainLeavingSensor.ActivateErroneously] p=1.0E-5
    [0]->[1]->[5]->[13]->[20]->[26]->[37]->[55]->[72]->[86]->[100]trainLeavingSensor.ActivateErroneously->[119]->[146]->[178] p=0.0000
    [0]->[1]->[5]->[13]->[20]->[26]->[37]->[55]->[72]->[86]trainLeavingSensor.ActivateErroneously->[102]->[125]->[152]->[185] p=0.0000
    [0]->[1]->[5]->[13]->[20]->[26]->[37]->[55]->[72]->[86]trainLeavingSensor.ActivateErroneously->[102]->[125]->[152]trainLeavingSensor.ActivateErroneously->[187]->[224]->[185] p=0.0000
    [0]trainLeavingSensor.ActivateErroneously->[3]->[10]->[1]->[5]->[13]->[20]->[26]->[37]->[55]->[72]->[86]->[100]trainLeavingSensor.ActivateErroneously->[119]->[146]->[178] p=0.0000
    [0]trainLeavingSensor.ActivateErroneously->[3]->[10]->[1]->[5]->[13]->[20]->[26]->[37]->[55]->[72]->[86]trainLeavingSensor.ActivateErroneously->[102]->[125]->[152]->[185] p=0.0000
  ▶ [crossing.ActivateGateClosedSensorFailure, trainLeavingSensor.ActivateErroneously] p=1.0E-12
  ▶ [network.ActivateDisconnected, trainLeavingSensor.ActivateErroneously] p=1.0E-8
```

Figure 6.8: Verification result with cut sets and counterexamples of the radio-controlled railroad crossing system

6.4 Simple Electrical System

The next example focuses on the built-in feature for qualitative energy flow analysis. Imagine a simple electrical system with a battery and two (conventional) light bulbs that can be controlled individually with a switch. A fuse provides overcurrent protection by disconnecting the electrical circuit if the electric current exceeds a specified level. Figure 6.10 visualizes the component structure in smartIflow Workbench.



Simulated states: 52 Unprocessed states: 0

```
[bulb1.ActivateShortCircuit] p=0.01
[bulb2.ActivateShortCircuit] p=0.01
```

Figure 6.9: Component view of a simple electrical system in smartIflow Workbench

Besides `Switch`, `Bulb` and `Fuse` a component class `Battery` is necessary. Essentially, the definition of the top-level class `ElectricalNetwork` looks as follows:

```
class ElectricalNetwork {
    Components:
        Battery source;
        Switch switch1;
        Switch switch2;
        Fuse fuse;
        Bulb bulb1;
        Bulb bulb2;

    Behavior:
        connect{r=1}(switch1.p1, source.plus);
        connect{r=1}(switch2.p1, source.plus);
        connect{r=1}(switch1.p2, bulb1.p1);
        connect{r=1}(switch2.p2, bulb2.p1);
        connect{r=1}(bulb1.p2, fuse.p1);
        connect{r=1}(bulb2.p2, fuse.p1);
        connect{r=1}(fuse.p2, source.minus);
}
```

In this class, components are instantiated and interconnected. Since the model is based on energy flows, each connection must be assigned an order of magnitude value for the resistance. The value `r=0` defines the reference resistance value (resistance of zero, reasonable for superconductors). The resistance value `r=1` (one order of magnitude above the reference resistance) is used for the connections between the components (electrical cables), switches in the closed state, and the fuse. The resistance of a light bulb is higher than the resistance of a cable. Therefore value `r=2` is used (two orders of magnitude above the reference resistance) for the light bulb. For simplicity, we assume that only one error can occur in the system: If water gets into the bulb (e.g., due to mechanical damage), the bulb will be destroyed and a short circuit will occur. The short circuit of the bulb is modeled with the resistance value `r=1`.

The operational mode of the bulb is changed by transition statements in which the flow between ports `p1` and `p2` is considered. If there is any flow in any direction, the operational mode is changed to `On`[8]. In the opposite case (no flow), the state of the bulb is set to `Off`. For the activation of the mechanical damage and intruding water, the external event `ActivateMechanicalDamage` is used. The complete definition of class `Bulb` is shown below:

```
import definitions.*;
class Bulb {
    Ports:
        Electrical p1;
        Electrical p2;

    Variables:
        Enum[Ok, Short] fm = Ok;
        Enum[Off, On] opm = Off;

    Events:
        ActivateMechanicalDamage{type=failure, p=1e-2};

    Behavior:
        if(fm == Ok)
            connect{r=2}(p1,p2);

        if(fm == Short)
            connect{r=1}(p1,p2);

    Transitions:
        when{name=bulbOn}(flow.om(p1,p2) != null)[fm == Ok && opm == Off]
            opm = On;

```

[8]The fact, that with low currents the bulb does not light up is neglected.

```
        when{name=bulbOff}(flow.om(p1,p2) == null)[fm == Ok && opm == On]
            opm = Off;

    EventHandlers:
        when(ActivateMechanicalDamage)[fm == Ok] {
            fm = Short;
        }

}
```

The switch is characterized by variable `opm` for the state (`Open`, `Closed`) and two ports (`p1`, `p2`). A connection between `p1` and `p2` is only established in state `Closed`. Switch operations (open, close) can be realized with external events. The fuse disconnects the electrical circuit when the current exceeds a specified level. This can be modeled with the following transition statement:

```
when{name=fuseBreakes}(flow.om(p1,p2) >= 1)[fm == Ok]
    fm = Broken;
```

If the flow between `p1` and `p2` is higher than one order of magnitude above the reference value, the state is set to `Broken`. For the battery, a bipolar effort source is used whose voltage level is two orders of magnitude above the reference value.

