Integrating Formal Verification into the Model-Based Development of Adaptive Embedded Systems
Dedicated to Giesela and Hillrich Alberts
Abstract

Embedded systems increasingly control safety-critical functionality in many domains. In this context, adaptation has become state-of-the art to meet the high demands on availability without additional hardware costs. However, adaptation significantly complicates system design. Model-based development is one approach to deal with the increased complexity by providing means to focus on adaptation and functionality of a system in isolation. Furthermore, formal verification of models and their properties increases reliability of adaptive systems. However, modelling concepts use a high-level of abstraction for specifying relevant system aspects, while input for verification tools is based on low-level mathematical concepts.

To alleviate this problem, this thesis presents a semantics-based integration of model-based development of adaptive embedded systems with existing verification techniques. The integration uses synchronous adaptive systems as formal semantics-based intermediate representation of adaptive systems. Static analysis can be applied to check models for structural consistency. Translating formal intermediate models to input of a theorem prover facilitates verifying properties directly. In order to make models amenable to automatic verification by model checking, verification complexity is reduced by model transformations using slicing techniques in different granularities and data domain abstraction. The model transformations are formally verified to be property-preserving using translation validation. Furthermore, compositional reasoning strategies reduce the verification effort significantly by splitting the global verification task into less complex local tasks. If the verification tasks are small enough to be handled by automatic verification tools, input for the tools is generated. The proposed integration framework is evaluated together with the development of an adaptive vehicle stability control system.
Acknowledgements

This thesis would not have been written without a number of people. Special thanks go to my supervisor Prof. Dr. Arnd Poetzsch-Heffter for giving me the opportunity to work in his group, for many inspiring discussions and for his guidance in scientific and other matters. Furthermore, I would like to thank Prof. Dr. Manfred Broy for acting as second reviewer and Prof. Dr. Peter Liggesmeyer for chairing my doctoral committee.

Over the last years, I have enjoyed collaborating with wonderful colleagues. I would like to thank my co-authors Rasmus Adler, Tobias Schüle, Eric Vecchié and Mario Trapp for the productive team work and for heated debates about how to integrate the different parts in our project. Many thanks go to Raphael Reitzig for implementing the great user interface of the AMOR wizard and to Jens Brandt for carefully proof reading this thesis. Further, I would like to thank the members of the Software Technology Group Jan Olaf Blech, Jean-Marie Gaillourdet, Marek Gawkowski, Kathrin Geilmann, Patrick Michel, Nicole Rauch, Markus Reitz, Jan Schäfer and Gabriele Weiler for the warm welcome in the group, for help with everyday computer problems, for answers to theoretical and practical questions and for many interesting discussions over coffee.

Besides the supportive working atmosphere, my personal environment was indispensable for completing this thesis. Very warm thanks go to my friends for their patience and encouragement when I was writing up and to my family for their constant support over many years. Finally, I want to express my deepest gratitude to Thomas for believing in me.
## Contents

1 Introduction ................................................................. 11  
1.1 Motivation ............................................................. 11  
1.2 Approach ................................................................. 13  
1.3 State of the Art ......................................................... 17  
1.4 Contribution and Outline ............................................. 24  

2 Modelling Embedded Adaptive Systems .................. 27  
2.1 Case Study - Adaptive Vehicle Stability Control .......... 27  
2.2 Modelling Adaptation Behaviour .............................. 31  
2.3 Related Work .......................................................... 36  

3 Specification of Synchronous Adaptive Systems .......... 39  
3.1 Synchronous Adaptive Systems ................................. 39  
3.1.1 A Formal Model for Synchronous Adaptive Systems .... 41  
3.1.2 A Language for Transitions .................................... 51  
3.1.3 Representing MARS Modelling Concepts .................... 55  
3.1.4 Functional Behaviour ............................................. 56  
3.2 Properties of Synchronous Adaptive Systems ............ 58  
3.2.1 Formal Specification Logic ..................................... 60  
3.2.2 Specifying Properties of Adaptive Systems .............. 64  
3.3 Related Work ............................................................ 67  

4 Verification of Synchronous Adaptive Systems .......... 69  
4.1 Static Analysis of Synchronous Adaptive Systems .......... 70  
4.1.1 Model Consistency Analysis .................................. 70  
4.1.2 Environment Model Analysis .................................. 71  
4.2 Theorem Proving of Synchronous Adaptive Systems ...... 74  
4.2.1 Verification of Fully Specified Modules ................. 74  
4.2.2 Verification with Underspecified Functionality .......... 82
4.3 Model Checking of Synchronous Adaptive Systems ........... 84
4.4 Related Work ........................................ 87

5 Model Transformations .................................... 91
  5.1 Property Preservation by System Simulation .................. 92
     5.1.1 Formal Foundations for Property Preservation .......... 92
     5.1.2 Translation Validation for Model Transformations ...... 107
  5.2 Model Reduction by Slicing ................................ 110
     5.2.1 System Slicing .................................... 112
     5.2.2 Module Slicing .................................... 115
     5.2.3 Adaptive Slicing .................................. 123
     5.2.4 Combining Reductions by Slicing ....................... 126
  5.3 Data Domain Abstractions ................................ 127
     5.3.1 Non-Deterministic SAS Semantics ....................... 130
     5.3.2 Abstracting System Models ........................... 133
     5.3.3 Abstracting Model Properties ........................ 141
     5.3.4 Abstraction for Verification of System Properties ..... 147
  5.4 Related Work ........................................... 152

6 Compositional Reasoning ................................... 157
  6.1 Foundations for Compositional Reasoning ..................... 158
     6.1.1 Property-Preservation for Combined Transformations ... 159
     6.1.2 Composition and Decomposition ....................... 166
     6.1.3 Maximal Models .................................... 175
  6.2 Compositional Reasoning Strategies ........................ 179
     6.2.1 Decomposition ...................................... 180
     6.2.2 Assume-Guarantee Reasoning ......................... 185
     6.2.3 Interface Abstraction ................................ 205
     6.2.4 Compositional Verification of Functionality ........... 209
     6.2.5 Properties of the Controlled System ................. 213
  6.3 Combining Verification Complexity Reductions ............... 218
  6.4 Related Work ........................................... 221

7 Experimental Evaluation ................................. 227
  7.1 AMOR Integration Framework ............................... 228
     7.1.1 Internal Representation of Verification Tasks .......... 230
     7.1.2 Translations from and to Verification Tasks .......... 230
     7.1.3 Transformations on Verification Tasks ................. 237
     7.1.4 AMOR User Interface ................................ 240
Chapter 1

Introduction

"The man of science has learned to believe in justification, not by faith, but by verification.”
(Thomas Huxley)

Integrating model-based development with formal verification offers a high potential for increasing reliability of embedded software systems. This thesis proposes a semantics-based integration of model-based development and formal verification using an independent formal intermediate layer. In this chapter, we present the motivation of our work and the main concepts of our approach. Moreover, we compare our integration framework with related work on integrating formal methods into the software development process.

1.1 Motivation

Embedded systems increasingly control safety-critical functionality in many domains, for example in automotive, medical or industrial control applications. In this context, adaptation and graceful degradation have become state-of-the-art to meet the high demands on safety and availability. The components of an adaptive embedded system are equipped with a number of different predetermined modes of operation, so-called configurations, that can be activated depending on the status of the system and its environment. Adaptation is frequently employed to react to changing environment conditions by selecting the best mode of operation in the current situation. Graceful degradation allows systems to continue operating properly in case of failures, e.g. defect sensors, by downgrading functionality
to a fallback operation mode. However, adaptation and graceful degrada-
tion significantly complicate system design due to the fact that adaptation
of a component affects the quality of its provided services, which may in
turn cause adaptations in other components. The result is a sequence of re-
configurations throughout the system that must be considered for system
design and correctness.

Model-based development \cite{Sel03} is one approach to deal with the in-
creasing complexity in modern software engineering. In particular in the
development of embedded \cite{SPHP02, SCDS04} and embedded
adaptive \cite{TAFJ07} systems, model-based development is a popular tech-
nique. The main idea is to shift the focus from writing source code to mod-
elling the system to be developed. A model in this context is an abstraction
of the system capturing essential features precisely in an intuitive manner.
This allows separation of concerns by focussing on one aspect at a time. The
possibility to model adaptation behaviour of systems independently of the
functional behaviour significantly reduces the design complexity in the de-
velopment of adaptive systems. In the design process, an initial model of a
system is stepwise refined until code generation is possible.

Furthermore, models can be analysed already at early stages of their
design. This allows discovering conceptual errors and design flaws before a
system is actually implemented. Hence, errors can be corrected more eas-
ily and less costly. In this context, formal verification of models and their
properties is a promising approach because it allows rigourously proving
that the developed systems meet their requirements. This is especially im-
portant in the context of safety-critical systems such as in the automotive
domain. However, while modelling concepts use a high level of abstraction
for specifying relevant system aspects, input for verification tools is based
on low-level mathematical concepts. For verification using theorem prov-
ing, the system under consideration has to be specified in the logic of the
prover. Using a set of deduction rules, properties of the considered sys-
tem can be derived. This requires heavy user interaction and expertise. In
contrast, model checking is an automatic verification technique. It uses a
system specification expressed as a transition system and a property in a
temporal logic. The state space of the system is systematically explored
and checked for the validity of the property. However, model checking is in
general restricted to finite-state systems. Furthermore, it suffers severely
from the state-explosion problem. Hence, it is only efficiently applicable for
systems of reduced sizes.
Due to these constraints, formal verification cannot be immediately applied in model-based development. Instead, the usability of formal concepts depends essentially on their integration in the design process [BLSS00] in order to be of effective use. An integration has to provide the modelling concepts with a formal semantics that captures the intuition at a high-level of abstraction. Modelling concepts should not be restricted because of the verification techniques to be applied. But the system designer should have the freedom to use intuitive modelling concepts neglecting the complexity they impose on verification. The integration framework should bridge the gap between modelling concepts and input required for verification tools. In this direction, verification complexity induced by the modelling concepts should be reduced without requiring additional effort from the system designer.

1.2 Approach

In this thesis, we propose a formal framework integrating model-based development of adaptive embedded system with existing verification techniques [ASSV07]. In this way, we bridge the gap between high-level modelling concepts and low-level verification input. We introduce a semantics-based intermediate layer providing models with a formal semantics essential for verification. Using this layer, verification tasks consisting of models and desired properties can be formulated in a semantically exact manner. This allows verifying model properties directly, e.g. using static analysis or theorem proving. In order to make models amenable to automatic verification by model checking the integration framework allows performing model transformations [SPH06b, SPH08] for verification complexity reduction. These transformations are formally verified to be property preserving using translation validation [BSPH07]. Furthermore, compositional reasoning strategies [SPH06a] can reduce verification effort significantly by splitting the global verification task into less complex local verification tasks. If the transformed models are small enough to be handled by automatic verification tools, input for these tools can be generated. Figure 1.1 gives an overview of the proposed integration approach.

As a starting point, we use the MARS modelling concepts (Methodologies and Architectures for Runtime Adaptive Systems) for the explicit modelling of adaptation behaviour by predetermined reconfiguration. The concepts have been successfully applied in industry and academia for several years.
and provide a seamlessly integrated approach for the development of adaptive systems [TAFJ07, ASS08]. In MARS, adaptation behaviour is modelled as a separated view by endowing data flow with qualities. In a module, a set of fixed configurations is specified which the module can adapt to. The selected configuration only depends on the provided input qualities attached to the functional data. This allows modelling adaptation behaviour of a module without reference to functionality alleviating design complexity. Composition of modules to a system provides a complete model of the system’s adaptation behaviour.

Synchronous adaptive systems (SAS) [SPH06b, ASSV07] serve as semantic basis for the formal intermediate representation in the integration framework. They are designed to represent MARS modelling concepts at a high level of abstraction capturing the separation of functional and adaptation behaviour also on the formal level. An SAS system is built from a set of modules. Each module contains a set of functional configurations the module can adapt to. The adaptation behaviour is captured by an adaptation aspect encapsulated inside the module for determining the appropriate functional configuration dependent on the module’s environment. This separation of adaptation and functionality can later be exploited for reducing model complexity with respect to verification. The SAS semantics is based on the notion of state-transition systems. Modules are executed in a time-synchronous fashion providing an efficiently analysable abstract model of a system’s timing behaviour. An embedding of the MARS modelling concepts into SAS provides MARS models with a formally exact semantics. Since in MARS functionality is specified independent from adaptation behaviour, we also deal with modules where functionality is only specified by means of pre-/postcondition specifications of functional behaviour.
Properties to be verified over adaptive systems can be classified in two dimensions. First, properties can refer to the adaptation behaviour, to the functional behaviour or to both aspects in combination. Second, properties can be generic, i.e. relevant for the whole class of adaptive systems, or application-specific. An important generic property of the adaptation behaviour is stability of the adaptation process [SPH06a, ASSV07] as in SAS systems adaptation is not regulated by a central authority. Other important generic properties of the adaptation process are safety or deadlock-freedom. Application-specific properties refer for instance to the correct implementation of the functional configurations or a correctly implemented fallback mode of operation in safety-critical systems. In order to capture properties over SAS models formally, we provide the temporal logic $\mathcal{L}_{SAS}$ for specification of model properties.

In the intermediate layer of our integration framework, a verification task consisting of a model and a property to be verified can be represented in a formally exact manner. As the intermediate representation is independent of concrete verification technique or tool, different formal verification techniques can be applied to complete the verification task. We use static analysis methods [NNH99] to guarantee that models are consistent with respect to syntactical constraints. Furthermore, static analysis can discover flaws in the structural design of the models at a very early design stage, e.g. by checking the dependencies of data types. By giving an embedding of the formal SAS semantics into the Isabelle/HOL theorem proving environment [NPW02], we are able to verify properties formulated over SAS models directly using theorem proving techniques.

For making SAS models amenable to automatic verification by model checking, our integration framework comprises model transformations for verification complexity reduction. First, slicing techniques [Wei84, Tip95], originally used for programming languages, are adapted to SAS models [SPH08]. Slicing for model reduction is fully automatic requiring no user-interaction while preserving the properties of the system. For a particular property to be verified, the parts of the model not influencing its validity are determined and removed. Slicing on the formal intermediate representation can be performed in different levels of detail. To this end, structural information can be exploited giving a fine-grained control over the analysis effort required for a reduction. For SAS models, we propose system slicing on the connection structure of the system modules, module slicing analysing the internal structure of modules and adaptive slicing exploiting the separation of adaptation and functional behaviour. Second,
data domain abstractions \cite{SPH06b} on SAS models can be employed to reduce infinite-state models to finite state and further to reduce large data domains to smaller abstract domains. A resulting finite state model can be analysed by a model checker whereas the original infinite state model cannot be handled. Both kinds of transformations can be proven to preserve the property to be verified by a translation validation approach \cite{BSPH07} integrated in our framework. For each run of a transformation algorithm, a proof script for the Isabelle/HOL theorem prover is generated certifying the correctness of the transformation.

Besides model transformations, compositional reasoning strategies exploit the modular structure of SAS for reducing the size of verification tasks \cite{SPH06a}. First, a model and also the property can be decomposed into a set of smaller models and local properties that can be checked in isolation. From the verification of the local properties, the global property can be deduced. If simple decompositions are a too coarse abstraction, assume-guarantee reasoning \cite{Pnu85} can be applied. The model is split into two parts where one part is assumed to satisfy a certain property. Under this assumption, the second part is shown to satisfy the guarantee. If also the first part can be shown to satisfy the assumption, the guarantee can be established as a property over the complete model. In order to reduce assume-guarantee reasoning to a standard model checking problem, we present a technique for construction of a maximal SAS model consistent with respect to an assumption \cite{GL94}. For checking that a system part satisfies the guarantee, it is composed with the maximal model of the assumption and translated to input for the model checker. If an assumption for assume-guarantee reasoning cannot be provided, interface abstractions can be applied to reduce a part of the model with respect to its interface behaviour \cite{KP98} by combination of the model transformation techniques. For the particular class of adaptive systems, we propose a compositional proof rule allowing verification of functional behaviour without considering adaptation behaviour. In embedded systems, the characteristics of the controlled system have to be considered for verification of the model for the controller. In this direction, we propose an approach to model the controlled system as an environment for the controller model and to verify properties in an assume-guarantee reasoning style.

The proposed integration framework is prototypically implemented in the AMOR tool (Automatic MOdel VerifieR) \cite{ASSV07} and has been evaluated together with the development of an adaptive vehicle stability control system \cite{ASS08}. The independent intermediate representation pro-
vides a platform for the integration of model-based design with a heterogeneous set of formal verification tools. The component-based structure of AMOR is centred around a verification task consisting of an SAS model and a $L_{SAS}$ property to be verified. Different model transformation techniques and compositional reasoning strategies can be combined in order to achieve the best possible verification complexity reduction. As the reductions are carried out on the intermediate layer, they are independent of the used modelling front ends and verification back ends and can be proved correct in isolation. The component-based architecture of AMOR further facilitates adding new transformations easily. Moreover, different modelling front ends and different verification back ends can be integrated into AMOR by providing translations to and from AMOR verification tasks.

1.3 State of the Art

By means of formal verification, properties of models can be rigorously proven which is in particular important in the area of safety-critical and embedded systems. In adaptive systems, formal verification of the adaptation behaviour has become an active area of research [KB04, Str05, ZC05, ZC06] in order to find models that lead to reasonable and safe degradation behaviour. However, all approaches so far provide no constructive modelling technique. Instead, they take a global specification of the adaptation behaviour in a low-level formalism as given whereas providing this specification is hardly feasible for complex systems.

Nonetheless, integrating model-based development with formal verification with respect to a broader class of systems is an active topic of research. A large body of work concentrates on integrating one modelling formalism with one formal verification tool by capturing the modelling concepts directly in the input language of the verification tool. Other approaches use an intermediate representation of models and properties bridging the gap between modelling concepts and verification tools, but still focus on a single tool. Some frameworks have been proposed for integrating different verification tools based on an intermediate model representation that are most closely related to the approach of this thesis.