```
class Battery {flowComponent=bipolarEffortSource, om=2} {
    Ports:
        Electrical{flowPort=source} plus;
        Electrical{flowPort=drain} minus;
}
```

By closing a switch, the circuit is closed and thus there is a flow through the bulb. The qualitative energy flow analysis algorithm assigns the resulting effort and values to each component in the system.

In the case of this simple system, the effort and flow values can be easily calculated manually. Assume that `switch1` is closed and all components are working properly. First, the equivalent resistor must be calculated. For serial resistors, the maximum value is used. Since the resistance of `bulb1` dominates the other values, the equivalent resistor has value `r=2`. The flow is calculated by the formula $F = E/R$ (noting that the OM values are subtracted). Since resistance and effort have the same order of magnitude, the flow corresponds to the reference value.

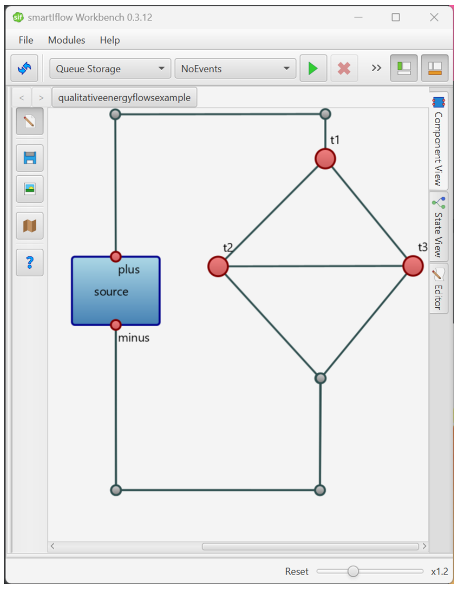
In the case of a short circuit, the flow is two orders of magnitude above the reference value. As a result, the state of `fuse` changes to `Broken` and thus the circuit disconnected. This behavior is also reflected in the results of the verification. The requirement "After pressing the switch, the associated lamp must light up." can be specified in OMT-CTL as follows:

```
Switch1Reaction          := AG(switch1.opm==Closed -> AF(bulb1.opm==On));
Switch2Reaction          := AG(switch2.opm==Closed -> AF(bulb2.opm==On));
CorrectReactionToSwitch  := Switch1Reaction && Switch2Reaction;
```

The model checker delivers a set of counterexamples, each describing a situation in which a short circuit leads to disconnection and therefore no more energy can flow.

Of course, this system can also be implemented without the flow calculation. Instead, properties can be used that indicate the presence of flow or overcurrent. However, this implementation quickly reaches its limits, when changes in the component structure result in new resistance values that affect voltage and current values in the system. Properties cannot be used to reflect such situations.

The system is based on a serial/parallel reducible circuit so that all flow and effort values can be calculated unambiguously. Problems may arise if the circuits are not serial/parallel reducible. Consider the following smartIflow system model:



```
import definitions.*;

class QualitativeEnergyFlowsExample {
    Components:
        // bipolarEffortSource, om=0
        EffortSource<Electrical> source;


    Ports:
        Electrical t1;
        Electrical t2;
        Electrical t3;

    Behavior:

        connect{r=1}(t1, t2);
        connect{r=1}(t1, t3);
        connect{r=3}(t2, t3);
        connect{r=1}(source.plus, t1);
        connect{r=1}(t2, source.minus);
        connect{r=2}(t3, source.minus);
}
```

Figure 6.10: Example of a nonserial/parallel reducible circuit (the structure of the system is taken from [SL14])

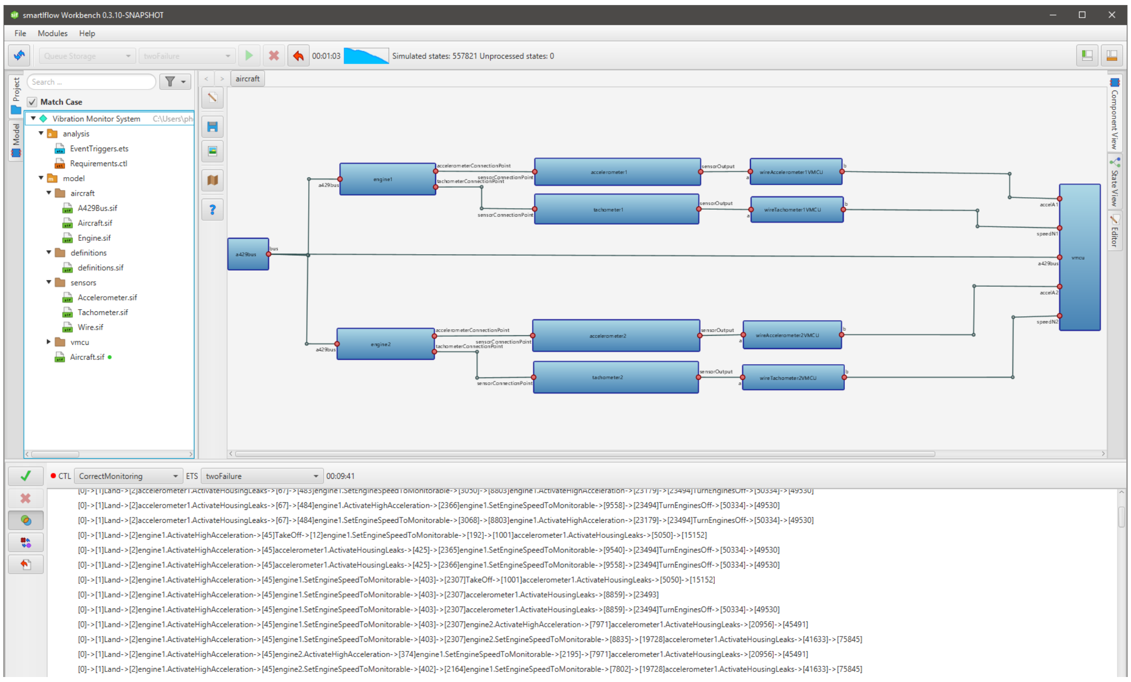
Obviously, the circuit is neither serial nor parallel reducible (because of the bridge from port `t2` to port `t3`). Port `t2` can be eliminated by a star-delta transformation. This transformation also eliminates the bridge from port `t2` to port `t3` and the resulting circuit becomes series/parallel reducible. Subsequently, all flow and effort values can be calculated unambiguously.

However, if the network reduction starts with a star-delta transformation at port `t1`, the situation is quite different. After the elimination of port `t1` (by means of a star-delta transformation), the bridge between port `t2` and port `t3` still exists. Therefore, another star-delta transformation for example at port `t2` is necessary. After the reduction of the circuit to a single equivalent resistor, the flow and effort values can be assigned successively. Ambiguities arise in the calculation of the flow between port `t1` and port `t2`. The flow between port `t1` and port `t2` is given by the sum of the flows of the relevant connections from the previous step. This results in a sum of two opposing flows of the same magnitude, which is obviously not possible.

Fortunately, an ambiguous flow value does not necessarily lead to problems at verification tasks if the behavior of higher-level components does not depend on the value and the model checker does not require it.

6.5 A Real World Application from the Aviation Industry

While the systems shown so far were still very manageable, the following example shows, that even large systems from industry can be analyzed with smartIflow. Figure 6.11 shows the architecture of a vibration monitor system as described for example in [Fv17]. The purpose of such a system is to monitor the vibrations in aircraft engines. The information about the vibrations helps, for instance, to detect material fatigue (e.g., cracks) on the turbine wheels at an early stage (defective turbine wheels will lead to vibrations). The system consists of a set of (redundant) sensors connected via a bus to a controller that processes the data. To evaluate the sensor data, the controller needs additional information about the state of the aircraft (e.g., whether the aircraft is on the ground or in the air).



00:01:03 Simulated states: 557821 Unprocessed states: 0

CTL CorrectMonitoring ETS twoFailure 00:09:41