In this section, we review approaches towards integrating model-based development with formal verification in three categories: (1) approaches directly translating models to verification input, (2) approaches translating models to one verification tool using an intermediate layer and (3) approaches integrating of several verification tools. Afterwards, we compare
integration approaches considering several verification tools with the approach presented in this thesis. More related work with respect to particular techniques applied for realising the proposed integration framework can be found in later chapters.

*Direct Translation of Models to Verification Input*

Many approaches aim at integrating UML-based modelling with formal verification because UML is a popular choice as a modelling language in model-based development. Mostly, model checkers are used in this direction as they facilitate automatic verification. In these approaches, the applied modelling concepts are directly translated into the input language of a verification tool by capturing their semantics by means of the respective input language. In general, properties to be verified over the considered models are also described in the input language of the verification tool, in a case of model checkers in temporal logic.

The considered UML modelling concepts are mostly UML state charts or the closely related Statemate [HN96] state charts. State charts can in general be easily mapped to input concepts of model checkers. In [LP99], the vUML tool takes UML state charts as input translating them into Promela, the input language of the SPIN [HNS00] model checker. SPIN checks the models for a number of predefined run time errors such as deadlocks or events sent to terminated objects and reports error traces as UML sequence diagrams. In the HUGO project [SKM01], UML statecharts are translated to Promela, but checked against UML collaboration diagrams as specification. In [LMM99], the UML state chart semantics is expressed by extended hierarical automata which are then translated to Promela. A similar approach in this direction [MLSH98] translates Statemate state charts into Promela via extended hierarchial automata. In [GBSQF05], the model checker SMV [McM93] is used for verification of UML state charts and activity diagrams. Similarly, [SCH02] uses SMV as verification backend for UML state charts. [RW06] presents an approach translating Statemate state charts to the process algebra CSP for checking properties by the model checker FDR. There are also approaches towards the interactive verification of UML models allowing to deal with infinite-state models without prior abstraction. In [BBK+04], UML state charts are verified in the KIV theorem prover by symbolic execution. Verification of UML class diagrams and state charts with the B method is proposed in [SB06].

Also, model-based design for real-time systems has been integrated with formal verification using verification tools able to deal with time constructs.
1.3. State of the Art

[KMC02] presents a translation of time extended UML state charts to the input language of UPPAAL in order to verify their properties as well as consistency between the collaboration and the state chart view of a system. [EKHL03] presents an approach where a subset of UML-RT is translated to the process algebra CSP and deadlock freedom is verified with the model checker FDR. The Fujaba Realtime Tool Suite [BGH+05] aims at the model-based development of safety-critical networked real-time mechatronic systems in combination with verification by model checking and code generation. The state explosion problem is addressed by a compositional verification approach [GTB+03] exploiting structural features of the considered models. Recently, system development with Simulink has been integrated with formal verification. [MBR06] selects a finite state set of Simulink modelling constructs for translating models to the NuSMV model checker [CCGR00]. [CDS07] translates Simulink models to the timed interval calculus and uses PVS [JR06] to verify discrete and continuous behaviour. The main problem with verification of Simulink models is that there is no clear formal semantics of the models such that each translation provides the modelling constructs with its own semantics.

Translation of Models with an Intermediate Representation

All approaches considered so far translate one source modelling language directly into the target language of a verification tool. During the translation, either the semantics of the modelling concepts is defined by the input language of the verification tools or by some low level formalism. Therefore, the semantic gap between high-level modelling constructs and input for verification tools is large. Furthermore, the modelling concepts that can be dealt with in verification are limited to the capabilities of the verification tools. To counter this, some approaches to UML verification use an intermediate representation that aims at closing the gap between modelling and verification.

The Rhapsody UML Verification Environment [STMW04] supports the verification of UML models using the VIS model checker via an intermediate language called SMI (System Modelling Interface). SMI is a simple imperative language for representing symbolic transition systems. Properties can be specified as life sequence charts (LSCs) [DH99], an extension of message sequence charts with conditions for mandatory and possible behaviour. LSCs are translated to CTL formulae and used as specification in the VIS model checker.
Chapter 1. Introduction

[XLKB04] proposes an approach for verifying an executable subset of UML, xUML, with the Cospan model checker using common abstraction representations (CAR) as intermediate layer. A CAR is a simplified representation of the target language where the CAR semantics is given by a mapping of the source language semantics to the basic concepts of the CAR. It is intended to minimize the gap between source and target languages such that only the translations from source to CAR and from CAR to target have to be validated. On a CAR, state space reductions [XB02] can be performed before the CAR is translated to the target language. Reductions include decomposition of properties over the structure of the model and abstractions of data values or irrelevant system parts. This approach [XB02, XLKB04] differs from ours in that a CAR itself has no formal semantics, but is defined by a translation from source to target language. This restricts verification to the capabilities of one target verification tool.

Integration of Different Verification Tools

In the literature, there are also approaches [BGL+00, BLSS00, BGO+04] closely related to the approach presented in this thesis that aim at integrating different verification techniques with model-based development. The approaches are based on a formal intermediate layer to represent models before verification input is generated.

SAL (Symbolic Analysis Laboratory) [BGL+00] is a framework for combining different tools for program analysis, theorem proving, and model checking towards the verification of concurrent systems. A key part of the SAL framework is an intermediate language for describing transition systems in a modular fashion. This language serves as target for translators that extract the transition system description from other modeling and programming languages, and as a common source for driving different analysis tools. However, SAL itself does not comprise a model-based language as input, instead it relies on external translators to the SAL intermediate language. As verification backends, SAL supports the theorem prover PVS [JR06] and the model checker SMV [McM93]. In SAL, a slicing algorithm of intermediate models is implemented, but the modular system structure is not exploited for reductions. Furthermore, SAL intermediate models can be reduced by Boolean abstraction.

AutoFocus [BHS99, HMS+98] is a CASE tool for the model-based development of correct embedded systems software. It offers formally founded specification techniques for different views onto a system. The specification techniques include system structure diagrams to model the system struc-
1.3. State of the Art

ture by synchronously operating components, state transition diagrams to model the behaviour of system components and extended event traces to capture behaviour by exemplary communication histories between components. AutoFocus allows the definition of hierarchy in all these constructs. The modelling concepts have a common formal semantics based on input and output streams and state-transition systems. Data type definitions use the algebraic notion of abstract data types. The common formal foundation allows consistency checks between different views of the system. Consistency checks include syntactical correctness as well as scenario consistency, i.e. that an state transition diagram implements an extended event trace, and refinement consistency, i.e. that the behaviour of a super component is a super set of the behaviours of the subcomponents. The semantics of the modelling concepts can be represented in the relational $\mu$-calculus for the model checker $\mu$cke such that consistency checks can be performed by model checking. Furthermore, AutoFocus offers simulation as well as code generation for rapid prototyping.

In the Quest project [BLSS00, Slo00], AutoFocus has been integrated with the CTE tool for test case generation and management, the models checkers SMV [McM93] and SATO (bounded model checking) and the interactive theorem proving environment VSE. The unifying intermediate representation in the integration is made up by the mathematical basis of the AutoFocus modelling constructs. However, special encodings for the integrated tools are necessary. For instance, translating AutoFocus models to SMV [PS99] requires to flatten the hierarchy of AutoFocus models and to represent algebraic data types by integer constants. As VSE has an asynchronous action-based semantics a scheduler at each level of hierarchy has to be added to capture the synchronous semantics of AutoFocus. For system models, that cannot be translated directly to the input of a model checker, because they are infinite or very large, Quest supports model reduction by abstraction. Abstractions are shown to be property preserving by trace inclusion [MN95] in the interactive theorem proving environment VSE.

The IF Tool Set [BFG+99b, BGO+04] is an environment for modelling and verification of heterogeneous realtime systems. It is based on the intermediate IF representation for timed asynchronous automata. As modelling front ends SDL and more recently UML state charts are supported and translated into the IF representation that is rich enough to capture the core concepts of both input modelling languages. Properties to be verified over a system can be specified by special observer modules. On the intermediate IF representation, static analysis techniques are applied in order
to simplify models for efficient verification. The reductions include dead code elimination, live variable analysis [BFG99a], redundant clock reduction, constant propagation and removal of irrelevant variables. While the other reduction techniques are property independent, removal of irrelevant variables is performed on basis of an input set of irrelevant variables. This set depends on the considered property and is either given by the user or inferred from the system description. However, the modular structure of IF models is not exploited for this reduction that is similar to slicing. After model reduction, output for verification and validation tools can be generated, for example Promela code for the SPIN model checker. With respect to other reduction techniques, e.g. abstractions or compositional reasoning, IF relies on the features of the applied verification tools. In case studies, such as verification of protocols for wireless ATMs [JG01] or verification of parts of the ARIANE5 flight program [BGO+04], the authors have shown that a combination of the proposed reduction techniques enables verifying large-scale industrial scenarios.

Comparison with Integration Approach of this Thesis

To sum up, the work on integration of model-based development with formal verification can be classified according to different dimensions. First, the approaches can be distinguished according to which kind of input models they accept. Popular choices for input models are UML and its extensions, SDL and very recently also Simulink. Mostly, a clearly defined subset of the modelling language is supported by the integration, e.g. UML state charts. Second, the approaches can be classified according to the applied formal verification techniques. These are either model checkers, e.g. SPIN or SMV, or theorem provers, such as KIV or VSE. A distinguishing feature in this direction is how close the supported modelling concepts have to be at the input of the verification tools, because the more high-level the modelling concepts at input level are, the larger is the gap to be bridged by the translation. Third, integration approaches differ in the usage of an independent intermediate layer. In many approaches, models are directly translated to the target language of a verification tool. If an intermediate layer is used, its semantics can be defined in itself or by a translation. Furthermore, the intermediate layers differ whether they are close to the target models or close to the input of the verification tools and whether they facilitate the use of several verification tools. Fourth, integration approaches can be distinguished whether they restrict the input models such that verification is efficiently possible by model construction or whether
they offer model reductions for making them amenable to verification. An interesting aspect in this direction is whether the reductions are validated themselves. Finally, approaches can be distinguished according to the specification formalism for properties. Some approaches only consider a fixed set of properties to be checked, whereas others allow specifying requirements in the input format of the verification tools. Few approaches come up with means to specify properties in an intuitive manner or assist with property specification.

Table 1.1 shows how the frameworks using an explicit intermediate layer for integrating multiple verification tools SAL [BGL+00], IF [BFG+99b, BGO+04] and AutoFocus/Quest [BLSS00, Slo00] compare in these dimensions with the AMOR integration framework we present in this thesis.

• The main difference between the existing approaches and our framework is that our work explicitly focusses on the verification of adaptive systems. We provide the MARS modelling concepts for adaptive systems [TAFJ07, ASS08] with a formal semantics making models of adaptive systems amenable to formal verification.

• In our framework, the formal intermediate representation is close to the concepts used on the modelling level allowing a translation that is easy to validate. In contrast, the other frameworks rely on more low-level mathematical concepts inducing a larger gap between modelling concepts and its formal representation.

• Input models are reduced by transformations to facilitate efficient verification in our framework. All transformations are proved correct using translation validation. While also in the other frameworks, reduction techniques on the input models are proposed, only AutoFocus uses a theorem prover to verify the abstractions formally. Nevertheless, establishing transformation correctness is important to facilitate a seamless integration of transformations.

• The model transformations used in the other frameworks are only a subset of the reductions implemented in the AMOR framework. Rather, the other approaches rely on the capabilities of the integrated tools for model reduction. However, implementing transformations on the intermediate layer itself allows a fine-grained control over the reductions for designated verification back ends and an easy addition of further transformations, if necessary.

• Compositional reasoning strategies exploiting the modular structure of SAS models can be used in order to reduce verification problem sizes.
<table>
<thead>
<tr>
<th>Input Models</th>
<th>SAL[BLG^00]</th>
<th>IF[BGO^04]</th>
<th>AutoFocus / Quest[BLSS00]</th>
<th>AMOR[ASSV07]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Backends</td>
<td>SMV, PVS</td>
<td>SPIN, CADP, INVEST et al.</td>
<td>VSE, SMV, SATO, µcke</td>
<td>Averest, NuSMV, Isabelle/HOL</td>
</tr>
<tr>
<td>Abstractions</td>
<td>Boolean Abstraction</td>
<td>Data Domain Abstraction</td>
<td>Data Domain Abstraction</td>
<td></td>
</tr>
<tr>
<td>Compositional Techniques</td>
<td></td>
<td></td>
<td></td>
<td>Decomposition, Assume Guarantee Reasoning, Interface Abstraction</td>
</tr>
<tr>
<td>Property Specification</td>
<td>Temporal Logic</td>
<td>Temporal Logic, Observer Modules</td>
<td>Temporal Logic</td>
<td>Temporal Logic ($\mathcal{L}_{SAS}$)</td>
</tr>
</tbody>
</table>

Table 1.1. Comparison of Integration Frameworks

Neither of the other frameworks uses compositional techniques on the intermediate level.

- In the AMOR framework, the temporal logic $\mathcal{L}_{SAS}$ is used for expressing properties over adaptive systems. Similar, all other integration frameworks use temporal logic specifications for properties.

### 1.4 Contribution and Outline

In this thesis, we make the following contributions: We propose an integration framework for model-based development and formal verification of adaptive embedded systems [ASSV07, ASS08] bridging the gap between high-level modelling concepts and and low-level verification input. We give a formal semantics-based model for adaptive embedded systems (SAS) capturing MARS modelling concepts at a high-level of abstraction [ASSV07]. For expressing properties over the models, we provide the temporal logic $\mathcal{L}_{SAS}$. Based on the formally exact representation, we show
how models and their properties can be verified directly by static analysis and theorem proving. In order to make models amenable to verification by model checking, we present model transformations by slicing and data domain abstraction for verification complexity reduction [SPH08, SPH06b]. The transformations are proved correct with a translation validation approach [BSPH07]. Additionally, we propose compositional reasoning strategies [SPH06a] in order to split complex verification tasks into a set of simpler subtasks. We have prototypically implemented the proposed concepts in the AMOR tool and evaluated them at the development of an adaptive vehicle stability control system [ASS08].

The outline of this thesis is as follows: In Chapter 2, we describe the adaptive vehicle stability control system used as case study in this thesis and introduce the high-level intuitive modelling concepts of MARS [TAFJ07, ASS08] for model-based development of embedded adaptive systems. In Chapter 3, we present SAS as formal semantics-based representation of adaptive systems and explain how MARS modelling concepts are mapped to SAS. Furthermore, we define the temporal logic $L_{SAS}$ to express properties over SAS models. In Chapter 4, we focus on verification of SAS models and their properties by static analysis, theorem proving and model checking using the previously introduced formal representation. In Chapter 5, we present slicing and data domain abstraction as transformations on SAS models in order to make them amenable to automatic verification by reducing the size of the model itself. The transformations verified correct by translation validation. In Chapter 6, we develop compositional reasoning strategies for SAS models in order to reduce the overall verification problem size. Compositional techniques include decompositions, assume-guarantee reasoning, interface abstractions and special compositional techniques for adaptive system. Furthermore, we show how to deal with assumptions on the controlled system in the embedded domain. In Chapter 7, we present our prototypical implementation of the proposed framework in the AMOR tool and results of the empirical evaluation in the case study. Chapter 8 concludes this thesis with a summary of the contributions made and an outlook to future directions of research.
Chapter 2

Modelling Embedded Adaptive Systems

"The soul never thinks without a picture."

(Aristotle)

Adaptation and graceful degradation have become state-of-the-art in embedded systems design to meet high demands on safety and availability without additional hardware costs. System modules are equipped with a set of predetermined behavioural configurations a module can adapt to. Adaptation is frequently employed to react to changing environment situations that require to switch between different modes of operation. Graceful degradation enables the systems to continue operating properly in case of failures, e.g. defect sensors. In this chapter, we present a case study of an adaptive embedded system in the automotive domain that is used as a running example throughout this thesis. Furthermore, we introduce the MARS modelling concepts for adaptive systems separating functional from adaptation behaviour in order to reduce design complexity. These modelling concepts serve as basis for the input to our integration framework.

2.1 Case Study - Adaptive Vehicle Stability Control

Adaptation is a state-of-the-art technique in modern vehicles to increase their reliability and survivability in case of changing environment conditions as well as in case of failures. As many functions in an automobile are based on approximated physical models assuming certain driving situations, it is necessary to adapt to the most reasonable calculation variant for a sensor value depending on the current situation. Additionally, a system can downgrade its functionality autonomously if a failure cannot be
compensated by dynamic adaptation. Of course, the system can upgrade again in case of transient failures to provide the best functionality possible with the currently available resources.

The adaptive vehicle stability control system [ASS08] used as case study in this thesis consists of three typical software-intensive vehicle functions, which have been implemented on a remote controlled car scaled 1:5. The first vehicle function is a steering angle delimiter, which restricts the steering angle depending on the current velocity of the car. This function prevents that the car starts spinning if the steering angle intended by the driver might cause instabilities. The second function is a traction control, which monitors the slip of the rear wheels because the car possesses a rear wheel drive. If the slip exceeds a given threshold, the acceleration of the car is restricted by reducing the gas at the rear wheels. The third function implements a yaw rate corrector influencing the brakes. For example, if the driver wants to go straight on, yaw rate correction keeps the yaw rate close to zero. In principle, a yaw rate corrector can be used in any driving situation. In our system, however, the function only considers straight-forward driving. This simplification is well-suited for our purposes, since it eases the implementation of the functional behaviour, but provides the same challenges for modelling and analysing adaptation and functional behaviour.

In order to implement the described functionality, the car is equipped with typical sensors. At each wheel, there is a sensor measuring the wheel speed $v_{\text{wheel}}$. Additionally, there are sensors measuring the longitudinal acceleration $a_x$, the lateral acceleration $a_y$, and the yaw rate $\text{yaw}_\text{rate}$. Besides the values measured by sensors, the system receives as input the desired steering angle $\text{steering}_\text{angle}_\text{input}$ and the desired gas or brake force as one combined value $\text{brakeGas}_\text{input}$ via remote control from the user. A value greater than zero for the brake/gas input $\text{brakeGas}_\text{input}$ corresponds to an acceleration, a value smaller or equal to zero to braking. The brake and gas input $\text{brakeGas}_\text{input}$ is normalised into a brake value $\text{brake}$ and a gas value $\text{gas}$ by a $\text{Brake}$ and a $\text{Gas}$ sensor module, respectively. The system controls four servos: The first servo controls the gas and actuates the rear brakes $\text{rear}_\text{brake}_\text{cntrl}$. Hence, the rear wheels are accelerated or decelerated with the same force. The front brakes have their own servos and are actuated independently from each other by $\text{left}_\text{brake}_\text{cntrl}$ and $\text{right}_\text{brake}_\text{cntrl}$. The fourth servo controls the steering $\text{steering}_\text{angle}_\text{cntrl}$. 
The system architecture is depicted in Figure 2.1. It is split into logical sensors, controllers and logical actuators. The logical sensors and logical actuators implement the mapping between physical sensor and actuator values and their logical interpretation. Sensor modules take physical sensor values as inputs and process them for the controller modules. Actuator modules produce physical output values for the respective actuators. The controller part of the system contains three modules SteeringAngleDelimiter, YawRateCorrector and TractionControl implementing the vehicle functions described above based on logical sensor and actuator modules.

The sensor modules are self-adaptive as the most reasonable processing variant for a sensor value typically depends on the current driving situation. This even holds for sensor values, where the relation to measured sensor values seems to be straightforward. For example, the module computing the lateral acceleration $ay$ implements two different modes of operation, each providing a different quality level. In the following, we will call these externally visible modes of operation configurations. For instance, configuration Measured derives $ay$ from the values provided by an acceleration sensor. However, when the car is driving down- or uphill, the acceleration sensor values are influenced by gravity. For this reason, an alternative configuration VYawVCarRefBased is used, which calculates $ay$ from the speed of the car along the x-axis $v_{carRef}$ and the angular speed of the car around its z-axis $v_{yaw}$.

Dynamic adaptation in the sensor modules is a powerful and cost-efficient way to minimise the effects of sensor defects. In many cases, it is even possible to fully compensate for the occurrence of faults. For instance, the module VYawCalculation has five different configurations to determine the angular speed of the car around its z-axis $v_{yaw}$. Configuration Measured uses a yaw rate sensor, configuration SteeringBased de-
rives $v_{\text{yaw}}$ from the steering angle and the speed of the car along the x-axis $v_{\text{carRef}}$, configuration \textit{FrontWheelBased} computes $v_{\text{yaw}}$ from the steering angle and the speed of the front wheels, configuration \textit{RearWheelBased} uses the rear wheel speeds to determine $v_{\text{yaw}}$, and configuration \textit{AyBased} uses the lateral acceleration $a_y$ and $v_{\text{carRef}}$. In all configurations, the quality of the provided output is sufficient to perform a yaw rate correction. This means that the module $V_{\text{YawCalculation}}$ is able to adapt to the most appropriate configuration according to the availability of the sensors without downgrading the functionality of the system.

In contrast to sensors and actuators, the configurations of the controller modules directly correspond to degradation levels of the vehicle functions. Since the considered vehicle functions are reasonably simple, each of the three controller modules shown in Figure 2.2 implements only one safe fallback layer. Besides the configuration that provides full functionality, there is another configuration that implements a safe subset of the functionality based on minimal input to ensure that the system is still operational even if the full functionality cannot be provided. Additionally, each module has a configuration \textit{Off}, in which the component is shut down. This configuration is primarily used for validation and verification as shutting down modules is undesirable in practice.

The controller module \textit{SteeringAngleDelimiter} contains the configurations \textit{Controlled}, \textit{Forwarded}, and \textit{Off}. Configuration \textit{Controlled} requires the steering angle input $\text{steering\_angle\_Input}$ from the driver and the current velocity of the car $v_{\text{carRef}}$. It checks whether the desired steering
2.2 Modelling Adaptation Behaviour

angle is safe at the current speed. If it is safe, it forwards the input steering angle to the steering angle actuator module by its output delimited_steering_angle. Otherwise, it restricts the steering angle to the maximal safe steering angle at the current speed. This avoids unstable driving situations. In order to prevent a total loss of the delimited steering angle, when the velocity of the car cannot be determined or is erroneous for some reason, the module has the fallback configuration Forwarded, which only requires the desired steering angle steering_angle_Input as input and assumes that the steering angle servo for setting the delimited steering angle delimited_steering_angle is available. This configuration simply forwards the input steering angle to the delimited steering angle.

The yaw rate correction module YawRateCorrector has a fully functional configuration YawCorrection, a fallback configuration Forwarded and the shutdown configuration Off. In configuration YawCorrection, it sets the brakes of the two front wheels left_brake_cntrl and right_brake_cntrl independently in order to control the yaw rate of the car given by the input v_yaw with respect to the current steering angle delimited_steering_angle. In the fallback configuration Forwarded, only the input brake force brake and the availability of the brake servos at the wheels is required in order to compensate for a failure of the yaw rate input. Then, the output brake forces are set to the input brake force brake.

Full functionality of the traction control is provided in configuration On of module TractionControl. This configuration sets the output gas value delimited_gas depending on the input gas value gas and the current slip of the wheels slip to avoid loss of traction. In order to compensate for a failure of the slip value, the module implements a fallback configuration SlowStart, where the output gas value delimited_gas is incremented at most by the maximal admissible acceleration. This configuration assumes that the input gas value and the rear brake and gas servo for setting the delimited_gas are available.

2.2 Modelling Adaptation Behaviour

As we have seen in the vehicle stability control example, adaptation by predetermined reconfiguration increases safety and reliability of embedded systems by adjusting the behaviour to the current environment situation or in case of failures. However, adaptation significantly complicates system design due to the fact that adaptation of a module affects the quality of
its provided services, which may in turn cause adaptations in other modules. As a result, sequences of adaptations may take place that are hard to analyse. A promising approach to deal with the increased complexity posed by adaptation is model-based design. As a major advantage, model-based design allows focusing on the needs of each phase in the design process in isolation by capturing the considered concepts in an intuitive but accurate manner. Furthermore, it supports the validation and verification of adaptation behaviour, before the actual functionality is implemented.

In this direction, the objective of the modelling concepts developed in the MARS (Methodologies and Architectures for Runtime Adaptive Systems) project is the explicit modelling of adaptation behaviour. The concepts have been successfully applied in industry and academia for several years and provide a seamlessly integrated approach for the development of adaptive systems [TAFJ07, ASS08]. The major difficulty in modelling adaptation behaviour are complex interdependencies between the modules of a system. To solve this problem, MARS models use the concept separation of concerns by separating functional from adaptation behaviour and the concept divide and conquer by defining the adaptation behaviour modularly within the system. This allows encapsulating the specification of adaptation behaviour by modelling it as a separated view within the modules. In this way, developers can focus on the adaptation behaviour of single components, until the whole system is specified.

A MARS system consists of a set of modules that communicate with each other by passing signals via ports. This is a common notion found in modelling languages and complies with the definition of architecture description languages [MT00a]. In contrast to non-adaptive modules, MARS modules have several functional behaviours, so-called configurations, which are selected dependent on the current situation of the module’s environment. In the adaptation-specific view of MARS models, functional behaviour is left unspecified. Instead, a clear interface to the functional behaviour of the configurations is provided. In later stages of the design process, models of the functional behaviours are inserted as refinement in place of the interfaces.

Quality Descriptions

A modular definition of adaptation behaviour is indispensable for handling the enormous complexity of most systems. For this reason, a quality flow in the system is established making modular definitions possible. Besides the actual data, each signal has an additional quality description. To this
end, ports are typed by datives (extended data type for adaptive systems) that do not only fix which data values a signal may take, but also how the quality of this data can be described. A dative consists of a data type and a quality type. The former describes the type of the data value, such as integers or real numbers. The quality type provides type-specific quality information. A general purpose quality information, like the relative error, is not reasonable in many cases, e.g. for Boolean signals. Since the quality is part of the type definition, module designers are able to define the adaptation behaviour solely on basis of quality descriptions available at a module’s local interface. Additionally, they define how the current quality of the provided signals is determined.

In order to define the quality of a signal, it is necessary to know which behavioural variant has been used to determine a value. In the first place, a quality type is defined by a set of possible modes. A developer using a signal knows the deficiencies associated with a certain mode and decides how a module must adapt in order to compensate for these deficiencies. Additionally, mode attributes can be used to describe the signal quality more precisely using mode-specific characteristics. Consequently, a mode is described by the mode itself and a set of mode attributes.

As an example, Figure 2.3 shows the definition of the dative \( v_{\text{yaw}} \). Its quality type contains five modes: The mode \( \text{sensor based} \) refers to a computation from the yaw rate sensor. The mode \( \text{wheel based} \) indicates that \( v_{\text{yaw}} \) is derived from the rear wheels or the front wheels. A deficiency associated with this mode is that the speed of the wheels may be influenced by the slip a wheel can have. So, a mode attribute \( \text{slip probability} \) is attached to the modes giving a measure how the wheel speeds used in the computation are influenced by a potential slip of the wheels. The mode \( \text{steering based} \) refers to a computation using the steering angle and \( \text{ay based} \) to a computation from the lateral acceleration and the vehicle speed. The last mode \( \text{unavailable} \) denotes that the value of the yaw rate computation cannot be used for further computations due to a complete failure of the module.

**Modules**

Based on datives, developers can modularly define the adaptation behaviour of single modules using two extensions made to conventional modules. First, the behaviour specification is not directly assigned. Several configurations can be assigned to a module, each of them representing one behavioural variant. Second, in addition to the input/output interface of the functional ports, a required/provided interface for the qualities in the
adaptation behaviour is defined. This distinction is used for describing the direction of the quality flow that is not always identical to the direction of the data flow between two connected module ports. Although a connection is typed with one dative, the data part of the dative flows from an output port of one module to an input port of another module, while the quality flows from a provided port to a required port. This is for instance the case for an actuator where a data value is propagated to the actuator while the actuator’s status is conveyed to the functional unit via the signal’s quality. For example, the rear brake gas servo propagates its status to the yaw rate corrector module while the actual functional value for the brakes is sent to the servo from the yaw rate corrector module.

**Configurations**

A module is defined by its interface containing input, output, required and provided ports as well as a set of predetermined configurations it can adapt to. A configuration is defined by the following elements: (1) an intuitive specification of the associated behavioural variant, (2) a guard defining under which conditions the configuration can be activated, (3) a priority and (4) an influence defining how the quality of the provided signals is determined. In a MARS configuration, the implemented functionality is assumed to be inserted later in the design process. However, MARS configurations provide a clean interface to the functionality of a configuration.
separating it completely from the adaptation behaviour. In our running example, the module \textit{V.YawCalculation} depicted in Figure 2.4 can be in one of six configurations, depending on how the yaw rate is determined. The inputs for this module are the sensor values coming from the environment and other values from other sensor modules. Their qualities are defined as required. The output $v_{\text{yaw}}$ is defined with a provided quality.

A guard is a Boolean expression. If the guard evaluates to true at run time, the configuration can be activated. Operands of guards are quality descriptions of required signals. A guard defines which signals are required in which mode and which values the mode attributes may have. For instance, the guard of the configuration \textit{SteeringBased} defines that the quality of the car’s velocity $v_{\text{carRef}}$ has to be in the mode \textit{wheel based} or in the mode \textit{ax based}. The quality of the steering angle input has to be \textit{available}. Additionally, it could be enforced that the mode attribute \textit{slip probability} is in a certain range. Figure 2.5 shows a graphical representation of the guard of configuration \textit{SteeringBased}. Often, guards of several configurations are satisfied at the same time. Therefore, an unambiguous priority is assigned
to each configuration. At run time, the configuration with the highest priority, i.e. with the smallest priority value, is activated and the associated behaviour is executed. In Figure 2.4, the configuration priority is indicated in brackets at the configuration name, e.g. configuration SteeringBased has priority 1. Influence rules describe how the values of the provided qualities are determined. Each influence rule consists of an influence guard and an influence function. The influence guard refines the configuration guard and defines a condition under which the respective influence function is applied. The influence function assigns the appropriate mode to each provided signal and calculates the mode attributes. In the V.YawCalculation module, each configuration only has one unconditional influence. For instance, the configuration SteeringBased assigns the quality of \( v_{yaw} \) to mode steering_based. Influence functions can also be represented graphically similar to configuration guards.

2.3 Related Work

From a general point of view, dynamic adaptation is a very diverse area of research including real-time systems [GSSR97], agent systems [MBS03] and component middleware [GNS04], just to name a few representatives. There are a number of approaches for modelling self-managed dynamic software architectures; for a survey, consult [BCDW04]. However, most of
these approaches focus on architectural adaptation instead of behavioural adaptation considered here.

In recent years, a number of frameworks for dynamic behaviour adaptation and reconfiguration have been developed, e.g. [RL05, COWL02]. Most of these frameworks are based on a dedicated control that contains all information on the adaptation behaviour and triggers adaptation of the components. Thus, even minor changes in one of the components require a complete re-design of the component controlling the adaptation behaviour. In contrast, the MARS modelling approach does not depend on a central control encoding the complete adaptation behaviour. This simplifies the design of adaptive systems and improves their maintainability.

Other approaches, e.g. [SKN01] or the MUSIC project\(^1\), have adopted ideas from variability modelling used to manage product lines. In [MFSS05], the concept of services as used in the service-oriented architecture approach is extended to services that can adapt dependent on a system’s context. The basic idea of these approaches is to shift the binding time of variation points from design time to run time. Thus, the systems autonomously determine all valid configurations at run time and select one according to a goal function. However, determining the next configuration in this way is computationally intensive and thus too slow for real-time systems. Furthermore, the complexity of computing the next configuration strongly limits model-based analysis, validation, and verification, which are indispensable in safety-critical areas like the automotive domain.

In contrast, MARS offers a constructive and explicit modelling technique for adaptation by predetermined reconfiguration. For a detailed description of MARS, the reader is referred to [TAFJ07, ASS08]. The modelling concepts separate adaptation from functionality by endowing data flow with qualities. The configuration behaviour of one module only depends on the quality transmitted with the input and output variables. The system designer can focus on the adaptation behaviour of single modules from which the adaptation behaviour of the system is determined. Integrating this constructive modelling technique with formal verification offers great potential in order to find more reasonable models of adaptation.

\(^1\) http://www.ist-music.eu/
A framework for integration of model-based development with formal verification has to be based on a solid formal foundation. A formal representation of the employed modelling concepts is necessary to give models a formal semantics. Furthermore, a specification logic for expressing properties over models is required. In this chapter, we first introduce synchronous adaptive systems (SAS) as a mathematical model for adaptive systems and show how MARS modelling concepts presented in Chapter 2 can be represented as SAS. Second, we introduce the temporal specification logic $L_{SAS}$ for properties of adaptive systems.

### 3.1 Synchronous Adaptive Systems

The MARS modelling approach [TAFJ07, ASS08] has been introduced for model-based development of adaptation by predetermined reconfiguration. Adaptation by predetermined reconfiguration requires that the different configurations a system can assume over time are fixed at design time. Further, it has to be determined under which circumstances a configuration is activated. This allows analysing the adaptation behaviour at design time which is required for safety-critical applications. To deal with the increased design complexity imposed by adaptation, MARS modelling concepts separate functionality from adaptation behavior and split a system into a set...
of time-synchronously operating modules to be designed independently. In order to facilitate a rigorous analysis of MARS models by verification techniques, their execution model (cf. Chapter 2) has to be captured formally.

Instead of embedding the intuitive semantics of MARS models into existing formalisms, such as standard state-transition systems [Kel76, MP95], our integration framework is based on synchronous adaptive systems (SAS) [SPH06b, ASSV07]. SAS are designed to represent MARS modelling concepts at a high level of abstraction. The structure of SAS models very closely resembles the structure of MARS models. The separation of functionality and adaptation inside MARS modules is maintained explicitly in SAS. Thus, system designers can be assured that their intuitive model is preserved in the formal representation. Furthermore, the high-level structure of SAS can be exploited for reducing model complexity with respect to verification. Encoding MARS models into an existing formalism would require to represent their high-level structure by more foundational concepts. This would cause that the separation of functionality and adaptation inside MARS modules is no longer captured directly. Hence, the formal model would be less understandable for system designers. Furthermore, the encoding would complicate verification complexity reductions because the model structure hidden in the encoding has to be retrieved by designated analyses before reductions can be carried out.

An SAS system is built from a set of interconnected modules representing the modular system structure of MARS models. Each module contains a set of functional configurations the module can adapt to. The activation of a configuration is determined by an adaptation aspect defined on top of the functional behaviour. The adaptation aspect formally captures the separation of functionality and adaptation behaviour in MARS models. SAS systems are open systems with non-deterministic input provided by an environment. Their semantics is based on standard state-transition system semantics. SAS modules operate time-synchronously as MARS modules in their intuitive execution model. Time-synchronous systems [Jan04, BCE+03] are efficiently analysable abstract models of systems with timing behaviour and traditionally used in modelling of hardware real-time systems. The formal notion of synchronous adaptive systems is derived from (labelled) state transition systems [Kel76, MP95] and their extension to modular transition systems [FMS97]. Other related formal system models are I/O automata [LT87] or communicating finite state machines [BZ83].
3.1. Synchronous Adaptive Systems

3.1.1 A Formal Model for Synchronous Adaptive Systems

Synchronous adaptive systems are a semantics-based formalism for capturing MARS modelling concepts at a high level of abstraction. Hence, no concrete syntax is introduced for defining SAS models. MARS models are directly translated into an abstract syntax representation of SAS. The abstract syntax is restricted to describe entities that are explicitly required for analysis of model properties. For representing configuration guards and influence functions of MARS models, we introduce an abstract syntax for expressions, constraints over expressions, and functions as lists of conditional and unconditional variable assignments. The abstract syntax is a minimal means to formally capture graphical representations occurring in MARS models, such as the configuration guard depicted in Figure 2.5, and facilitates a rigorous analysis of the adaptation behaviour expressed in MARS models. The abstract syntax can be also used to express functionality to be implemented in module configurations for analysis of functional behaviour.