```
[0]->[1]Land->[2]accelerometer1.ActivateHousingLeaks->[67]->[483]engine1.SetEngineSpeedToMonitorable->[3050]->[8803]engine1.ActivateHighAcceleration->[23179]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]accelerometer1.ActivateHousingLeaks->[67]->[484]engine1.ActivateHighAcceleration->[2366]engine1.SetEngineSpeedToMonitorable->[9558]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]accelerometer1.ActivateHousingLeaks->[67]->[484]engine1.SetEngineSpeedToMonitorable->[3068]->[8803]engine1.ActivateHighAcceleration->[23179]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]TakeOff->[12]engine1.SetEngineSpeedToMonitorable->[192]->[1001]accelerometer1.ActivateHousingLeaks->[5050]->[15152]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]accelerometer1.ActivateHousingLeaks->[425]->[2365]engine1.SetEngineSpeedToMonitorable->[9540]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]accelerometer1.ActivateHousingLeaks->[425]->[2366]engine1.SetEngineSpeedToMonitorable->[9558]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine1.SetEngineSpeedToMonitorable->[403]->[2307]TakeOff->[1001]accelerometer1.ActivateHousingLeaks->[5050]->[15152]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine1.SetEngineSpeedToMonitorable->[403]->[2307]accelerometer1.ActivateHousingLeaks->[8859]->[23493]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine1.SetEngineSpeedToMonitorable->[403]->[2307]accelerometer1.ActivateHousingLeaks->[8859]->[23494]TurnEnginesOff->[50334]->[49530]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine1.SetEngineSpeedToMonitorable->[403]->[2307]engine2.ActivateHighAcceleration->[7971]accelerometer1.ActivateHousingLeaks->[20956]->[45491]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine1.SetEngineSpeedToMonitorable->[403]->[2307]engine2.SetEngineSpeedToMonitorable->[8835]->[19728]accelerometer1.ActivateHousingLeaks->[41633]->[75845]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine2.ActivateHighAcceleration->[374]engine1.SetEngineSpeedToMonitorable->[2195]->[7971]accelerometer1.ActivateHousingLeaks->[20956]->[45491]
[0]->[1]Land->[2]engine1.ActivateHighAcceleration->[45]engine2.SetEngineSpeedToMonitorable->[402]->[2164]engine1.SetEngineSpeedToMonitorable->[7802]->[19728]accelerometer1.ActivateHousingLeaks->[41633]->[75845]
```

Figure 6.11: Component view of a vibration monitor system in smartIflow Workbench

Each sensor can fail in several ways which are represented in the model as external events. The model contains a total of 20 external events. Due to the combinatorial complexity, the state space can become enormously large depending on the applied ETS. Even with a limitation to a maximum of two failure events on a path, the unfolding results in a state space with 557821 different states (unfolding took about one minute).

One important property of the vibration monitor system is that possible vibrations are properly captured (e.g., vibrations must not remain undetected). The verification of this requirement (formulated in OMT-CTL) takes over 9 minutes and results in 133986 counterexamples.

This example shows that even a large system can be analyzed in smartIflow. But it also shows the importance of the Event Trigger Specifications that help to keep the search space manageable. Without suitable ETS, this system would hardly be manageable.

7 Summary, Conclusion & Outlook

7.1 Summary

smartIflow is a new approach to model-based safety analysis that attempts to overcome the limitations of existing tools by combining mechanisms from existing approaches and adding new techniques (Contribution C1). Models are created in the smartIflow language on a very high level of abstraction and quite close to the specifications. The smartIflow language is component- and object-oriented and strongly inspired by the programming language Java. This makes the modeling language very beginner-friendly. In addition, system models can also be composed graphically in Simulink/Simscape (Contribution C6). smartIflow components are stored in a custom block library where each block is assigned its raw smartIflow class. Models can be created as usual by adding blocks and creating connections between them. An integrated converter translates the block diagrams to the smartIflow language. For existing Simulink/Simscape models, at least the component structure can be converted to the smartIflow language.

In smartIflow, component behavior is described by means of finite state machines. The finite state machines can include states for nominal behavior as well as fault states. The communication between the components is realized by means of typed ports and connections. In contrast to other approaches, connections between components are in general bidirectional. Furthermore, connections can change depending on the state. Thus, the complete topology of systems can change depending on the current state. For the interaction between components, two mechanisms are supported: Qualitative energy flow analysis for reasoning about physical quantities and message exchange based on properties. Properties are simple messages that enable communication between components in a very flexible way. Components can publish properties and also react to properties sent by other components. Properties published by components are automatically propagated to all other reachable ports. The other mechanism for interaction between components relies on effort and flow calculation based on network analysis. Connections are assigned order of magnitude resistance values, and for sinks and sources there exist special built-in components. The integrated solver works in two steps. First, the connection structure is reduced to a single connection with a single equivalent resistance. After that, the connection structure is incrementally expanded and with each extension flow and effort values are assigned to the connections. With this feature, physical quantities in networks can be modeled realistically and context-free. Components change their variables based on internal or external events. The reaction to these events is specified using transition statements. Since the components react to events at quite different speeds, an order of magnitude value for the delay can be specified for each transition statement. Thus, delayed reactions can be modeled very easily.

By unfolding the smartIflow system model an explicit representation of the transition system is created. The unfolding consists of three main steps, namely the processing of the behavior, the qualitative energy flow analysis, and the processing of the internal and external events. Event Trigger Specifications (ETS) define the maximum number of external events on a path. These specifications can be applied during unfolding to keep the search space manageable (Contribution C3). The transition system can be verified by a special model checker. In contrast to most other implementations, the model checker delivers all relevant counterexamples which are paths in which the specification is not fulfilled (Contribution C4). Requirements are specified in Orders of Magnitude timed CTL (OMT-CTL), an extended version of the Computation Tree Logic (CTL). OMT-CTL is extended by predicates to express timed properties for smartIflow system models (Contribution C5). The comprehensive counterexamples provide the base to create minimal cut-sets and Fault Trees.

To get a clear understanding of the new modeling formalism, the semantics of smartIflow system model, the underlying transition systems, and the new timed variant of CTL are formally defined (Contribution C2).

7.2 Conclusion & Outlook

In conclusion, safety engineers can benefit a lot from the smartIflow approach when analyzing the safety of technical systems in the early design phases. One benefit is the level of abstraction on which the systems are described. Systems are described on an as-high-as-possible level of abstraction without losing too much prediction power. In smartIflow, the mechanisms for modeling state transitions are more extensive compared to other approaches. For example, the differentiation between internal and external events leads to a smaller state space, since internal transitions are executed synchronously. The extremely flexible message exchange between the components by means of properties has proven to be quite practicable. Contrary to our expectations, communication could be realized in almost all our experiments using the property concept. Nevertheless, there are scenarios where the property concept reaches its limit and the qualitative energy flow analysis can show its strength. The energy flow analysis works quite well for most circuits, but in some scenarios, multiple executions of the star-delta transformation can lead to ambiguities in the calculation of flow values. Possibly these ambiguities can be solved by clever selection of the star account at which the star-delta transformation is performed (if several nodes for this transformation are possible). This could be a potential extension for the energy flow analysis. The smartIflow language is a quite intuitive modeling language. Our experiments have shown that modeling in smartIflow Workbench is much more efficient than the graphical modeling approach in Simulink. The reason for this is that whenever the system model is modified, the model must first be converted to the smartIflow language and afterward the analyses can be performed in the smartIflow Workbench. With a little practice, textual modeling in the smartIflow Workbench performs very well. Nevertheless, the graphical modeling approach in Simulink is a helpful tool to convert the structure of existing Simulink/Simscape into the smartIflow language.

As important as ETS are, they also come with some pitfalls:

- If event counters are updated during unfolding by referencing an existing state, in the worst case all subsequent paths must be checked to see whether additional events need to be triggered.

- The verification must follow cycles when events occur in the cycle that are part of the ETS.

Nevertheless, Event Trigger Specifications are an extremely important instrument to keep the state space manageable but also to limit the analysis to the relevant paths. Possible extensions for the Event Trigger Specifications are:

- A limitation of paths containing all event combinations up to a certain probability of occurrence.
- The definition of scenarios to analyze the effect of a sequence of events ($event_1 \rightarrow event_2 \rightarrow \ldots \rightarrow event_n$)
- The introduction of more complex rules, e.g., to define that an event of type A is always followed by an event of type B.

The explicit-state model checking algorithm is quite special, but unlike existing tools, it uses Event Trigger Specifications to focus on relevant paths. The model-checking-based approach to FTA is unique and has a high potential. At first glance, the transition system resulting from unfolding often looks manageable, also for the model checker. However, the transition system may contain an extremely large number of paths, which leads to performance issues in model checking (depending on the specification, in the worst case all paths have to be analyzed). In future work, it could be analyzed whether a model checker based on satisfaction sets can handle larger state spaces more efficiently.

A Grammar

This appendix contains the grammar definition in Extended Backus-Naur Form (EBNF) of the smartIflow language, Orders of Magnitude timed CTL and Event Trigger Specifications. The common lexer rules are *<Identifer>* for identifiers and *<Number>* for any number (integer or double). Parser rules start in lowercase, lexer rules in uppercase. The grammar is used for the ANTLR parser generator.

A.1 smartIflow Language

For the sake of clarity, the grammar of logical formulas for if- and when-expressions as well as features and properties are described separately.

```
⟨compilationUnit⟩       ::= ⟨packageDeclaration⟩? ⟨importDeclaration⟩*
                             (⟨classDeclaration⟩
                           | ⟨enumTypeDefinition⟩
                           | ⟨portTypeDefinition⟩)*

⟨packageDeclaration⟩    ::= 'package' ⟨qualifiedName⟩ ';'

⟨importDeclaration⟩     ::= 'import' ⟨qualifiedName⟩ ('.' '*')? ';'

⟨classDeclaration⟩      ::= 'class' ⟨Identifier⟩ ⟨typeVariables⟩? ⟨superclass⟩?
                             ⟨classFeatures⟩? '{' ⟨memberDeclaration⟩ '}'

⟨classFeatures⟩         ::= ⟨featureList⟩

⟨typeVariables⟩         ::= '<' ⟨Identifier⟩ (',' ⟨Identifier⟩ )* '>'

⟨superclass⟩            ::= 'extends' ⟨qualifiedName⟩ ⟨typeParameters⟩?

⟨typeParameters⟩        ::= '<' ⟨qualifiedName⟩ (',' ⟨qualifiedName⟩ )* '>'

⟨memberDeclaration⟩     ::= ⟨componentsSection⟩?
                             ⟨portsSection⟩?
                             ⟨variablesSection⟩?
                             ⟨eventsSection⟩?
                             ⟨behaviorsSection⟩?
                             ⟨transitionsSection⟩?
                             ⟨eventHandlersSection⟩?

⟨componentsSection⟩     ::= 'Components:' ⟨componentInstance⟩*
```

```
⟨componentInstance⟩        ::= ⟨qualifiedName⟩ ⟨typeParameters⟩? ⟨Identifier⟩ ‘;’

⟨portsSection⟩             ::= ‘Ports:’ ⟨portInstance⟩*

⟨portInstance⟩             ::= ⟨qualifiedName⟩ ⟨portInstanceFeatures⟩? ⟨Identifier⟩ ‘;’

⟨portInstanceFeatures⟩     ::= ⟨featureList⟩

⟨variablesSection⟩         ::= ‘Variables:’
                               (⟨anonymousTypeInstance⟩ | ⟨variableInstance⟩)*

⟨anonymousTypeInstance⟩ ::= ‘Enum’ ‘[’ ⟨potentialValues⟩ ‘]’ ⟨Identifier⟩
                               ⟨featureList⟩? (‘=’ ⟨Identifier⟩)? ‘;’

⟨potentialValues⟩          ::= (⟨Identifier⟩ (‘,’ ⟨Identifier⟩)*)*

⟨variableInstance⟩         ::= ⟨qualifiedName⟩ ⟨Identifier⟩ ⟨featureList⟩?
                               (‘=’ ⟨Identifier⟩)? ‘;’

⟨eventsSection⟩            ::= ‘Events:’ (⟨Identifier⟩ ⟨featureList⟩? ‘;’)*

⟨behaviorsSection⟩         ::= ‘Behavior:’ (⟨ifStatement⟩ | ⟨action⟩)*

⟨ifStatement⟩              ::= ‘if’ ‘(’ ⟨ifLogicalExpression⟩ ‘)’
                               ⟨ifStatement⟩
                             | ⟨action⟩
                             | ⟨statementList⟩
                               ⟨elseIfStatement⟩? ⟨elseStatement⟩?