Abstract Syntax of Synchronous Adaptive Systems

A synchronous adaptive system (SAS) is defined over a set of values $Val$ and a set of variables $Var$. An expression in SAS models can either be a variable, a value or a term using binary operations on expressions. An SAS constraint is built from atomic propositions that are binary predicates over expressions and Boolean negation and conjunction. Other Boolean operators can be expressed in terms of negation and conjunction.

Definition 3.1 (Syntax of SAS Expressions and Constraints). Let $x \in Var$ be a variable, $v \in Val$ a value and $b \in BinOp$ a binary operation symbol. The following grammar defines the abstract syntax of an SAS expression

$$expr \in Expr ::= x \mid v \mid b(expr_1, expr_2)$$

For an expression $expr \in Expr$, we define $Var(expr)$ as the set of variables contained in the expression inductively by:

- For $expr = x$, we have $Var(x) = \{x\}$, and for $expr = v$, we have $Var(expr) = \emptyset$.
- For $expr = b(expr_1, expr_2)$, we have $Var(expr) = Var(expr_1) \cup Var(expr_2)$.
Let \( r \in \text{Rel} \) be a binary predicate symbol and \( \text{expr}_1, \text{expr}_2 \in \text{Expr} \) two expressions over \( \text{Var} \) and \( \text{Val} \). The following grammar defines the abstract syntax of an SAS constraint.

\[
c \in \text{Constraint} ::= \text{atom} | \neg c_1 | c_1 \land c_2
\]

\[
\text{atom} \in \text{Atom} ::= r(\text{expr}_1, \text{expr}_2) | \text{true}
\]

For a constraint \( c \in \text{Constraint} \), we define \( \text{Var}(c) \) as the set of variables contained in the constraint inductively by:

- For \( c = r(\text{expr}_1, \text{expr}_2) \), we have \( \text{Var}(c) = \text{Var}(\text{expr}_1) \cup \text{Var}(\text{expr}_2) \)
- For \( c = \neg c_1 \), we have \( \text{Var}(c) = \text{Var}(c_1) \) and for \( c = c_1 \land c_2 \), we have \( \text{Var}(c) = \text{Var}(c_1) \cup \text{Var}(c_2) \).

The smallest construction element of a synchronous adaptive systems is a module. It contains a set of different predetermined configurations the module can adapt to. Adaptation is realised by an adaptation aspect inside the module separated from the functional configurations. There are two disjoint sets of variables in a module, functional and adaptive variables, that can be divided uniquely into input, output and local variables. Functional variables refer to the functional data computed by the configurations, whereas adaptive variables are used by the adaptation aspect in order to propagate information about the adaptation behaviour to other system modules. Input variables assume input values from the environment. Local and output variables are computed by the module in a state transition. Each configuration has a configuration guard determining under which circumstances the configuration can be activated. Before executing the actual functionality, the adaptation aspect evaluates the configuration guards and selects the configuration to use. Note that the configuration guards may only depend on adaptive variables in order to have a clean syntactic separation between functional and adaptation behaviour. Configuration guards are expressed by SAS constraints as defined in Definition 3.1. Figure 3.1 depicts the general structure of an SAS module. The separation of functional and adaptive module aspects in SAS modules captures the separation already present in the MARS modelling concepts and can later be used for model complexity reduction with respect to verification.

**Definition 3.2 (SAS Module).** An SAS module \( m \) is a tuple

\[
m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt})
\]

with
3.1. Synchronous Adaptive Systems

- \( \text{in} \subseteq \text{Var} \), a set of functional input variables, \( \text{out} \subseteq \text{Var} \), a set of disjoint functional output variables, \( \text{loc} \subseteq \text{Var} \), a set of disjoint functional local variables and \( \text{init} : \text{in} \cup \text{loc} \cup \text{out} \rightarrow \text{Val} \) their initial values

- \( \text{confs} = \{ (\text{guard}_j, \text{next}_\text{state}_j, \text{next}_\text{out}_j) \ | \ j = 1, \ldots, n \} \) the configurations of the module, where
  - \( \text{guard}_j \): an SAS constraint on \( \{ \text{adapt}_\text{in}, \text{adapt}_\text{loc} \} \) determining when configuration \( j \) is enabled with \( \text{adapt}_\text{in} \) and \( \text{adapt}_\text{loc} \) defined below
  - \( \text{next}_\text{state}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \rightarrow (\text{loc} \rightarrow \text{Val}) \) the next state function for configuration \( j \)
  - \( \text{next}_\text{out}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \rightarrow (\text{out} \rightarrow \text{Val}) \) the output function for configuration \( j \)

The adaptation aspect is defined as a tuple \( \text{adapt} = (\text{adapt}_\text{in}, \text{adapt}_\text{out}, \text{adapt}_\text{loc}, \text{adapt}_\text{init}, \text{adapt}_\text{next}_\text{state}, \text{adapt}_\text{next}_\text{out}) \) where

- \( \text{adapt}_\text{in} \subseteq \text{Var} \), a disjoint set of adaptation input variables, \( \text{adapt}_\text{out} \subseteq \text{Var} \), a disjoint set of adaptation output variables, \( \text{adapt}_\text{loc} \subseteq \text{Var} \), a disjoint set of adaptation local state variables and \( \text{adapt}_\text{init} : \text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \cup \text{adapt}_\text{out} \rightarrow \text{Val} \) their initial values
- \( \text{adapt}_\text{next}_\text{state} : (\text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \rightarrow \text{Val}) \rightarrow (\text{adapt}_\text{loc} \rightarrow \text{Val}) \) the adaptation next state function
- \( \text{adapt}_\text{next}_\text{out} : (\text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \rightarrow \text{Val}) \rightarrow (\text{adapt}_\text{out} \rightarrow \text{Val}) \) the adaptation output function

As an example of an SAS module, consider the module \( V_{\text{YawCalculation}} \) in the vehicle stability control system depicted in Figure 2.4. The functional input variables carry data values, such as the wheel speeds \( v_{\text{wheel}} \)
for each of the wheels, the speed of the car \( v_{\text{carRef}} \), the steering angle input from the driver \( \text{steering\_angle\_Input} \), the input from the yaw rate sensor \( \text{yaw\_rate} \) and the lateral acceleration \( ay \). For conveying the qualities attached to the inputs, we have an adaptive input variable with the suffix \( .\text{quality} \) for each input taking modes as values, e.g. \( ay.\text{quality} \) for the quality of the lateral acceleration. The only functional output variable is \( v_{\text{yaw}} \) which has an associated adaptive output variable \( v_{\text{yaw}.\text{quality}} \) capturing the behavioural variant used for computing the output by the respective mode. The module does not have a local state, thus the sets of local functional and adaptive variables remain empty. The functional output variable \( v_{\text{yaw}} \) is initialised with 0 and the associated adaptive variable \( v_{\text{yaw}.\text{quality}} \) with unavailable. The functional and adaptive input variables remain undefined. The configurations of the SAS module directly correspond to the configurations in Figure 2.4. The SAS module has six configurations \text{Measured}, \text{SteeringBased}, \text{FrontWheelBased}, \text{RearWheelBased}, \text{AyBased} \) and \text{Off} \) where the configuration index is the same as the configuration priority. For instance, configuration \text{SteeringBased} \) is assigned index 2. The configuration guard of the configuration \text{SteeringBased} \) is represented by \( \text{guard}_2 \) which can be described as SAS constraint\(^1\).

\[
\text{guard}_2 = \text{steering\_angle\_Input}.\text{quality} = \text{available} \land (v_{\text{carRef}}.\text{quality} = \text{wheel\_based} \lor v_{\text{carRef}}.\text{quality} = \text{ax\_based})
\]

The adaptive next output function \( \text{adapt\_next\_out} \) is used to represent the influences of the configurations assigning the appropriate mode to the adaptive output variable \( v_{\text{yaw}.\text{quality}} \). In Section 3.1.2, we show how to represent influences by SAS function descriptions. As the module has no local variables, the adaptive next state function \( \text{adapt\_next\_state} \) as well as the functional next state functions \( \text{next\_state}_j \) in all configurations remain undefined. The functional output functions \( \text{next\_out}_j \) of the configurations are used to describe how the functional output \( v_{\text{yaw}} \) is computed from the functional inputs. This is not modelled in the adaptation-specific view of MARS models. In Section 3.1.4, we focus on inserting functionality into SAS models.

An SAS system is composed from a set of modules with a system border containing system input and output variables, both functional and adaptive. The system border provides the system with a clear interface towards the environment. The system border and the system modules are interconnected with their input and output variables. Connected variables have

\(^1\) For simplicity, we use an infix notation of predicates and operators.
3.1. Synchronous Adaptive Systems

to be of the same kind, i.e. functional variables can only be connected to functional variables, adaptive variables only to adaptive variables. This maintains the separation between functionality and adaptation behaviour also on system level. Furthermore, no module or system variable may be unconnected in a valid system. A synchronous adaptive system is an open system with an environment providing non-deterministic input. We store the input coming from the environment in the input variables of the system and the modules. This simplifies reasoning about behaviour depending on inputs. For technical reasons, we assume that the variable names of all modules and the system border in a composed system are pairwise disjoint.

Definition 3.3 (SAS System). A synchronous adaptive system $SAS$ is a tuple

$$S = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)$$

where

- $M = \{m_1, \ldots, m_n\}$ is a set of SAS modules with $m_i = (in_i, out_i, loc_i, init_i, confs_i, adapt_i)$
- $input_a \subseteq \text{Var}$ are adaptation inputs and $input_d \subseteq \text{Var}$ disjoint functional inputs to the system
- $output_a \subseteq \text{Var}$ are disjoint adaptation outputs and $output_d \subseteq \text{Var}$ disjoint functional outputs from the system
- $conn_a$ is a total bijective function connecting module adaptation outputs to module adaptation inputs, system adaptation inputs to module adaptation outputs and module adaptation outputs to system adaptation outputs, $conn_a : (\bigcup_{j=1,\ldots,n} adapt\_out_j \cup input_a) \to (\bigcup_{j=1,\ldots,n} adapt\_in_j \cup output_a)$ where $conn_a(input_a) \subseteq \bigcup_{j=1,\ldots,n} adapt\_in_j$
- $conn_d$ is a total bijective function connecting module outputs to module inputs, system inputs to module inputs and module outputs to system outputs, $conn_d : (\bigcup_{j=1,\ldots,n} out_j \cup input_d) \to (\bigcup_{j=1,\ldots,n} in_j \cup output_d)$ where $conn_d(input_d) \subseteq \bigcup_{j=1,\ldots,n} in_j$.

Figure 3.2 shows an synchronous adaptive system composed from the modules $m_1, m_2, m_3$ and $m_4$. The solid arrows show functional connections, the dashed arrows adaptive connections. Note, that functional and adaptive connections do not have to be parallel such that data and adaptation flow can be completely decoupled.

The following definition introduces notations for variables of modules and systems. With $Var(m_i)$, we denote the variables of the module $m_i$ where $Var_a(m_i)$ are the adaptive variables and $Var_d(m_i)$ the functional variables.
The variables $Var(M)$ of a set of modules $M$ are the union of the module variables. The variables of a system SAS are denoted by $Var(SAS)$ containing the variables of the modules and the system border variables.

**Definition 3.4 (Module and System Variables).** For an SAS module $m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt})$ we define the following sets of variables:

- $Var(m) = \text{in} \cup \text{out} \cup \text{loc} \cup \text{adapt}_\text{in} \cup \text{adapt}_\text{out} \cup \text{adapt}_\text{loc}$
- $Var_a(m) = \text{adapt}_\text{in} \cup \text{adapt}_\text{out} \cup \text{adapt}_\text{loc}$
- $Var_d(m) = \text{in} \cup \text{out} \cup \text{loc}$.

For a set of modules $M = \{m_1, \ldots, m_n\}$, we define $Var(M) = \bigcup_{i=1,\ldots,n} Var(m_i)$. The sets $Var_a(M)$ and $Var_d(M)$ are defined accordingly.

For a system SAS with $M = \{m_1, \ldots, m_n\}$, the set of variables is defined as $Var(SAS) = Var(M) \cup \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d$. The sets $Var_a(SAS)$ and $Var_d(SAS)$ are defined accordingly.

**Semantics of Synchronous Adaptive Systems**

The semantics of SAS is defined in a two-layered approach. We start by defining the local semantics of single modules based on the interpretation of SAS expressions and constraints. The semantics uses the notion of state-transition systems, where states are valuations of variables and transitions transform a state into a successor state. From this, we define the global system semantics by synchronous composition of modules. Such time-synchronous systems [Jan04, BCE+03] are efficiently analysable abstract models of systems with timing behaviour.

An SAS expression is interpreted over a state $\sigma$ which is an assignment of variables to values $\sigma : Var \rightarrow Val$. A binary operation symbol $b \in BinOp$ is interpreted as a partial function $b_T : Val \times Val \rightarrow Val$, not necessarily
3.1. Synchronous Adaptive Systems

defined over all pairs of values. For instance, the addition over integers is only defined for integers and not for strings. Expressions must be well formed. Therefore, each variable \( x \in \text{Var} \) used in an expression must be declared as variable of a module or the system such that it can be evaluated in the respective state. Furthermore, in an expression the binary operation must be defined on the evaluation of the subexpressions which must be defined themselves. Constraints are similarly interpreted over SAS states.

A predicate symbol \( r \in \text{Rel} \) is interpreted by a partial Boolean function \( r_I : \text{Val} \times \text{Val} \rightarrow \{ \text{true}, \text{false} \} \). For an atomic proposition, the predicate must be defined over the evaluations of the two subexpressions in a state \( \sigma \).

Definition 3.5 (SAS Expressions and Constraints Semantics). The semantics of an SAS expression \( \text{expr} \in \text{Expr} \) is defined in a state \( \sigma : \text{Var} \rightarrow \text{Val} \) by

\[
\text{eval} (\text{expr})[\sigma] : \text{Expr} \times \Sigma \rightarrow \text{Val}
\]

where

- \( \text{for } x \in \text{Var(SAS)}, \text{eval}(x)[\sigma] = \sigma(x), \text{and for } v \in \text{Val}, \text{eval}(v)[\sigma] = v \)
- \( \text{eval}(b(\text{expr}_1, \text{expr}_2))[\sigma] = b_I(\text{eval}(\text{expr}_1)[\sigma], \text{eval}(\text{expr}_2)[\sigma]) \) if \( \text{eval}(\text{expr}_1)[\sigma], \text{eval}(\text{expr}_2)[\sigma] \) are defined.

The semantics of a constraint \( c \in \text{Constraint} \) in a state \( \sigma : \text{Var} \rightarrow \text{Val} \) is defined by \( \text{eval}(c)[\sigma] : \text{Constraint} \times \Sigma \rightarrow \{ \text{true}, \text{false} \} \) where

- \( \text{eval}(\text{true})[\sigma] = \text{true} \)
- \( \text{eval}(r(\text{expr}_1, \text{expr}_2))[\sigma] = r_I(\text{eval}(\sigma)[\text{expr}_1], \text{eval}(\sigma)[\text{expr}_2]) \) iff \( \text{eval}(\sigma)[\text{expr}_1], \text{eval}(\sigma)[\text{expr}_2] \) and \( r_I(\text{eval}(\sigma)[\text{expr}_1], \text{eval}(\sigma)[\text{expr}_2]) \) are defined
- \( \text{eval}(\neg c_1)[\sigma] = \neg \text{eval}(c_1)[\sigma] \)
- \( \text{eval}(c_1 \land c_2)[\sigma] = \text{eval}(c_1)[\sigma] \land \text{eval}(c_2)[\sigma] \)

A local state of a module is defined by a valuation of the module’s variables, i.e., functional input, output and local variables and their adaptive counterparts. A local state is initial if its functional and adaptation variables are set to their initial values. A local transition between two local states evolves in two stages: First, the adaptation aspect computes the new adaptation local state and the new adaptation output from the current adaptation input and the previous adaptation state. The adaptation aspect further selects the configuration with the smallest index that has a valid guard with respect to the current input and the previous functional and adaptive state. The system designer should ensure that the system has a shutdown configuration, that is usually called \( \text{Off} \) and enabled when no other configuration is. The selected configuration is used to compute the new local state and the
new output from the current functional input and the previous functional state. This means that if the configuration used in a transition is different than the one before, the current configuration uses the functional state produced by the previous configuration. If the system designer wants that each configuration starts from its initial state, when it becomes active, this has to be programmed by setting the variables a configuration solely uses to their initial state when another configuration is active.

In a local transition between two states \( s \) and \( s' \), the current input coming from the environment is stored in the input variables of the state \( s' \) and used to compute the values of the local and output variables of \( s' \). A different encoding for the input would be to parametrise the transitions between two states \( s \) and \( s' \) consisting of local and output variables, i.e. \( s = (\text{loc}, \text{out}) \), by \( s \xrightarrow{\text{in}} s' \). However, capturing the environment input in designated state variables allows a simpler reasoning about properties depending on inputs. In the following definition, we denote the restriction of a function \( f : A \rightarrow B \) to a subset \( M \subseteq A \) of this domain by \( f|_M : M \rightarrow B \).