⟨elseIfStatement⟩          ::= ‘else’ ⟨ifStatement⟩

⟨elseStatement⟩            ::= ‘else’
                             | ⟨action⟩
                             | ⟨statementList⟩

⟨statementList⟩            ::= ‘{’ (⟨ifStatement⟩ | ⟨action⟩)* ‘}’

⟨action⟩                   ::= ⟨setAction⟩ | ⟨connectAction⟩

⟨setAction⟩                ::= ‘set’ ‘(’ ⟨qualifiedName⟩ ‘,’ ⟨propertyList⟩ ‘)’ ‘;’

⟨connectionAction⟩         ::= ‘connect’ ⟨featureList⟩? ‘(’ ⟨qualifiedName⟩ ‘,’
                               ⟨qualifiedName⟩ ‘,’ ⟨propertyList⟩? ‘);’

⟨transitionsSection⟩       ::= ‘Transitions:’ ⟨condition⟩* ⟨conditionalStateTransition⟩*

⟨condition⟩                ::= ⟨Identifier⟩ ‘=’ ⟨whenLogicalExpression⟩ ‘;’
```

```
⟨conditionalStateTransition⟩ ::= ‘when’ ⟨featureList⟩? ‘(’⟨whenLogicalExpression⟩‘)’
                                 ⟨guardExpression⟩?
                                 ⟨stateTransitionList⟩
                               | ⟨stateTransition⟩

⟨eventHandlersSection⟩ ::= ‘EventHandlers:’ ⟨eventHandler⟩*

⟨eventHandler⟩         ::= ‘when’ ⟨featureList⟩? ‘(’⟨Identifier⟩‘)’ ⟨guardExpression⟩?
                           ⟨stateTransitionList⟩
                         | ⟨stateTransition⟩

⟨stateTransitionList⟩  ::= ‘{’ ⟨stateTransition⟩* ‘}’

⟨stateTransition⟩      ::= ⟨Identifier⟩ ‘=’ ⟨Identifier⟩ (‘or’ ⟨Identifier⟩) ‘;’

⟨guardExpression⟩      ::= ‘[’ ⟨whenLogicalExpression⟩ ‘]’
```

Qualified Names, Properties and Features

```
⟨qualifiedName⟩        ::= ⟨Identifier⟩ (‘.’ ⟨Identifier⟩)*

⟨propertyList⟩         ::= ‘{’ ⟨property⟩ (‘,’ ⟨property⟩)* ‘}’

⟨property⟩             ::= ⟨qualifiedName⟩ ‘=’ ⟨propertyValue⟩

⟨propertyValue⟩        ::= ⟨Identifier⟩ | ⟨Number⟩

⟨featureList⟩          ::= ‘{’ ⟨feature⟩ (‘,’ ⟨feature⟩)* ‘}’

⟨feature⟩              ::= ⟨Identifier⟩ ‘=’ (⟨Number⟩ | ⟨Identifier⟩ | ⟨stringLiteral⟩)
```

Logical Expressions for if-Statements

```
⟨ifLogicalExpression⟩  ::= ⟨ifConjunction⟩ (‘||’ ⟨ifConjunction⟩)*

⟨ifConjunction⟩        ::= ⟨ifUnaryExpression⟩ (‘&&’ ⟨ifUnaryExpression⟩)*

⟨ifUnaryExpression⟩    ::= ⟨ifAtomExpression⟩ | (‘!’ ⟨ifAtomExpression⟩)

⟨ifAtomExpression⟩     ::= ⟨equalityExpression⟩ | ‘(’ ⟨ifLogicalExpression⟩ ‘)’

⟨equalityExpression⟩   ::= ⟨Identifier⟩ (‘==’ | ‘!=’) ⟨Identifier⟩
```

Logical Expressions for when-Statements

⟨*whenLogicalExpression*⟩ ::= ⟨*whenConjunction*⟩ (‘`||`’ ⟨*whenConjunction*⟩)*

⟨*whenConjunction*⟩ ::= ⟨*whenUnaryExpression*⟩ (‘`&&`’ ⟨*whenUnaryExpression*⟩)*

⟨*whenUnaryExpression*⟩ ::= ⟨*whenAtomExpression*⟩ | (‘`!`’ ⟨*whenAtomExpression*⟩)

⟨*whenAtomExpression*⟩ ::= ⟨*propertyExistsFunction*⟩
| ⟨*propertyKeyExistsFunction*⟩
| ⟨*equalityExpression*⟩
| ⟨*portEqualityExpression*⟩
| ⟨*flowOMRelExpression*⟩
| ⟨*flowDirRelExpression*⟩
| ⟨*effortOMRelExpression*⟩
| ⟨*effortDirRelExpression*⟩
| ⟨*conditionReference*⟩
| ‘`(`’ ⟨*whenLogicalExpression*⟩ ‘`)`’

⟨*propertyKeyExistsFunction*⟩ ::= ⟨*qualifiedName*⟩ ‘`.exists(`’ ⟨*qualifiedName*⟩ ‘`)`’

⟨*propertyExistsFunction*⟩ ::= ⟨*qualifiedName*⟩ ‘`.exists(`’ ⟨*qualifiedName*⟩ ‘`=`’
⟨*propertyValue*⟩ ‘`)`’

⟨*equalityExpression*⟩ ::= ⟨*Identifier*⟩ (‘`==`’ | ‘`!=`’) ⟨*Identifier*⟩

⟨*portEqualityExpression*⟩ ::= ⟨*portValueFunction*⟩ (‘`==`’ | ‘`!=`’) ⟨*portValueFunction*⟩

⟨*portValueFunction*⟩ ::= ⟨*qualifiedName*⟩ ‘`.value(`’ ⟨*qualifiedName*⟩ ‘`)`’

⟨*OmRelationalOperator*⟩ ::= ‘`==`’ | ‘`!=`’ | ‘`>`’ | ‘`>=`’ | ‘`<`’ | ‘`<=`’

⟨*flowOMRelExpression*⟩ ::= ‘`flow.om(`’ ⟨*qualifiedName*⟩ ‘`,`’ ⟨*qualifiedName*⟩ ‘`)`’
⟨*OmRelationalOperator*⟩ (‘`null`’ | ⟨*Number*⟩)

⟨*flowDirRelExpression*⟩ ::= ‘`flow.dir(`’ ⟨*qualifiedName*⟩ ‘`,`’ ⟨*qualifiedName*⟩ ‘`)`’
⟨*dirRelationalOperator*⟩ ⟨*Number*⟩

⟨*effortOMRelExpression*⟩ ::= ‘`effort.om(`’ ⟨*qualifiedName*⟩ ‘`,`’ ⟨*qualifiedName*⟩ ‘`)`’
⟨*OmRelationalOperator*⟩ (‘`null`’ | ⟨*Number*⟩)

⟨*effortDirRelExpression*⟩ ::= ‘`effort.dir(`’ ⟨*qualifiedName*⟩ ‘`,`’ ⟨*qualifiedName*⟩ ‘`)`’
⟨*dirRelationalOperator*⟩ ⟨*Number*⟩

⟨*conditionReference*⟩ ::= ⟨*Identifier*⟩

Enumeration Definitions

⟨*enumTypeDefinition*⟩ ::= ‘`enum`’ ⟨*Identifier*⟩ ⟨*superclass*⟩?
‘`{`’ ⟨*enumConstantList*⟩ ‘`}`’

⟨*enumConstantList*⟩ ::= ⟨*enumConstant*⟩ (‘`,`’ ⟨*enumConstant*⟩)*

⟨*enumConstant*⟩ ::= ⟨*Identifier*⟩ (‘`{`’ ⟨*featureList*⟩ ‘`}`’)?

Port Type Definitions

⟨*portTypeDefinition*⟩ ::= ‘`port`’ ⟨*Identifier*⟩ ⟨*superclass*⟩? ‘`{`’ ⟨*propertySection*⟩? ‘`}`’

⟨*propertySection*⟩ ::= ‘`Properties:`’ ⟨*propertyDeclaration*⟩*

⟨*propertyDeclaration*⟩ ::= ⟨*qualifiedName*⟩ ‘`=`’ ‘`{`’ ⟨*Identifier*⟩ (‘`,`’ ⟨*Identifier*⟩)* ‘`}`’ ‘`;`’

A.2 Orders of Magnitude timed CTL

⟨*ctlDefinitions*⟩ ::= (⟨*ctlDefinition*⟩)*

⟨*ctlDefinition*⟩ ::= ⟨*Identifier*⟩ ‘`:=`’ ⟨*biimplication*⟩ ‘`;`’

⟨*biimplication*⟩ ::= ⟨*implication*⟩ (‘`<->`’ ⟨*implication*⟩’)*

⟨*implication*⟩ ::= ⟨*disjunction*⟩ (‘`->`’ ⟨*disjunction*⟩’)*

⟨*disjunction*⟩ ::= ⟨*conjunction*⟩ (‘`||`’ ⟨*conjunction*⟩’)*

⟨*conjunction*⟩ ::= ⟨*expression*⟩ (‘`&&`’ ⟨*expression*⟩’)*

⟨*expression*⟩ ::= ⟨*unaryCTLOperator*⟩
| ⟨*binaryCTLOperator*⟩
| ⟨*unaryExpression*⟩

⟨*unaryCTLOperator*⟩ ::= (‘`AG`’ | ‘`AF`’ | ‘`EG`’ | ‘`EF`’ | ‘`AX`’ | ‘`EX`’) ⟨*pathTimeConstraint*⟩?
⟨*unaryExpression*⟩

⟨*binaryCTLOperator*⟩ ::= (‘`A`’ | ‘`E`’) ‘`(`’ ⟨*unaryExpression*⟩ ‘`U`’ ⟨*pathTimeConstraint*⟩?
⟨*unaryExpression*⟩ ‘`)`’

⟨*unaryExpression*⟩ ::= (⟨*atomExpression*⟩ | (‘`!`’ ⟨*atomExpression*⟩))
⟨*stateTimeConstraint*⟩?

⟨*atomExpression*⟩ ::= ‘`(`’ ⟨*biimplication*⟩ ‘`)`’
| ⟨*relationalExpression*⟩
| ‘`true`’ | ‘`false`’
| ⟨*Identifier*⟩

⟨*relationalExpression*⟩	::=	⟨*qualifiedName*⟩ (‘`==`’ | ‘`!=`’) (⟨*Identifier*⟩ | ⟨*Number*⟩)
⟨*stateTimeConstraint*⟩	::=	‘`{stable`’ (‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ | ‘`==`’ | ‘`!=`’) ⟨*Number*⟩‘`}`’
⟨*pathTimeConstraint*⟩	::=	‘`{delay`’ (‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ | ‘`==`’ | ‘`!=`’) ⟨*Number*⟩‘`}`’

A.3 Event Trigger Specifications

⟨*etsDefinitions*⟩	::=	⟨*etsDefinition*⟩*
⟨*etsDefinition*⟩	::=	⟨*Identifier*⟩ ‘`:=`’ ⟨*ets*⟩ ‘`;`’
⟨*ets*⟩	::=	⟨*disjunction*⟩ (‘`->`’ ⟨*disjunction*⟩’)*
⟨*disjunction*⟩	::=	⟨*conjunction*⟩ (‘`||`’ ⟨*conjunction*⟩’)*
⟨*conjunction*⟩	::=	⟨*equalityExpression*⟩ (‘`&&`’ ⟨*equalityExpression*⟩’)*
⟨*equalityExpression*⟩	::=	⟨*unaryExpression*⟩ | ⟨*relationalExpression*⟩ (‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ | ‘`==`’ | ‘`!=`’) ⟨*unaryExpression*⟩
⟨*unaryExpression*⟩	::=	⟨*primary*⟩ | ‘`!`’ ⟨*unaryExpression*⟩ | ⟨*Identifier*⟩
⟨*primary*⟩	::=	⟨*function*⟩ | ⟨*Number*⟩ | ‘`(`’ ⟨*ets*⟩ ‘`)`’ | ⟨*booleanFunction*⟩
⟨*function*⟩	::=	(‘`hist`’ | ‘`path`’) (‘`.`’ ⟨*filter*⟩)* ‘`.count()`’
⟨*filter*⟩	::=	‘`filter`’ ‘`(`’ ⟨*feature*⟩ ‘`)`’

List of Figures

List of Tables

Bibliography

[Ake78] S. B. Akers. "Binary Decision Diagrams." In: *IEEE Transactions on Computers* C-27.6 (June 1978), pp. 509–516. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675141.

[Alj+09] Husain Aljazzar, Manuel Fischer, Lars Grunske, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. "Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples." In: *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems.* QEST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 299–308. ISBN: 978-0-7695-3808-2. DOI: 10.1109/QEST.2009.8.

[Avi+04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[Bat+13] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul. "The AltaRica 3.0 project for model-based safety assessment." In: *2013 11th IEEE International Conference on Industrial Informatics (INDIN).* July 2013, pp. 741–746. DOI: 10.1109/INDIN.2013.6622976.

[Bay+14] Bernward Bayer, Axel Büse, Paul Linhoff, Bernd Piller, Peter E. Rieth, Stefan Schmitt, Bernhard Schmittner, Jürgen Völkel, and Chen Zhang. "Electro-Mechanical Brake Systems." In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort.* Ed. by Hermann Winner, Stephan Hakuli, Felix Lotz, and Christina Singer. Cham: Springer International Publishing, 2014, pp. 1–11. ISBN: 978-3-319-09840-1. DOI: 10.1007/978-3-319-09840-1_31-1.