**Definition 3.6 (Local States and Transitions).** A **local state** \( s \) of an SAS module \( m \) is defined as variable assignment

\[
\begin{align*}
  s : \text{in} \cup \text{out} \cup \text{loc} \cup \text{adapt\_in} \cup \text{adapt\_out} \cup \text{adapt\_loc} & \rightarrow \text{Val} \\
\end{align*}
\]

A local state \( s \) is called **initial** if

\[
\begin{align*}
  s|_{\text{in} \cup \text{loc} \cup \text{out}} &= \text{init} \\
  s|_{\text{adapt\_in} \cup \text{adapt\_loc} \cup \text{adapt\_out}} &= \text{adapt\_init}
\end{align*}
\]

A **local transition** between two local states \( s \) and \( s' \) is defined as \( s \rightarrow_{\text{loc}} s' \) iff the following conditions hold:

\[
\begin{align*}
  s'|_{\text{adapt\_loc}} &= \text{adapt\_next\_state}(s'|_{\text{adapt\_in}} \cup s|_{\text{adapt\_loc}}) \\
  s'|_{\text{adapt\_out}} &= \text{adapt\_next\_out}(s'|_{\text{adapt\_in}} \cup s|_{\text{adapt\_loc}}) \\
  \forall 0 < j < i. \text{eval}(\text{guard}_j)[s'|_{\text{adapt\_in}} \cup s|_{\text{adapt\_loc}}] &= \text{false} \\
  \text{eval}(\text{guard}_i)[s'|_{\text{adapt\_in}} \cup s|_{\text{adapt\_loc}}] &= \text{true} \\
  s'|_{\text{loc}} &= \text{next\_state}_i(s'|_{\text{in}} \cup s|_{\text{loc}}) \text{ and } s'|_{\text{out}} &= \text{next\_out}_i(s'|_{\text{in}} \cup s|_{\text{loc}})
\end{align*}
\]

The state of a system is the product of the local states of the contained modules together with an evaluation of the system inputs and outputs. A system state is initial if all states of the contained modules are initial and the system inputs and outputs are assigned to the initialisation of the connected module variables. A transition between two global states is
performed in three stages. First, each module reads its input either from another module’s output of the previous cycle or from the system inputs in the current cycle. Second, each module synchronously performs a local transition. Third, the modules directly connected to system outputs write their results to the output variables. Figure 3.3 shows a global transition between two global states $\sigma$ and $\sigma'$ where each of the local states $s_1$, $s_2$ and $s_3$ performs a synchronous local transition to $s'_1$, $s'_2$ and $s'_3$, respectively.

**Definition 3.7 (Global States and Transitions).** A **global state** $\sigma$ of a system SAS consists of the local states $\{s_1, \ldots, s_n\}$ of the contained modules, where $s_i$ is the state of $m_i \in M$, and an evaluation of the functional and adaptive system input and output variables

$$\sigma = s_1 \cup \ldots \cup s_n \cup ((input_a \cup input_d \cup output_a \cup output_d) \rightarrow Val)$$

A global state $\sigma$ with $\{s_1, \ldots, s_n\} \subseteq \sigma$ is called **initial** iff all local states $s_i$ for $i = 1, \ldots, n$, are initial and the system inputs and outputs are assigned to the initial values of the connected module variables, i.e. for $x \in input_a \cup output_a$, it holds that $\sigma(x) = v$ iff for $y \in Var_a(M)$ we have $(x, y) \in conn_a$ or $(y, x) \in conn_a$ and $adapt_{init}(y) = v$, and for $x \in input_d \cup output_d$, it holds that $\sigma(x) = v$ iff for $y \in Var_d(M)$ we have $(x, y) \in conn_d$ or $(y, x) \in conn_d$ and $init(y) = v$

Two states $\sigma$ and $\sigma'$ perform a **global transition**, $\sigma \rightarrow_{glob} \sigma'$, iff

- for all $x, y \in Var(SAS) \setminus (input_d \cup input_a)$ with $(x, y) \in conn_d$ or $(x, y) \in conn_a$, it holds that $\sigma'(y) = \sigma(x)$, for all $x \in input_a$ and $y \in Var(SAS)$ with $(x, y) \in conn_a$, it holds that $\sigma'(y) = \sigma'(x)$, and for all $x \in input_d$ and $y \in Var(SAS)$ with $(x, y) \in conn_d$, it holds that $\sigma'(y) = \sigma'(x)$
- for all $s_j \in \sigma$ and for all $s'_j \in \sigma'$, it holds that $s_j \rightarrow_{loc} s'_j$
• for all \( x \in \text{Var}(\text{SAS}) \) and \( y \in \text{output}_d \) with \( (x, y) \in \text{conn}_d \) it holds that \( \sigma'(y) = \sigma'(x) \), and for all \( x \in \text{Var}(\text{SAS}) \) and \( y \in \text{output}_a \) with \( (x, y) \in \text{conn}_a \) it holds that \( \sigma'(y) = \sigma'(x) \)

This definition of SAS system states and transitions induces a transition system as a different notation. An SAS system transition system consists of a set of global SAS states, a set of initial states and a global transition relation given by the global transitions.

**Definition 3.8 (SAS Transition System).** The transition system induced by an SAS system \( \text{SAS} \) is defined as \( T_{\text{SAS}} = (\Sigma_{\text{SAS}}, \text{Init}_{\text{SAS}}, \sim_{\text{glob}_{\text{SAS}}}) \) where

- \( \Sigma_{\text{SAS}} \) is the set of global states of SAS
- \( \text{Init}_{\text{SAS}} \) is the set of initial states of SAS
- \( \sim_{\text{glob}_{\text{SAS}}} \) is the global transition relation, where \( (\sigma, \sigma') \in \sim_{\text{glob}_{\text{SAS}}} \) iff \( \sigma \rightarrow_{\text{glob}_{\text{SAS}}} \sigma' \).

The semantics of an SAS system is defined by the set of possible computation paths of its transition system. In each step, system input is provided by the environment.

**Definition 3.9 (Paths).** A path of an SAS transition system \( T = (\Sigma, \text{Init}, \sim_{\text{glob}}) \) is defined as a sequence of global states \( \sigma_0 \sigma_1 \ldots \) where \( \sigma_0 \in \text{Init} \), and for all \( 0 \leq i \) we have \( \sigma_i \sim_{\text{glob}} \sigma_{i+1} \). With \( \pi_j \), we denote the suffix of path \( \pi = \sigma_0 \sigma_1 \sigma_2 \ldots \) starting in state \( \sigma_j \). The set \( \text{Paths}(T) = \{ \sigma_0 \sigma_1 \ldots \text{ a path of } T \} \) is the set of possible computation paths of \( T \) and defines the SAS semantics.

Additionally, we can define the notion of an SAS module transition system to reason about a module’s behaviour. Structurally, a module transition system is the same as a system transition system. Results we prove for a system transition system immediately hold for module transition systems as well.

**Definition 3.10 (Module Transition System).** The transition system induced by an SAS module \( m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{conf}, \text{adapt}) \) is defined by \( T_m = (S_m, \text{init}_m, \rightarrow_m) \) with

- \( S_m \) is the set of local states of the module \( m \), \( \{ s : \text{Var}(m) \rightarrow \text{Val} \} \)
- \( \text{init}_m \) the set of initial states of \( m \)
- \( \rightarrow_m \) the local transition relation, where \( (s, s') \in \rightarrow_m \) iff \( s \rightarrow_{\text{loc}_m} s' \)

A path of \( T_m \) is a sequence of states \( s_0 s_1 \ldots \) where \( s_0 \in \text{init}_m \), and for all \( i \geq 0 \), we have \( s_i \rightarrow_m s_{i+1} \).
In the example of the yaw rate module $V_{\text{YawCalculation}}$, a local module state is a valuation of the module's functional and adaptive input and output variables. A computation starts in the initial state, where the functional and adaptive input variables are undefined, the functional output $v_{\text{yaw}}$ is set to 0, and the adaptive output $v_{\text{yaw}.\text{quality}}$ is set to unavailable. In each state transition, the current input from the environment and from other modules for functional and adaptive input variables is read. Then, the adaptive next output function $\text{adapt}_\text{next}_\text{out}$ determines the value of the adaptive output variable $v_{\text{yaw}.\text{quality}}$. For instance, if $\text{guard}_1$, the guard of configuration $\text{Steering\_Based}$, evaluates to true and $\text{guard}_0$, the guard of configuration $\text{Measured}$, evaluates to false on the current input, $v_{\text{yaw}.\text{quality}}$ is assigned to $\text{steering\_based}$ according to the influence of the $\text{Steering\_Based}$ configuration. In this case, also the next output function of this configuration $\text{next}_\text{out}_1$ is used to compute the value for the functional output $v_{\text{yaw}}$.

### 3.1.2 A Language for Transitions

Up to now, SAS models are defined semantics-based without a syntax for expressing transitions. The ingredients of an SAS module are given as semantic constructs, e.g. the next state function is simply defined as a function $\text{next}_\text{state}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \rightarrow (\text{loc} \rightarrow \text{Val})$ mapping a valuation of input and local variables to a new valuation of the local variables. However, an abstract syntax for the SAS transition functions $\text{next}_\text{state}_j$, $\text{next}_\text{out}_j$, $\text{adapt}_\text{next}_\text{state}$ and $\text{adapt}_\text{next}_\text{state}$ allows capturing the specifications included in a MARS model explicitly and facilitates a detailed analysis of the modelled behaviour. Furthermore, syntactic transformations on SAS models can be used to reduce verification complexity.

The abstract syntax for SAS function descriptions comprises expressions over variables and values, constraints over variables and values and conditional and unconditional assignments of variables to expressions. An SAS transition function is defined by a list of assignments. This suffices for describing all relevant concepts of transition functions in SAS models. An iteration operator is not necessary since each variable can be assigned only once in a cycle because of the synchronous SAS semantics. SAS function descriptions are similar to guarded commands [Dij75] without the repetitive construct.

A simple SAS assignment consists of a variable $x \in \text{Var}$ to which an expression $\text{expr} \in \text{Expr}$ is assigned. An assignment is defined if the vari-
able is declared in the system and the evaluation of the expression in a state $\sigma$ is defined as well. A conditional assignment consists of the variable $x \in \text{Var}$ to be assigned and a list of conditionals $\text{cond} \in \text{Conditional}$. Each conditional $\text{cond}$ consists of a constraint $c$ and an expression $\text{expr}$. If the constraint evaluates to true $x$, the expression $\text{expr}$ is assigned, else the next conditional is evaluated. If no condition evaluates to true, the variable $x$ remains unchanged. A conditional assignment is defined, if the variable $x$ is declared and for each conditional, the evaluation of the constraint and the evaluation of the expression are defined. An SAS function is a list of simple and conditional assignments. In a well-formed function, each variable is assigned at most once and each assignment, either conditional or unconditional, is defined. A function $f \in \text{Func}$ is evaluated in a state $\sigma$ and returns a new state $\sigma'$ determined by the assignments in $f$.

**Definition 3.11 (SAS Functions).** Let $x \in \text{Var}$, $\text{expr} \in \text{Expr}$ and $c \in \text{Constraint}$. The syntax of an SAS function is defined by the following grammar.

\[
\begin{align*}
    f & \in \text{Func} \quad ::= \ [a_1, \ldots, a_n] \\
    a & \in \text{Assign} \quad ::= (x := [\text{cond}_1, \ldots, \text{cond}_n]) \mid (x := \text{expr}) \\
    \text{cond} & \in \text{Conditional} \quad ::= (c, \text{expr})
\end{align*}
\]

For a function description $f \in \text{Func}$, we define $\text{Var}(f)$ as the set of variables contained in $f$ inductively by:

- For $f = (x := \text{expr})$, we have $\text{Var}(f) = \{x\} \cup \text{Var(}\text{expr})$
- For $f = (x := [\text{cond}_1, \ldots, \text{cond}_n])$, we have $\text{Var}(f) = \{x\} \cup \bigcup_{i=1,\ldots,n} \text{Var(}\text{cond}_i)$ where $\text{Var(}\text{cond}_i) = \text{Var}(c) \cup \text{Var(}\text{expr})$
- For $f = [a_1, \ldots, a_n]$ we have $\text{Var}(f) = \bigcup_{i=1,\ldots,n} \text{Var}(a_i)$

The semantics of a function $f \in \text{Func}$ is defined by $\text{eval}(f)[\sigma] : \text{Func} \times \Sigma \rightarrow \Sigma$ where

- for a simple assignment, $\text{eval}(x := \text{expr})[\sigma] = \sigma'$, where for $x \in \text{Var(SAS)}$, $\sigma'(x) = \text{eval(}\text{expr})[\sigma]$, and for all $y \neq x$, $\sigma'(y) = \sigma(y)$, if $\text{eval(}\text{expr})[\sigma]$ is defined.
- for a conditional assignment, $\text{eval}(x := [(c, \text{expr}), \text{cond}_2, \ldots, \text{cond}_n])[\sigma] = \sigma'$, where for $x \in \text{Var(SAS)}$, $\sigma'(x) = \text{eval(}\text{expr})[\sigma]$ if $\text{eval}(c)[\sigma] = \text{true}$, and else $\sigma'(x) = \text{eval}(x := [\text{cond}_2, \ldots, \text{cond}_n])[\sigma]$, and for all $y \neq x$, $\sigma'(y) = \sigma(y)$ with $\text{eval}(c)[\sigma]$ and $\text{eval(}\text{expr})[\sigma]$ defined. For the empty list of conditionals $\text{eval}(x := [])[\sigma]$, it holds that $\sigma(x) = \sigma'(x)$.
3.1. Synchronous Adaptive Systems

• for a list of assignments, $\text{eval}([a_1,a_2,\ldots,a_n])[\sigma] = \sigma'$, if $a_1$ is an assignment to $x \in \text{Var}(\text{SAS})$, then $\sigma'(x) = \text{eval}(a_1)[\sigma]$ and 

$\sigma'_{\text{Var}(\text{SAS})\setminus\{x\}} = \text{eval}([a_2,\ldots,a_n])[\sigma]$ where $a_2,\ldots,a_n$ do not assign $x$ and 

$\text{eval}(a_1)[\sigma]$ is defined.

For the empty function list $[]$, it holds $\text{eval}([],)[\sigma] = \sigma$

In the vehicle stability control system, we use integers, real numbers and string constants as values in $\text{Val}$. As binary operations, we use the set $\text{BinOp} = \{+, -, \times, /, \text{max}, \text{min}\}$ with the standard interpretation over the integers and reals. For strings, binary operations are undefined. As predicate symbols in our examples, we use $\text{Rel} = \{=, \neq, \leq, \leq, \geq\}$ with their standard interpretation over the integers or reals. For string constants, only equality is defined and evaluates to true if applied to the same strings. In Figure 3.4, an example of an SAS function for an assignment occurring in the configuration $\text{SetBrakes}$ in the sensor module $\text{Brake}$ is depicted. If the input gas or brake force $\text{brakeGas} \_\text{Input}$ is greater than zero, this indicates that the car is to be accelerated and hence the brake value $\text{brake}$ is set to zero. If the input $\text{brakeGas} \_\text{Input}$ is smaller or equal to zero, this means that the car should be slowed down, and the absolute value of the input is assigned to absolute value of the input. Taking the absolute value of a negative number is encoded in SAS function descriptions by a subtraction from zero.

```latex
\begin{verbatim}
\text{brake} := [ ( \text{brakeGas} \_\text{Input} > 0 , 0) , \\
( \text{brakeGas} \_\text{Input} \leq 0, - (0 , \text{brakeGas} \_\text{Input}) ) ]
\end{verbatim}
```

Fig. 3.4. $\text{next \_out}$ Function of $\text{Brake}$ Module as SAS Function

An SAS module can now by syntactically described using the abstract syntax for constraints and functions. A syntactic SAS description is well-formed if the used syntactic constructs are well-formed.

**Definition 3.12 (SAS Syntactic Representation).** A syntactic SAS module description is a tuple

$\text{Descr}[m] = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{Descr}[\text{confs}], \text{Descr}[\text{adapt}])$

with

• $\text{in} \subseteq \text{Var}$, a set of functional input variables, $\text{out} \subseteq \text{Var}$, a disjoint set of functional output variables, $\text{loc} \subseteq \text{Var}$, a disjoint set of functional local variables and $\text{init} : \text{in} \cup \text{loc} \cup \text{out} \rightarrow \text{Val}$ a mapping to their initial values
Chapter 3. Specification of Synchronous Adaptive Systems

- $\text{Descr}[\text{confs}] = \{(\text{guard}_j, \text{Descr}[\text{next\_state}_j], \text{Descr}[\text{next\_out}_j]) \mid j = 1, ..., n\}$

the configuration descriptions of the module, where
- $\text{guard}_j \in \text{Constraint\ over\ } \{\text{adapt\_in}, \text{adapt\_loc}\}$ is an SAS constraint
- $\text{Descr}[\text{next\_state}_j] \in \text{Func}$ the next state function description where the variables in the conditions and expressions range over $(\text{in} \cup \text{loc})$ and the assigned variables range over $\text{loc}$
- $\text{Descr}[\text{next\_out}_j] \in \text{Func}$ the next out function description where the variables in the conditions and expressions range over $(\text{in} \cup \text{loc})$ and the assigned variables range over $\text{out}$

The adaptation aspect is described by a tuple $\text{Descr}[\text{adapt}] = (\text{adapt\_in}, \text{adapt\_out}, \text{adapt\_loc}, \text{adapt\_init}, \text{Descr}[\text{adapt\_next\_state}], \text{Descr}[\text{adapt\_next\_out}])$ where
- $\text{adapt\_in} \subseteq \text{Var}$, a disjoint set of adaptation input variables,
- $\text{adapt\_out} \subseteq \text{Var}$, a disjoint set of adaptation output variables,
- $\text{adapt\_loc} \subseteq \text{Var}$, a disjoint set of adaptation local state variables and
- $\text{adapt\_init} : \text{adapt\_in} \cup \text{adapt\_loc} \cup \text{adapt\_out} \rightarrow \text{Val}$ a mapping to their initial values
- $\text{Descr}[\text{adapt\_next\_state}] \in \text{Func}$ the adaptation next state function description where the variables in the conditions and expressions range over $\text{adapt\_in} \cup \text{adapt\_loc}$ and the assigned variables over $\text{adapt\_loc}$
- $\text{Descr}[\text{adapt\_next\_out}] \in \text{Func}$ the adaptation next out function description where the variables in the conditions and expressions range over $\text{adapt\_in} \cup \text{adapt\_loc}$ and the assigned variables over $\text{adapt\_out}$

The syntactic descriptions of SAS functions can be interpreted such that they implement the transition functions in an SAS module. An SAS module in Definition 3.3 can be defined by the syntactic SAS module description $\text{Descr}[m]$.