[Bie+04] Pierre Bieber, Christian Bougnol, Charles Castel, Jean-Pierre Heckmann Christophe Kehren, Sylvain Metge, and Christel Seguin. "Safety Assessment with Altarica." In: *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France.* Ed. by Renè Jacquart. Boston, MA: Springer US, 2004, pp. 505–510. ISBN: 978-1-4020-8157-6. DOI: 10.1007/978-1-4020-8157-6_45.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[Boi+06] M. Boiteau, Y. Dutuit, A. Rauzy, and J.-P. Signoret. “The AltaRica data-flow language in use: modeling of production availability of a multi-state system.” In: *Reliability Engineering & System Safety* 91.7 (2006), pp. 747 –755. ISSN: 0951-8320. DOI: `http://dx.doi.org/10.1016/j.ress.2004.12.004`.

[BPR17] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. “AltaRica 3.0 assertions: The whys and wherefores.” In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 231 (Sept. 2017), p. 1748006X1772820. DOI: `10.1177/1748006X17728209`.

[Bry86] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation.” In: *IEEE Trans. Comput.* 35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340. DOI: `10.1109/TC.1986.1676819`.

[CE82] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic.” In: *Logics of Programs: Workshop, Yorktown Heights, New York, May 1981*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3. DOI: `10.1007/BFb0025774`.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[Cla+02] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. “Tree-Like Counterexamples in Model Checking.” In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 19–29. ISBN: 0-7695-1483-9.

[Cla08] Edmund M. Clarke. “The Birth of Model Checking.” In: *25 Years of Model Checking: History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–26. ISBN: 978-3-540-69850-0. DOI: `10.1007/978-3-540-69850-0_1`.

[Die00] Reinhard Diestel. *Graphentheorie*. BerlinHeidelbergNew YorkBarcelonaHongkongLondonMailandParisSingapurTokio: Springer, 2000. ISBN: 3540676562.

[Don+99] Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David S. Warren. “Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 74–88. ISBN: 978-3-540-49059-3.

[Ecu] *Number of automotive ECUs continues to rise*. `https://www.eenewsautomotive.com/news/number-automotive-ecus-continues-rise`. Accessed: 2020-05-06.

[EH86] E. Allen Emerson and Joseph Y. Halpern. “"Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic.” In: *J. ACM* 33.1 (Jan. 1986), pp. 151–178. ISSN: 0004-5411. DOI: `10.1145/4904.4999`.

[Eme+92] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. “Quantitative temporal reasoning.” In: *Real-Time Systems* 4.4 (Dec. 1992), pp. 331–352. ISSN: 1573-1383. DOI: `10.1007/BF00355298`.

[Fra+10] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. “Comparison of Model Checking Tools for Information Systems.” In: *Formal Methods and Software Engineering*. Ed. by Jin Song Dong and Huibiao Zhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 581–596. ISBN: 978-3-642-16901-4.

[Fv17] Stanislav Fabry and Marek Češkovič. “Aircraft gas turbine engine vibration diagnostics.” In: *MAD - Magazine of Aviation Development* 5 (Nov. 2017), p. 24. DOI: `10.14311/MAD.2017.04.04`.

[GA99] G.Point and A.Rauzy. “AltaRica – Constraint automata as a description language.” English. In: *Journal Européen des Systèmes Automatisés* 33.8–9 (1999), pp. 1033–1052. ISSN: 1269-6935.

[Gar+11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. “CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 372–387. ISBN: 978-3-642-19835-9.

[Gir+20] Gaëlle Girard, Ivan Baeriswyl, Jonathan Hendriks, Roland Scherwey, Christian Müller, Philipp Hönig, and Rüdiger Lunde. “Model based Safety Analysis using SysML with Automatic Generation of FTA and FMEA Artifacts.” In: Jan. 2020, pp. 5051–5058. DOI: `10.3850/978-981-14-8593-0_4941-cd`.

[GO10] Matthias Gudemann and Frank Ortmeier. “A Framework for Qualitative and Quantitative Formal Model-Based Safety Analysis.” In: *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering*. HASE ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 132–141. ISBN: 978-0-7695-4292-8. DOI: `10.1109/HASE.2010.24`.

[GO11] M. Güdemann and F. Ortmeier. “Towards model-driven safety analysis.” In: *2011 3rd International Workshop on Dependable Control of Discrete Systems*. June 2011, pp. 53–58. DOI: `10.1109/DCDS.2011.5970318`.

[HL14] Philipp Hönig and Rüdiger Lunde. “A New Modeling Approach for Automated Safety Analysis Based on Information Flows.” In: *25th International Workshop on Principles of Diagnosis (DX14)*. Graz, Austria, 2014.

[HLH15] Philipp Hönig, Rüdiger Lunde, and Florian Holzapfel. “Modeling Technical Systems with smartIflow for Safety Related Tasks.” In: *Proceedings of the International Workshop on Applications in Information Technology (IWAIT-2015)*. Ed. by E. Pyshkin and V. Klyuev. Aizu-Wakamatsu, Japan: The University of Aizu Press, 2015. ISBN: 978-4-900721-03-6.

[HLH16] Philipp Hönig, Rüdiger Lunde, and Florian Holzapfel. “Formal Verification of Technical Systems Using smartIflow and CTL.” In: *Proceedings of the 2nd International Conference on Applications in Information Technology (ICAIT-2016)*. Ed. by Evgeny Pyshkin, Vitaly Klyuev, and Alexander Vazhenin. Aizu-Wakamatsu, Japan: The University of Aizu Press, 2016. ISBN: 978-4-900721-04-3.

[HLH17] Philipp Hönig, Rüdiger Lunde, and Florian Holzapfel. “Model Based Safety Analysis with smartIflow.” In: *Information* 8.1 (2017). ISSN: 2078-2489. DOI: 10.3390/info8010007.

[Hol97] G.J. Holzmann. “The model checker SPIN.” In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[JH07] A. Joshi and M. P. E. Heimdahl. “Behavioral Fault Modeling for Model-based Safety Analysis.” In: *High Assurance Systems Engineering Symposium, 2007. HASE ’07. 10th IEEE.* Nov. 2007, pp. 199–208. DOI: 10.1109/HASE.2007.58.