**Definition 3.13 (SAS Transition Functions).** For an SAS module $m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adaptation})$, the syntactic SAS module description

$\text{Descr}[m] = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{Descr}[\text{confs}], \text{Descr}[\text{adapt}])$

defines the following transition functions over states $s : \text{Var}(m) \rightarrow \text{Val}$:
- $\text{next\_state}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \rightarrow (\text{loc} \rightarrow \text{Val})$ the next state function for configuration $j$ where $\text{next\_state}_j(s) = \text{eval}(\text{Descr}[\text{next\_state}_j])[s]$.
- $\text{next\_out}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \rightarrow (\text{out} \rightarrow \text{Val})$ the output function for configuration $j$ where $\text{next\_out}_j(s) = \text{eval}(\text{Descr}[\text{next\_out}_j])[s]$. 
3.1. Synchronous Adaptive Systems

- \( \text{adapt\_next\_state} : (\text{adapt\_in} \cup \text{adapt\_loc} \to \text{Val}) \to (\text{adapt\_loc} \to \text{Val}) \) the adaptation next state function
  where \( \text{adapt\_next\_state}(s) = \text{eval}(\text{Descr}[\text{adapt\_next\_state}])[s] \).

- \( \text{adapt\_next\_state} :: (\text{adapt\_in} \cup \text{adapt\_loc} \to \text{Val}) \to (\text{adapt\_out} \to \text{Val}) \) the adaptation output function
  where \( \text{adapt\_next\_out}(s) = \text{eval}(\text{Descr}[\text{adapt\_next\_out}])[s] \).

The influences of the module \( V_{\text{YawCalculation}} \) can be represented as SAS function descriptions determining the adaptive next output function \( \text{adapt\_next\_out} \). It is a conditional assignment where in the conditionals the configuration guards serve as conditions and the values assigned in the influences as assigned expressions (cf. Figure 3.5). The order of the conditionals reflects the priority of the configurations.

3.1.3 Representing MARS Modelling Concepts

The MARS modelling concepts for adaptive systems introduced in Section 2.2 can be represented as synchronous adaptive systems (SAS). This provides MARS models with a formally exact semantics. SAS are designed to capture MARS modelling concepts at a high-level of abstraction. Thus, the gap between high-level intuitive modelling concepts and the formal representation is reduced.

A MARS system is represented as an SAS system where each MARS module is represented by an SAS module. All system modules in SAS models are assigned a distinct index which is also attached to all variables of the module in order to avoid name clashes. Ports of the MARS system are

```plaintext
v_yaw.quality := [(yaw_rate.quality = measured, sensor_based),
(steering_angle_Input.quality = available &
  (v_carRef.quality = wheel_based | v_carRef.quality = ax_based), steering_based),
(steering_angle_Input.quality = available &
  v_wheelFR.quality = available &
  v_wheelFL.quality = available),
(v_wheelRR.quality = available &
  v_wheelRL.quality = available, wheel_based),
(ay.quality = available &
  (v_carRef.quality = wheel_based |
  v_carRef.quality = ax_based), ay_based),
(true, unavailable)]
```

Fig. 3.5. \( \text{adapt\_next\_out} \) Function of \( V_{\text{YawCalculation}} \) Module as SAS Function Description
represented by SAS system variables and ports of MARS modules by module variables. For each port, the associated dative is mapped to a set of variables. One functional variable is used for conveying the data value of the port. A variable \textit{portname.quality} is used to store the quality attached to the data value using the different modes defined in the respective dative. If there are also mode attributes attached to a mode, additional variables \textit{portname.mode.modeattribute} are introduced. For each input port, we have a functional input variable, and for each output port, a functional output variable. If a port is declared as required, the variables for the quality part of the dative are defined as adaptive input variables. For a provided port, the adaptive variables generated from the dative become output variables.

The connections between ports in MARS models are captured by SAS system connections. Since in SAS models quality and data flow are completely decoupled, the quality flow is modelled independently of the dataflow using separate adaptive connections. For each connection in MARS, we add a functional connection in \textit{conn}_d to the SAS model that links the functional variables for the connected ports in the same direction as in the MARS model. For each adaptive variable, we add an adaptive connection to \textit{conn}_a. The connections link variables of the same kind, e.g. adaptive \textit{portname.quality} variables. The direction of these connections is such that the quality flows from provided to required ports.

A configuration of a MARS module is represented by an SAS configuration. The MARS configuration guard is mapped to an SAS configuration guard. MARS configuration guards include only constructs representable by syntactic SAS constraints. The priority of a MARS configuration is mapped to the SAS configuration index such that the configuration with the smallest priority value gets the smallest index. The influence function of a MARS configuration is mapped to the \textit{adapt.next.out} function of the SAS module’s adaptation aspect as the influence function only assigns adaptive variables. It can be expressed by a syntactic SAS function description as a list of conditional assignments with the configuration guards as conditions (cf. Figure 3.5). As an example for a SAS module representation of a MARS module, consider the SAS module \textit{V_YawCalculation} described in Sections 3.1.1 and 3.1.2.

3.1.4 Functional Behaviour

MARS models do not explicitly specify the functionality implemented by a configuration. In the MARS development process, the functionality is sup-
posed to be inserted after the successful verification and validation of the adaptation behaviour. By translating a MARS model to an SAS system, the transition functions next_state and next_out of SAS configurations remain unspecified.

In order to reason about functional behaviour or the combination of functionality and adaptation behaviour, functionality implemented in a configuration has to be specified by the system designer at least to some extend in an SAS model, after it has been generated from a MARS model. For verification, the functionality of a configuration is assumed to satisfy this specification. In the implementation, it has then to be ensured that the specifications are actually fulfilled in order to transfer the established properties. Functional specifications can be inserted on SAS model level or given as annotations in MARS models and translated automatically to an internal SAS representation.

The functionality of SAS configurations can be described in two ways. First, the next_state and next_out functions of the configurations can be specified by SAS function descriptions and interpreted as defined in Definition 3.11. Specifying functionality by SAS functions is only possible if the functionality can be expressed with the operations included in SAS function descriptions. As an example, consider the functionality of the configuration SetBrakes in the module Brakes in the vehicle stability control system as depicted in Figure 3.4. If the input brakeGas_Input is greater than zero, the output brake is assigned to zero, else the absolute value of the input is taken. This functionality is simple enough to be represented directly as an SAS function description.

Second, functional behaviour can be described by logical pre-/post-condition specifications over functional inputs and outputs of a module. In our framework, we use representations of the theorem prover Isabelle/HOL [NPW02]. Semantically, a pre-/post-condition specification induces a relation on the variables in a transition. Logical specifications have high expressiveness since, for example, arbitrary arithmetical operations over infinite data domains can be used. The trade-off, however, is

\[
\begin{align*}
( & (\text{steering_angle}_\text{Input} \leq \text{max}\_\text{Angle}(v,\text{carRef}) \implies \\
& \quad \text{delimited}\_\text{steering}\_\text{angle} = \text{steering}_\text{angle}_\text{Input}) ) \\
\land & (\text{steering}\_\text{angle}_\text{Input} > \text{max}\_\text{Angle}(v,\text{carRef}) \implies \\
& \quad (\text{delimited}\_\text{steering}\_\text{angle} < \text{max}\_\text{Angle}(v,\text{carRef}) \\
& \quad \land \text{delimited}\_\text{steering}\_\text{angle} < \text{steering}\_\text{angle}_\text{Input} )))
\end{align*}
\]

Fig. 3.6. Functionality of Configuration Controlled as Pre-/Post-Condition Specification
that verification is limited to theorem provers that are able to handle the employed logic (cf. Section 4.2). As an example of a pre-/post-condition specification, consider the functionality of the steering delimiter module in configuration Controlled depicted in Figure 3.6. If the input for the steering angle steering_angle_Input exceeds the maximal possible steering angle \( \text{max.Angle} \) at the current speed of the car \( v_{\text{carRef}} \), the output value delimited_steering_angle is set to a value smaller than the maximal possible steering angle and also smaller than the input steering angle. Otherwise, the input is simply transferred to the output. The maximal possible steering angle at the current speed of the car \( \text{max.Angle} (v_{\text{carRef}}) \) is modelled as a function assigning a maximal possible steering angle to each speed value.

### 3.2 Properties of Synchronous Adaptive Systems

Properties of adaptive systems [ASSV07, ASS08] can be distinguished whether they concern the adaptation behaviour, the functional behaviour or combined aspects. Furthermore, system properties can be classified into generic properties desirable for all systems and properties considering particular characteristics of an application. Figure 3.7 shows the different dimensions of properties over adaptive embedded systems.

![Classification of Properties over Adaptive Systems](image)

**Fig. 3.7.** Classification of Properties over Adaptive Systems

Generic functional properties refer to absence of functional runtime errors, such as division by zero or overflows. Generic adaptive properties deal
with the correctness of the adaptation process. Although the generic adaptive properties in the following refer to single modules, their validity for all modules guarantees correctness of the overall adaptation behaviour of a system. First, in the adaptation process no configuration of a module should be redundant, i.e. all configurations can be reached during a system run. Second, in all cycles, one of the predefined configurations of a module is actually taken. Otherwise, undefined and inconsistent situations occur. A third property is that no module deadlocks in the shutdown configuration $\text{Off}$ where output remains unavailable. This is important for the responsiveness of a system. A fourth important property of the adaptation behaviour is that no configuration of a module in a system is always only transient, i.e. it can be active for an arbitrary amount of time. A crucial property for the adaptation behaviour is further the stabilisation of the adaptation process. Since adaptation in the considered class of systems is not regulated by a central authority, adaptation in one module may trigger adaptations of other modules. If this process of reconfigurations continues infinitely, the system may become unstable and inconsistent. Hence, we want to be sure that the configurations used in each module stabilise if the inputs of a system remain stable.

Application-specific properties are first concerned with the adaptation behaviour resulting in the concrete assembly of different system modules. It can be that adaptation in one module to one particular configuration leads to a particular configuration in another module after a defined number of cycles or excludes activation of a configuration. In the vehicle stability control system, an application-specific adaptive property is for example that none of the core modules is in the shutdown configuration $\text{Off}$ if the basic sensors and actuators are available. Second, application-specific properties address functional and functional-adaptive properties of a system. These include the correctness of functionality implemented in one configuration or the correctness of functionality influenced by adaptation. In the vehicle stability control system, an application-specific property including functional behaviour is, for example, a safety property of the steering angle delimiter module saying that the output delimited steering angle is always smaller or equal to the input steering angle.

In order to verify properties of adaptive systems given as SAS models, we need means to formally specify properties over computation paths of SAS systems. In this section, we introduce the specification logic $\mathcal{L}_{\text{SAS}}$ and its fragments to capture properties of SAS formally. Furthermore, we show
how properties of the adaptive vehicle stability control system are formally expressed in this specification logic.

3.2.1 Formal Specification Logic

We define $L_{SAS}$ as the language to express properties of a synchronous adaptive system over a set of variables $\text{Var}$ and a set of values $\text{Val}$. Since the semantics of SAS is given by a set of computation paths, we need constructs to reason about single SAS states as well as about sequences of states on a computation path. Our logic $L_{SAS}$ is a variant of the computational tree logic $\text{CTL}^*$ [Eme90].

The atomic propositions of $L_{SAS}$ are atomic constraints as in Definition 3.1. Atomic constraints are binary predicates over expressions evaluated in an SAS state. For the vehicle stability control system example, we use the set of predicates $\text{Rel} = \{=, \neq, \leq, \leq, \geq, \geq\}$ with their standard interpretation over the integers or reals and equality on string constants in atomic propositions. Boolean connectives can be used for combining atomic propositions over a state. Additionally, $L_{SAS}$ provides temporal operators for speaking about the properties of a computation path. The formula $X \varphi$ ("next") denotes that a property holds in the next state of the computation. $G \varphi$ ("globally") says that $\varphi$ holds on all states of a computation path. $F \varphi$ ("finally") expresses that there exists a state on a computation path such that $\varphi$ holds.

The formula $\varphi U \psi$ ("until") describes a computation path where in each state $\varphi$ is true until there is a state in which $\psi$ becomes true. Additionally, there are path quantifiers $A$ and $E$ where $A \varphi$ requires that $\varphi$ holds on all computation paths that can start in a state and $E \varphi$ only that there exists a computation path such that $\varphi$ holds. Formulae starting with the path quantifier $A$ are also called universal formulae, formulae starting with $E$ are called existential. The following definition gives the grammar of $L_{SAS}$. It distinguishes state and path formulae which are evaluated on states and computation paths, respectively.

**Definition 3.14 (Syntax of Logic $L_{SAS}$).** Let $\text{expr}_1, \text{expr}_2 \in \text{Expr}$ be two SAS Expressions over $\text{Var}$ and $\text{Val}$ as in Definition 3.1 and $\text{Rel}$ a set of binary predicate symbols with $r \in \text{Rel}$.

- $A\text{toms } a ::= r(\text{expr}_1, \text{expr}_2)$
- $\text{State Formula } \varphi ::= \text{true} \mid a \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid E \psi$
- $\text{Path Formula } \psi ::= \varphi \mid \neg \psi \mid \psi_1 \land \psi_2 \mid X \psi \mid \psi_1 U \psi_2$
We define the other temporal operators with the usual abbreviations as follows:

\[
\begin{align*}
F \varphi & \equiv \text{true} \\
G \varphi & \equiv \neg F \neg \varphi \\
A \varphi & \equiv \neg E \neg \varphi
\end{align*}
\]

The set of variables contained in an \( \mathcal{L}_{\text{SAS}} \) formula is inductively defined on the structure of the formula and collects all variables from expressions used in the atomic propositions.

**Definition 3.15 (Variables of \( \mathcal{L}_{\text{SAS}} \) Formulae).** For an \( \mathcal{L}_{\text{SAS}} \) formula \( \varphi \), we define the set of variables \( \text{Var}(\varphi) \) as the set of all variables occurring in the property \( \varphi \) inductively on the formula structure by:

- for \( \varphi = r(\text{expr}_1, \text{expr}_2) \), \( \text{Var}(\varphi) = \text{Var}(\text{expr}_1) \cup \text{Var}(\text{expr}_2) \)
- \( \text{Var}(\neg \varphi) = \text{Var}(\varphi) \)
- \( \text{Var}(\varphi_1 \land \varphi_2) = \text{Var}(\varphi_1) \cup \text{Var}(\varphi_2) \)
- \( \text{Var}(X \varphi) = \text{Var}(\varphi) \)
- \( \text{Var}(\varphi_1 U \varphi_2) = \text{Var}(\varphi_1) \cup \text{Var}(\varphi_2) \)
- \( \text{Var}(A \varphi) = \text{Var}(\varphi) \) and \( \text{Var}(E \varphi) = \text{Var}(\varphi) \)

After defining the syntax of an \( \mathcal{L}_{\text{SAS}} \) formulae, we define its semantics over the computation paths of an SAS model. State formulae are evaluated in a SAS state and the computation paths starting in this state. Path formulae are evaluated over paths of an SAS. An atomic proposition is true in a state if the expressions are defined in this state and if the interpretation of the predicate symbol over their evaluations is defined and yields true. Boolean connectives and temporal operators as well as the path quantifiers are defined as in CTL* [Eme90]. Note, that each state formula is also a path formulae holding in the first state of a path.

**Definition 3.16 (Semantics of \( \mathcal{L}_{\text{SAS}} \) Formulae).** Let \( T \) be an SAS transition system and \( r_T : \text{Val} \times \text{Val} \rightarrow \{\text{true}, \text{false}\} \) for each \( r \in \text{Rel} \) the interpretation of the predicate symbol \( r \). For a state formula \( \varphi \), \( (T, \sigma) \models \varphi \) is inductively defined on the formula structure of \( \varphi \):

- \( (T, \sigma) \models \text{true} \) always
- \( (T, \sigma) \models r(\text{expr}_1, \text{expr}_2) \) iff \( r_T(\text{eval}(\text{expr}_1)[\sigma], \text{eval}(\text{expr}_2)[\sigma]) = \text{true} \) and \( \text{eval}(\text{expr}_1)[\sigma] \) and \( \text{eval}(\text{expr}_2)[\sigma] \) and \( r_T(\text{eval}(\text{expr}_1)[\sigma], \text{eval}(\text{expr}_2)[\sigma]) \) are defined in \( \sigma \)
- \( (T, \sigma) \models \neg \varphi \) iff \( (T, \sigma) \not\models \varphi \)
- \( (T, \sigma) \models \varphi_1 \land \varphi_2 \) iff \( (T, \sigma) \models \varphi_1 \) and \( (T, \sigma) \models \varphi_2 \)
• \((T, \sigma) \models E\psi\) iff there exists a path \(\pi = \sigma\sigma_1\sigma_2\ldots\) in \(T\) such that \((T, \pi) \models \psi\).