[JK04] Shengbing Jiang and R. Kumar. “Failure diagnosis of discrete-event systems with linear-time temporal logic specifications.” In: *IEEE Transactions on Automatic Control* 49.6 (June 2004), pp. 934–945. ISSN: 0018-9286. DOI: 10.1109/TAC.2004.829616.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. “Design by Contract: The Lessons of Ariane.” In: *IEEE Computer* 30 (Jan. 1997), pp. 129–130.

[Jos+05] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl. “A proposal for model-based safety analysis.” In: *24th Digital Avionics Systems Conference*. Vol. 2. 2005, 13 pp. Vol. 2–.

[Jos+05] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl. “A proposal for model-based safety analysis.” In: *24th Digital Avionics Systems Conference*. Vol. 2. Oct. 2005, 13 pp. Vol. 2–. DOI: 10.1109/DASC.2005.1563469.

[JWH] Anjali Joshi, Mike Whalen, and Mats Heimdahl. “Model-Based Safety Analysis Final Report.” In: ().

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems.” In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591. ISBN: 978-3-642-22110-1.

[Koy90] Ron Koymans. “Specifying real-time properties with metric temporal logic.” In: *Real-Time Systems* 2.4 (Nov. 1990), pp. 255–299. ISSN: 1573-1383. DOI: 10.1007/BF01995674.

[Lee+85] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. “Fault Tree Analysis, Methods, and Applications - A Review.” In: *IEEE Transactions on Reliability* R-34.3 (Aug. 1985), pp. 194–203.

[Lev17] Nancy G. Leveson. “The Therac-25: 30 Years Later.” In: *Computer* 50.11 (2017), pp. 8–11. DOI: 10.1109/MC.2017.4041349.

[LHM18] Rüdiger Lunde, Philipp Hönig, and Christian Müller. "Reasoning about Different Orders of Magnitude of Time with smartIflow." In: *IFAC-PapersOnLine* 51.24 (2018). 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2018, pp. 131–138. ISSN: 2405-8963. DOI: `https://doi.org/10.1016/j.ifacol.2018.09.546`.

[Lip+15] Michael Lipaczewski, Frank Ortmeier, Tatiana Prosvirnova, Antoine Rauzy, and Simon Struck. "Comparison of Modeling Formalisms for Safety Analyses: SAML and AltaRica." In: *Reliability Engineering & System Safety* 140 (Aug. 2015). DOI: `10.1016/j.ress.2015.03.038`.

[LKN11] Oleg Lisagor, Tim Kelly, and Ru Niu. "Model-based safety assessment: Review of the discipline and its challenges." In: *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*. 2011, pp. 625–632. DOI: `10.1109/ICRMS.2011.5979344`.

[MMB09] Robin E. McDermott, Raymond J. Mikulak, and Michael R. Beauregard. *The basics of FMEA*. 2nd ed. New York: CRC Press/Productivity Press, 2009. ISBN: 9781563273773.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Berlin, Heidelberg: Springer-Verlag, 1992. ISBN: 0-387-97664-7.

[ORS05] Frank Ortmeier, Wolfgang Reif, and Gerhard Schellhorn. "Formal Safety Analysis of a Radio-Based Railroad Crossing Using Deductive Cause-Consequence Analysis (DCCA)." In: *Dependable Computing - EDCC 5*. Ed. by Mario Dal Cin, Mohamed Kaâniche, and András Pataricza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 210–224. ISBN: 978-3-540-32019-7.

[OW08] Joël Ouaknine and James Worrell. "Some Recent Results in Metric Temporal Logic." In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Franck Cassez and Claude Jard. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–13. ISBN: 978-3-540-85778-5.

[Pay60] Henry M. Paynter. *Analysis and Design of Engineering Systems: Class Notes for M.I.T. Course 2.751*. Cambridge, Massachusetts: M.I.T. Press, 1960.

[Pel98] Doron Peled. "Ten years of partial order reduction." In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 17–28. ISBN: 978-3-540-69339-0.

[PM01] Y. Papadopoulos and M. Maruhn. "Model-based synthesis of fault trees from Matlab-Simulink models." In: *2001 International Conference on Dependable Systems and Networks*. July 2001, pp. 77–82. DOI: `10.1109/DSN.2001.941393`.

[PM99] Yiannis Papadopoulos and John A. McDermid. "Hierarchically Performed Hazard Origin and Propagation Studies." In: *Computer Safety, Reliability and Security: 18th International Conference, SAFECOMP'99 Toulouse, France, September 27–29, 1999 Proceedings*. Ed. by Massimo Felici and Karama Kanoun. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 139–152. ISBN: 978-3-540-48249-9. DOI: `10.1007/3-540-48249-0_13`.

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs.” In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PR14] T. Prosvirnova and A. Rauzy. “AltaRica 3.0 project.” In: *Safety, reliability and risk analysis: beyond the horizon*. Ed. by R. D. J. M. Steenbergen. Boca Raton, Fla.: CRC Press, 2014, pp. 1121–1128. ISBN: 978-1-138-00123-7. DOI: \url{10.1201/b15938-169}.

[Pro14] Tatiana Prosvirnova. “AltaRica 3.0: a Model-Based approach for Safety Analyses.” Theses. Ecole Polytechnique, Nov. 2014.

[Rau08] A B Rauzy. “Guarded transition systems: A new states/events formalism for reliability studies.” In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 222.4 (2008), pp. 495–505. DOI: 10.1243/1748006XJRR177.

[SL14] Neal Andrew Snooke and Mark H. Lee. “Qualitative Order of Magnitude Energy-Flow-Based Failure Modes and Effects Analysis.” In: *CoRR* abs/1402.0581 (2014). arXiv: 1402.0581.

[Tar71] R. Tarjan. “Depth-first search and linear graph algorithms.” In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. Oct. 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[Tes] *A Fatal Crash has Recently Occurred between a Tesla vehicle in Self-driving Mode and a Tractor Trailer.* https://site.ieee.org/connected-vehicles/2016/06/30/fatal-crash-recently-occurred-tesla-vehicle-self-driving-mode-tractor-trailer/. Accessed: 2021-11-15.