For a path formula \(\psi\), \((T, \pi) \models \psi\) is inductively defined on the formula structure of \(\psi\):

• For a state formula \(\varphi\), \((T, \pi) \models \varphi\) iff \(\pi = \sigma\sigma_1\sigma_2\ldots\) and \((T, \sigma) \models \varphi\)
• \((T, \pi) \models \neg\psi\) iff \((T, \pi) \not\models \psi\)
• \((T, \pi) \models \psi_1 \land \psi_2\) iff \((T, \pi) \models \psi_1\) and \((T, \pi) \models \psi_2\)
• \((T, \pi) \models X\psi\) iff there exists \(\pi_1\) such that \((T, \pi_1) \models \psi\)
• \((T, \pi) \models \psi_1 \lor \psi_2\) iff there exists \(k \geq 0\) such that \((T, \pi_k) \models \psi_2\)

The validity of a universal state formula \(A\varphi\) can be defined using the syntactic abbreviation (see above), but it can also be defined directly by

\[(T, \sigma) \models A\psi\) iff for all paths \(\pi = \sigma\sigma_1\sigma_2\ldots\) it holds that \((T, \pi) \models \psi\)

For a state formula \(\varphi\), \(T \models \varphi\) holds if and only if for all \(\sigma_0 \in \text{Init}\) we have \((T, \sigma_0) \models \varphi\). For a path formula \(\psi\), \(T \models \psi\) holds if and only if \((T, \pi_0) \models \varphi\) for all \(\pi_0 \in \text{Paths}(T)\) starting in initial states \(\sigma_0 \in \text{Init}\).

Negation, Boolean conjunction over state and path formulae, the existential path quantifier \(E\) and the temporal operators \(X\) and \(U\) are sufficient to express every \(L_{SAS}\) formula. Furthermore, every formula of \(L_{SAS}\) can be expressed in positive normal form where negation is only applied to atomic propositions. Figure 3.8 shows equivalences valid in \(L_{SAS}\) (for same equivalences in \(CTL^*\) cf. [CGP99]) that allow transforming an arbitrary \(L_{SAS}\) formula to negation normal form. In order to avoid loss of expressiveness, we need Boolean conjunction and disjunction over state and path formulae and the release operator \(R\). The release operator \(R\) is the dual of the until operator \(U\) and can be defined as \(\varphi_1 R \varphi_2 \equiv \neg(\neg\varphi_1 U \neg\varphi_2)\). It can also be defined directly over paths by

\[(T, \pi) \models \varphi_1 R \varphi_2\) iff either \(\forall i. (T, \pi_i) \models \varphi_2\)

or \(\exists n. (T, \pi_n) \models \varphi_1\) and \(\forall i < n (T, \pi_i) \not\models \varphi_2\)

For technical reasons, e.g. for establishing soundness of compositional reasoning strategies, it is sometimes more convenient to restrict the variables

\[
\neg X \varphi \equiv X \neg \varphi \\
\neg(\varphi_1 U \varphi_2) \equiv (\neg \varphi_2 \land \neg \varphi_1) R \neg \varphi_2 \\
\neg(\varphi_1 R \varphi_2) \equiv \neg \varphi_1 U (\neg \varphi_2 \land \neg \varphi_1) \\
\neg(\varphi_1 \land \varphi_2) \equiv \neg \varphi_1 \lor \neg \varphi_2 \\
\neg(\varphi_1 \lor \varphi_2) \equiv \neg \varphi_1 \land \neg \varphi_2
\]

Fig. 3.8. Equivalences in \(L_{SAS}\)
occurring in a property to the variables of the modules included in a system. In this direction, if a property contains system variables, the property can be rephrased as an equivalent property over the module variables. The following lemma provides us with the result that any property \( \varphi \) over a system \( SAS \) containing system variables can be transformed to an equivalent property \( \varphi' \) where the border variables are replaced by the connected module variables. The restriction to module variables is merely for ease of presentation.

**Lemma 3.17 (Equivalent Property without System Variables).** Let \( SAS \) be an SAS system

\[
SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d)
\]

with the set of modules \( M \) and \( \varphi \) a \( \mathcal{L}_{SAS} \) property over \( SAS \). Let further \( \varphi' = \varphi[v/w] \) be a \( \mathcal{L}_{SAS} \) property that is constructed from \( \varphi \) by replacing all occurrences of the variable \( v \) by the variable \( w \) if for \( v \in \text{input}_a \cup \text{output}_a \), there exists \( w \in \text{Var}(M) \) such that \( \text{conn}_a(v, w) \) or \( \text{conn}_a(w, v) \), and for \( v \in \text{input}_d \cup \text{output}_d \), there exists \( w \in \text{Var}(M) \) such that \( \text{conn}_d(v, w) \) or \( \text{conn}_d(w, v) \). Then it holds that

\[
T_{SAS} \models \varphi \iff T_{SAS} \models \varphi'
\]

**Proof.** By induction on the structure of \( \varphi \).

- **Base Case:** Let \( a \) be atomic. Assume \( \text{Var}(a) \in \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d \). By definition of \( \models \), we have that for each \( \sigma_0 \in \text{Init} \), it holds \( \sigma_0 \models a \). For each \( v \in \text{Var}(a) \) and any state \( \sigma \in \Sigma \), we have that \( \sigma(v) = \sigma(w) \) for \( w \in \text{Var}(M) \) with \( \text{conn}_a(v, w) \) or \( \text{conn}_a(w, v) \) or \( \text{conn}_d(v, w) \) or \( \text{conn}_d(w, v) \). Hence, \( \sigma_0 \models a' \) and also the converse.

- **Induction Step:** For each Boolean or temporal connective, the proposition can be easily shown using the induction hypothesis. \( \square \)

Properties of \( \mathcal{L}_{SAS} \) can be classified according to their structure into safety and liveness properties [AS85]. A safety property \( G \varphi \) denotes that \( \varphi \) invariantly holds. A liveness property \( F \varphi \) expresses that eventually \( \varphi \) will become true. Every property in \( \mathcal{L}_{SAS} \) can be expressed as a conjunction of a safety and a liveness property. Besides safety and liveness, [MP95] defines fairness properties \( GF \varphi \) denoting that the property \( \varphi \) holds infinitely often on a path and persistence properties \( FG \varphi \) expressing that there is a point in time from that point on \( \varphi \) invariantly holds.
Furthermore, we can define certain fragments of $\mathcal{L}_{SAS}$ with restricted usage of the temporal operators and path quantifiers. These fragments are used in the following chapters. The fragment of $\mathcal{L}_{SAS}$ where only the universal path quantifier $A$ is used is called the universal fragment $AL_{SAS}$. In order to avoid the implicit occurrence of the existential path quantifier, we assume that formulae of $AL_{SAS}$ are given in positive normal form, i.e. negations are only applied to atomic propositions. Hence, to avoid loss of expressiveness, $AL_{SAS}$ uses disjunction and conjunction over state and path formulae, the temporal operators $X$, $U$ and $R$ and the universal path quantifier $A$. The fragment $AL_{SAS}$ corresponds to ACTL* [CGP99].

The linear time fragment $LL_{SAS}$ only contains formulae of the form $A\varphi$ where $\varphi$ does not contain any path quantifiers. This fragment describes properties over all linear computation paths and corresponds to the temporal logic LTL [Pnu77]. Furthermore, we can define the branching time fragment $BL_{SAS}$ of $L_{SAS}$ which contains only formulae including combinations of a temporal operator with a path quantifier, i.e. $AX$, $AU$, $EF$ and so on. $BL_{SAS}$ corresponds to the branching time logic CTL [Eme90] and can further be restricted only to contain combinations with the universal path quantifier to the universal branching time fragment $ABL_{SAS}$.

Linear and branching time logics, and hence also $LL_{SAS}$ and $BL_{SAS}$, have different expressiveness [CGP99]. There are properties which are expressible in linear time logic, but not in branching time, such as $AGF\varphi$. There are also properties expressible in branching time, but not in linear time, such as $AGEF\varphi$. $L_{SAS}$ as a variant of the universal branching time logic CTL* also contains properties not expressible in either fragment, such as the disjunction of these two properties, $AGF\varphi \lor AGEF\varphi$.

### 3.2.2 Specifying Properties of Adaptive Systems

In order to reason about the active configuration of an SAS module in each cycle, a special adaptive variable $useconf$ is added to the set of adaptive variables $Var_a(m)$ for each module $m$. In an initial state, the variable $useconf$ is set to $Off$ which is the default shutdown configuration of a module. In a transition $\rightsquigarrow$ from state $s$ to state $s'$, $useconf$ is assigned as follows:

$$s'(useconf) = k \text{ iff } \begin{align*}
\forall 0 < j < k, \ \text{eval}(\text{guard}_j)[s'|_{\text{adapt.in}} \cup s|_{\text{adapt.loc}}] &= \text{false} \\
\text{eval}(\text{guard}_k)[s'|_{\text{adapt.in}} \cup s|_{\text{adapt.loc}}] &= \text{true}
\end{align*}$$

As we have seen, properties of adaptive systems can be classified whether they refer to functional, adaptive or combined aspects of a system. Using
3.2. Properties of Synchronous Adaptive Systems

$L_{SAS}$, this notion can be formally captured. A functional property over a system only contains functional variables, an adaptive property contains only adaptive variables, the same applies to properties over modules.

**Definition 3.18 (Adaptive and Functional Properties).** Let $SAS$ be an SAS model $SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)$ and $\varphi$ a $L_{SAS}$ property.

- $\varphi$ is a purely functional property of $SAS$ if $\operatorname{Var}(\varphi) \subseteq \operatorname{Var}_d(SAS)$.
- $\varphi$ is a purely adaptive property of $SAS$ if $\operatorname{Var}(\varphi) \subseteq \operatorname{Var}_a(SAS) \cup \{\text{useconf}_i\}$ for all modules $m_i$ in $SAS$.
- For a module $m \in M$, $\varphi$ is a functional property of $m$ if $\operatorname{Var}(\varphi) \subseteq \operatorname{Var}_d(m)$, and $\varphi$ an adaptive property of $m$ if $\operatorname{Var}(\varphi) \subseteq \operatorname{Var}_a(m) \cup \{\text{useconf}\}$

Most of the generic adaptive properties can be expressed in the branching time fragment $BL_{SAS}$, which allows employing standard model checking techniques for verification. Figure 3.9 shows the temporal logic specification of the generic adaptive properties mentioned earlier for a module $m$. Establishing these properties for all modules in the system provides guarantees for the overall adaptation behaviour. Property 1 specifies that $m$ does not stick in its shutdown configuration $Off$. Property 2 states that the module can reach all configurations at all times. This guarantees that no configuration is redundant. Moreover, module $m$ should always be able to activate one of the defined configurations such that no inconsistent states can be reached (Property 3). Property 4 requires that no configuration of $m$ is always only transient, i.e. that it can be active for an arbitrary amount of time.

<table>
<thead>
<tr>
<th>Property 1 (liveness): $AG(\text{useconf} = \text{Off} \rightarrow EF \text{useconf} \neq \text{Off})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 2 (reachability): $AG(\bigwedge_{j=1}^{n} \text{EF useconf} = \text{config}_j)$</td>
</tr>
<tr>
<td>Property 3 (safety): $AG(\bigvee_{j=1}^{n} \text{useconf} = \text{config}_j)$</td>
</tr>
<tr>
<td>Property 4 (persistence): $\bigwedge_{j=1}^{n} \text{EFEG useconf} = \text{config}_j$</td>
</tr>
</tbody>
</table>

**Fig. 3.9.** Generic Adaptive Properties for Module $m$

Stability of the adaptation process is an important property for synchronous adaptive systems as adaptation is not controlled by a central authority. Intuitively, stability means that if the inputs to a system do not change, there will be a point in time from which on also the outputs remain stable. In SAS systems, a reconfiguration of a module can trigger reconfigurations of other modules such that there can be a chain reaction.
of reconfigurations in the system. If this sequence does not terminate, the
system is unstable. Stability cannot be expressed in $BL_{SAS}$, but in $LL_{SAS}$
which allows checking stability by standard linear time logic model checking.
Suppose that $\varphi_{in}$ holds iff the inputs of an SAS do not change for one
unit of time. Moreover, let $\varphi_{so}$ hold iff the state variables and the outputs
do not change for one time unit. Then, an SAS is stable iff the LTL formula
$\Phi$ holds:

$$\Phi \equiv AG (G\varphi_{in} \to FG\varphi_{so})$$

For the vehicle stability control system, we can formulate application-
specific properties of the adaptation behaviour. An important property of
the adaptation behaviour is that the three core modules correctly imple-
ment a safe fallback layer. This means that even if the full functionality
cannot be provided, the controller models do not assume the shutdown con-
figuration $Off$ if the basic sensors and actuators are available. Figure 3.10
shows the formalisation of those properties in the linear time fragment of
$L_{SAS}$. The variable names refer to the input and output variables of the
controller modules (cf. Figure 2.2). The variables with suffix $quality$ are
adaptive variables and convey the qualities attached to the data values.

As an example of an application-specific property concerning adaptation
as well as functional behaviour, consider the following safety property of
the steering angle delimiter module in the vehicle stability control system
expressible in the linear time fragment of $L_{SAS}$. The property says that
if the module is not in its shutdown configuration $Off$, the output of the
module $delimited\_steering\_angle$ is always smaller or equal to the input the
module receives from the driver $steering\_angle\_Input$. This implies that the

---

| Traction Control: | $AG ((\text{gas.quality} = \text{available})$
| | $\land (\text{delimited.gas.quality} = \text{available})$
| | $\to \text{useconf.TractionControl} \neq \text{Off})$ |
| Steering Angle Delimiter: | $AG ((\text{steering}\_\text{angle}\_\text{Input.quality} = \text{available})$
| | $\land (\text{delimited.steering}\_\text{angle}\_\text{quality} = \text{available})$
| | $\to \text{useconf.SteeringAngleDelimiter} \neq \text{Off})$ |
| Yaw Rate Correction: | $AG ((\text{brake}\_\text{quality} = \text{available})$
| | $\land (\text{corrected.left.brake}\_\text{quality} = \text{available})$
| | $\land (\text{corrected.right.brake}\_\text{quality} = \text{available})$
| | $\land (\text{corrected.rear.brake}\_\text{quality} = \text{available})$
| | $\to \text{useconf.YawRateCorrector} \neq \text{Off})$ |

---

Fig. 3.10. Application Specific Adaptive Properties of Vehicle Stability Control System
steering angle delimiter module never produces a value for the steering angle that is larger than desired which would pose a severe safety-risk.

\[
AG \ (useconf_{\text{SteeringAngleDelimiter}} \neq \text{Off} \rightarrow (\text{delimited.steering.angle} \leq \text{steering.angle.Input}))
\]

Another application-specific safety property of the functionality in the vehicle stability control system is that the output of the brake and gas modules is mutually exclusive. As the brake and gas input from the driver is given by only one combined input, this input has to be split into brake force and gas value (cf. Figure 7.11 in Section 7.2.1). For correct operation of the system, if both modules are not in their shutdown configuration \textit{Off}, we require that if the output of the brake module \textit{brake} is greater than zero, the output of the gas module \textit{gas} has to be equal to zero, and if the gas output is greater than zero, the brake output has to be zero. This property is expressed in the universal fragment of \(\mathcal{L}_{\text{SAS}}\) by

\[
AG \ (\text{useconf}_{\text{Brake}} \neq \text{Off} \land \text{useconf}_{\text{Gas}} \neq \text{Off}) \rightarrow (\text{brake} > 0 \rightarrow \text{gas} = 0) \land (\text{gas} > 0 \rightarrow \text{brake} = 0))
\]

3.3 Related Work

Recently, formal modelling of adaptive systems with respect to verification has become an active area of research in order to find reasonable and safe adaptation behaviour models. In [Str05], an architecture model is presented for systems reconfigurable in case of hardware or software failures. It contains a central authority controlling and managing the reconfiguration of system modules. The architecture is specified in the PVS theorem proving environment such that generic safety properties of the reconfiguration architecture can be verified. However, the notion of adaptation in this work differs from our approach as adaptation is controlled by a central authority. Furthermore, verification of application-specific properties of the system modules is not considered.

A formal model for adaptation not governed by a central authority is presented in [ZC06] where each behavioural variant of a process is modelled as a Petri net. The Petri nets contain additional transitions determining when a reconfiguration of a process can occur and lead to an initial state of a Petri net of a different behavioural variant. In [BK07], adaptation is modelled by an automaton switching between programs that are also represented by automata. The system models in the above approaches are rather low-level
representations of global adaptation behaviour. Specifying the global adaptation behaviour of a complex system, such as the vehicle stability control system, is hardly feasible. Therefore, we aim at capturing MARS [TAFJ07] modelling concepts at a high-level of abstraction by SAS models in order to augment this constructive and explicit modelling technique for adaptive systems with existing verification techniques. The formal notion of SAS models is derived from (labelled) state transition systems [Kel76, MP95] and their extension to modular transition systems [FMS97]. Other related formal system models are I/O automata [LT87] or communicating finite state machines [BZ83].

With respect to specification of properties of adaptive systems, in [ZC05] a linear temporal logic is extended with an adapt operator for specifying requirements on the system before, during and after the adaptation using a similar model for representing adaptive systems as in [ZC06]. In [KB04], a transitional-invariant lattice is proposed for capturing invariants of programs before, during and after adaptation. Since adaptation is modelled as an explicit aspect in SAS systems, a special logic with operators for adaptation behaviour is not necessary in order to specify their properties. Instead, the standard temporal logic CTL* [Eme90] and its fragments can be employed for property specification facilitating the use of standard model checking procedures for verification.
Based on their formal semantics, synchronous adaptive systems can be formally analysed whether they satisfy specified properties. To this end, existing state of the art verification techniques can be applied including static analysis, theorem proving and model checking. Static analysis [NNH99] can be used to examine the syntactic structure of an SAS model automatically, for instance to check structural model consistency or consistency of the employed data model. For verification of SAS model properties specified in $L_{\text{SAS}}$, theorem proving [NPW02] and model checking [CGP99, CS01] can be employed. Theorem proving can deal with models using general logical constructs, such as infinite data domains or functionality specified by pre/post conditions, but higher-order theorem proving requires user-interaction and a great deal of expertise. In contrast, model checking can automatically establish properties given in temporal logic and provide counterexamples if a property fails. However, model checking is only applicable to finite-state systems because it is based on an exhaustive exploration of a model's state space. Furthermore, it suffers severely from the state-explosion problem making it efficiently applicable to models of just limited sizes.

In this chapter, we show how static analysis, theorem proving and model checking can be used for verification of SAS model properties. In particular, we explain how to verify functional model properties by theorem proving if functionality is given by pre/post condition specifications. Further, we describe how to reduce verification complexity of SAS models towards more
feasible model checking and present a technique to check stability of the adaptation process more efficiently than by standard model checking.

4.1 Static Analysis of Synchronous Adaptive Systems

Static analysis of programs [NNH99] is originally used to derive properties from program text without actually executing the program. Static analysis includes type checking, data flow analysis and abstract interpretation [CC77, CC79] of programs. Analysis results can be used to establish correctness properties of programs. Furthermore, they can be used for code optimisations in compilers, such as dead code elimination or constant propagation.

SAS models can, for example, be statically analysed for structural consistency. This may reveal conceptual errors and flaws in the design. As type checking of programs ensures that programs are well-formed, SAS models are analysed whether they are valid systems. They have to satisfy consistency constraints on the syntactic descriptions in the modules, as well as structural constraints on system level. Secondly, an SAS model translated from a MARS representation can be analysed whether the environment model describing dependencies between datives attached to module and system ports is appropriate. This allows important insights into the original MARS design potentially indicating conceptual errors.

4.1.1 Model Consistency Analysis

Model consistency of an SAS system is checked automatically by means of static analysis. Checking consistency is in particular important for SAS systems derived from MARS representations. While for SAS systems most of the consistency conditions are enforced by the definition of SAS syntax, consistency of MARS models is not implied from its mere syntactic structure. A MARS model translated to an SAS by the principles described in Section 3.1.3 has to be checked to be a valid system, before it is further analysed with respect to other properties.

The consistency conditions imposed on SAS models can be distinguished whether they refer to the structure on system level, to single modules or to module parts. With respect to the system structure, the consistency conditions referring to connections between system and module variables are as follows:
4.1. Static Analysis of Synchronous Adaptive Systems

- All module and system variables must be connected to other module or system variables.
- Connections may only link output variables with input variables. System inputs can only be connected to module inputs and module outputs to system outputs.
- Adaptive variables can only be connected to adaptive variables, functional variables only to functional variables.
- Each variable may only have one connection to another variable.

On module level, the consistency conditions refer to variable declarations and syntactic descriptions of transition functions, for instance:

- An initial value has to be assigned to each variable.
- Each variable used in a function description or a guard must be declared in the variable list.
- Input variables cannot be assigned in transition functions.
- Functional and adaptive output and local variables can only be assigned in the respective next state and next output functions.
- In expressions and constraints, operators and predicates have to be defined on their arguments.

The consistency conditions are checked on the syntax of an SAS model. Consistency checks can, for instance, be implemented as type checks by attaching types to variables and values and type constraints to operators and predicates.

4.1.2 Environment Model Analysis

SAS models translated from MARS models offer the possibility to statically analyse the physical domain model contained in MARS models. The physical domain model in MARS describes assumptions on the data that can be provided by a system’s environment. It is very important for the correct functionality of a system that the physical domain model exactly represents the actual environment a system is supposed to operate in. The reason is that system development is based on assumptions on possible data values and their ranges. If a special case that can occur in the environment is not considered in the domain model, the system will not be able to handle the respective situation causing major problems in particular for safety-critical applications.

For adaptive systems represented in MARS, the physical domain model contains the datives that ports can be typed with. As described in Section 2.2, a dative contains the definition of a functional value together with
its quality type. The quality type is used to capture the variant a functional value has been computed with by means of different modes and associated mode attributes. The mode and the mode attributes describe how a functional value can be used in further computations. By the specification of MARS modules, dependencies between datives in the domain model are introduced. In the configuration guards, constraints on modes and mode attributes are imposed. Influence functions assign modes and values for mode attributes to the output of a module. In this way, the realisation of a mode and its mode attributes depends on the modes and the mode attributes required in the respective configuration guards. As an example, consider the module \( V\text{.YawCalculation} \) determining the yaw rate of a car. The output value is typed with the dative \( v_{yaw} \). Amongst others, it contains the mode \( steering\text{.based} \). This mode can only be assigned by the configuration \( Steering\text{Based} \). This configuration requires the input \( v_{carRef} \) to be of the modes \( wheel\text{.based} \) or \( ax\text{.based} \). Further, the input \( steering\text{.angle }Input \), an instance of the dative \( steering\text{.angle} \), has to be of mode \( available \).

In [Tra05, DST06], different analyses of the physical domain model for adaptive systems are introduced that allow discovering unreasonable realisations of quality types. Unreasonable realisations, for instance, contain cycles between modes of a dative. In such a dative cycle, a mode of one dative requires an input of the same dative to be in a different mode. Furthermore, analyses are concerned with fault-tolerance properties of the physical domain model. These analyses can also be carried out statically on an SAS model translated from a MARS model. Their results can give valuable insights into the structure of an SAS system.

A physical domain model is not explicitly represented in SAS. However, the realisations of the modes are obtained by analysis of the adaptation aspects in the modules. In the adaptive output function of each module, output qualities are set to modes of the respective dative. These modes depend on the modes of the adaptive input variables. For the analysis of the physical domain model, these dependencies are represented in a data-dependancy graph [Tra05]. In this directed graph, it is captured which modes of output datives require which modes of the input datives in the SAS system. For an output mode and each dependent input mode, a node in the data-dependency graph is generated. AND-nodes representing Boolean conjunctions and OR-nodes representing disjunctions\(^1\) are used to express the structure of the constraints on the inputs by connecting data nodes

\(^1\) MARS constraints only use conjunctions and disjunctions.
with directed edges. Figure 4.1 shows a part of the data-dependency graph for mode \textit{steering\_based} in the module \textit{V\_YawCalculation}.

A realisation of a dative mode corresponds to a subtree in the data-dependency graph. In such a subtree, we find all modes of other datives that must be present such that this dative mode can be set. A realisation subtree for a mode is constructed by choosing the respective mode as root and all nodes without incoming edges at the leaves. In between, the subtree contains all connected nodes and edges where for each OR-node one incoming edge and for an AND-node all incoming edges are selected. The number of different realisations a mode has in an SAS system is the number of realisation subtrees for this mode. On the data-dependency graph, different analyses (cf. [DST06]) are carried out in order to validate that the specified system is a reasonable realisation of the physical domain model.

- \textbf{Feasibility analysis} determines whether a dative can be realised in the specified system. This means that there must be at least one mode of the dative not including \textit{unavailable} that can be realised. Feasibility of a dative is checked by computing the set of paths in the data-dependency graph from leaves to modes of this dative. If there is at least one path to a mode of a dative, the mode and thus the dative is realisable.

- \textbf{Dative cycle analysis} checks if the realisation of a dative mode depends on another mode of the same dative. This usually indicates an unreasonable realisation. In terms of the data-dependancy graph, this means that in no realisation of the mode a node with the same dative in a different mode occurs. Similarly, \textbf{mode cycle analysis} detects if the computation of a specific mode depends on the mode itself. This situa-
tion is not unusual, as values are often determined iteratively. Nevertheless, mode cycles can indicate a faulty model.

- **Dative independence analysis** and **mode independence analysis** consider the dependencies between realisations of a dative mode. Two realisations of a mode are dative independent if there are different realisations not sharing a common dative. Two realisations are mode independent if there exist different realisations that do not share the same mode of a dative. Dative and mode independence are important with respect to fault tolerance. If one mode can no longer be set due to a failure, mode independent realisations are not affected. The same holds for dative independent realisations which are not affected by a complete failure of a module.

- **Fault tree analysis** considers realisations of the mode **unavailable** of a dative. A quality is set to unavailable if the shutdown configuration **Off** is activated in a MARS module. Fault tree analysis determines which combination of other modes must also be present such that a shutdown occurs. The probability of a shutdown decreases, the more modes have to be unavailable as well. The result of this analysis can be taken as a measure for the fault tolerance of a system.

### 4.2 Theorem Proving of Synchronous Adaptive Systems

Properties of synchronous adaptive systems specified in the temporal logic $L_{SAS}$ can be directly verified by theorem proving. This requires representing the system model and the property in the input language of a theorem prover. Using axioms and deduction rules formalised inside the prover, model properties can be interactively derived. In this section, we show at the example of the Isabelle/HOL theorem prover [NPW02] how SAS models and their properties can be verified with theorem proving techniques, both for fully specified modules where functionality is given by SAS function descriptions as well as for underspecified modules where functionality is given by pre/post-condition specifications.

#### 4.2.1 Verification of Fully Specified Modules

Basically, there are two different ways how a formal specification language for system models can be translated to a representation of a theorem prover. The first is to give a deep embedding of the syntactic constructs of
the specification language to syntactic constructs, i.e. data types, in the input language of the theorem prover. The syntactic constructs in the theorem prover are provided with the execution semantics of the specification language constructs. This approach is in particular useful if one wants to reason about the constructs of the specification language itself by performing meta-proofs. The second approach is to provide a shallow embedding of the specification language into the theorem prover. In this direction, the semantics of the specification language is directly encoded in the input language of the theorem prover. The semantics of the models is captured by semantic constructs of the theorem prover. This approach is less complicated as it spares one level of indirection and suffices if only concrete system instances are to be considered. Therefore, we chose to implement a shallow embedding of SAS semantics into the Isabelle/HOL theorem prover. This also simplifies dealing with underspecified functionality in SAS models which can be given directly as pre/post-conditions in Isabelle/HOL syntax. For a more detailed comparison between deep and shallow embedding, consider [WN04].

In order to define a shallow embedding of SAS transitions system in Isabelle/HOL, we have to find a representation for SAS system states, for the initial states and for the transition functions in terms of the Isabelle/HOL semantics. Due to the finite number of variables in a system, we encode states as tuples of values rather than as mappings from variable names to values. This simplifies conducting the proofs. Variable references are encoded as selectors to such tuples. We do not distinguish between different kinds of variables (i.e. input, output, local, adaptation and functional variables) in the state encoding. The data types of the variables are chosen

```plaintext
datatype brakeConfs = brakes_setBrakes | brakes_Off
consts brakeGas_Input_quality_unavailable :: "int" (* 0u *)
consts brakeGas_Input_quality_available :: "int" (* 1u *)
consts brake_quality_available :: "int" (* 0u *)
consts brake_quality_unavailable :: "int" (* 1u *)
record brakes_params =
  useCONFIG_brake :: brakeConfs
  brakeGas_Input_brake :: int
  brakeGas_Input_quality:: "int"
  brake:: int
  brake_quality:: "int"
```

**Fig. 4.2.** State Representation in Isabelle for Module Brake
depending on the values of $Val$ a variable can assume. In our examples, we encode all variables as integers where string constants are encoded as integer constants. The data type for the used configuration variable is modelled as an enumeration type containing the names of the configurations as constants. This simplifies reasoning as the configurations are disjoint by definition.

Figure 4.2 shows the state representation for the sensor module $Brake$ in the vehicle stability control example. It is used to normalise the input brake force from the driver and propagate it to the yaw rate corrector. The module has two configurations $SetBrakes$ and $Off$ encoded in the datatype $brakeConfs$. For the qualities attached to input and output values, constants $available$ and $unavailable$ are defined. A module state is a record containing a variable to capture the used configuration $useCONFIG_{brake}$, variables for the input brake force $brakeGas_{Input_{brake}}$ and the attached quality $brakeGas_{Input_{quality}}$ and variables for the normalised output brake force $brake_{brake}$ and the attached quality $brake_{quality}$. Initial states are encoded as functions assigning initial values to the variables of a state. In Figure 4.3, the initialisation of the brake module is shown. The output brake force $brake$ is set to zero, and the output quality is set to $unavailable$. In Isabelle, $\%$ denotes the $\lambda$-operator. Symbolic brackets ($| and |)$ are used to encapsulate assignments to record fields.

An SAS module is divided into an adaptation aspect for the adaptation behaviour and the functional configurations. Before evaluating the functionality of a configuration, the adaptive part is evaluated using the $adapt\_next\_state$ and $adapt\_next\_out$ functions. The actual functionality of a configuration in the $next\_state$ and $next\_out$ functions is selected depending on the evaluation of the configuration guards. In Isabelle, we encode this behaviour by evaluating the Isabelle representation of the $adapt\_next\_state$ and $adapt\_next\_out$ functions first. Afterwards, we make a case distinction on the guard formulae by several if-clauses selecting the appropriate Isabelle representation for the $next\_state$ and $next\_out$ function of the active configuration.

```isar
constdefs init_brakes:: "brakes_params => brakes_params"
"init_brakes ==
  (\% f. f(|brake := 0, brake_quality := brake_quality_unavailable|))"
```

Fig. 4.3. Initial State Representation in Isabelle for Module $Brake$
4.2. Theorem Proving of Synchronous Adaptive Systems

Figure 4.4 shows the Isabelle representation of the state transition function \texttt{brakes} of the module \textit{Brake}. It contains an if-statement in which first the configuration guard of the configuration \texttt{setBrakes} is evaluated. If it is true, the configuration \texttt{setBrakes} is activated setting the used configuration variable appropriately and the quality of the output \texttt{brake\_quality} to \texttt{available}. If the input \texttt{brakeGas\_Input\_brake} is smaller or equal to zero, the output \texttt{brake} is set to the absolute value of the input, else it is set to zero. This reflects, that a value smaller or equal to zero for the brake and gas input is interpreted as a brake force, whereas a value greater than zero is interpreted as a gas value. If the configuration guard of the configuration \texttt{SetBrakes} is not true, the shutdown configuration \texttt{Off} is activated setting the used configuration to \texttt{Off} and the output quality to \texttt{unavailable}. Note, that for technical reasons state transition functions in Isabelle take two states as input for computing an output state. It is assumed that both input states are the same such that one can be used for reading values from the previous state while the second input state can be stepwise modified, until the modifications are propagated to the output state. In Isabelle, for each record field, there exists a selector function with the same name, e.g. \texttt{brakeGas\_Input\_quality g} yields the value of the field \texttt{brakeGas\_Input\_quality} in the record \texttt{g}.

For representing an SAS system, the module states are combined in a record to a global state together with the system variables. Connectors between different modules are encoded as copy operations between variables within the global state transition function. As the execution semantics of

\begin{verbatim}
constdefs brakes :: "brakes_params => brakes_params => brakes_params"
"brakes == ( % g.
  ( if ((brakeGas_Input_quality g) = brake_gas_Input_quality_available)
   then
     ( % f. f ((useCONFIG_brake := brakes__setBrakes)) ) o
     ( % f. f (|brake_quality := brake_quality_available|) )
   o
     (if ((brakeGas_Input_brake g) <= 0) then
       ( %f. f(|brake := (-(brakeGas_Input_brake g))|) )
     else (%f. f(|brake := 0|) )
   )
  (* end conf *)
else ( ( % f. f ((useCONFIG_brake := brakes__Off)) ) o
  ( % f. f (|brake_quality := brake_18_quality_unavailable|) )
 ) ) ) "
\end{verbatim}

Fig. 4.4. Transition Function in Isabelle for Module \textit{Brake}
SAS is synchronous, the module transition functions are evaluated one after the other in a global transition. In Figure 4.5, a part of the Isabelle representation of the vehicle stability control system is depicted. It contains the sensor modules Brake and Gas. Similar to the brake module, the gas module normalises the input for the gas. If the brake and gas input from the driver is greater than zero, the gas module propagates the value to its output. If it is smaller or equal to zero, the gas value is set to zero as a negative input is interpreted as brake force. A global state record global_state of this system part consists of two records, one for the brake module local state brakesT and one for the gas module local state gasT. The initial global state is obtained by initialising both local states. In a global transition, both modules perform a local transition. System input in a transition is encoded by input variables in the global system state that are set non-deterministically. Copy statements for system connections are not necessary in the example. Instead, we use an explicit assumption that the inputs brakeGas_Input_brake and brakeGas_Input_gas are equal.

The semantics of an SAS transition system is the set of possible computation paths. The paths of a system (or a module, if operating on module level) have to be formalised in Isabelle as a sequence of states. In this direction, we define a primitive recursive function global_path taking as input a state and a natural number (cf. Figure 4.6). For a natural number n as input, n global transitions are computed. The recursion base is the initial global state. The function global_step is only necessary to identify the first and second argument in the global transition function. If the input variables are (implicitly) universally quantified, all possible

```plaintext
record global_state =
  brakesT :: brakes_params
  gasT :: gas_params

constdefs init_global_state::"global_state => global_state => global_state" |
  "init_global_state p ==
  (%f. f(|brakesT := (init_brakes (brakesT p)),
       gasT := (init_gas (gasT p))|) )"

c constdefs global_transition::"global_state => global_state => global_state" |
  "global_transition p ==
  (%f. f(|brakesT := (brakes (brakesT p) (brakesT p)),
        gasT := (gas (gasT p) (gasT p))|) )"
```

Fig. 4.5. SAS Transition System for Brake and Gas Modules
4.2. Theorem Proving of Synchronous Adaptive Systems

execution paths with respect to inputs are considered. Because the input \( n \) can be arbitrary, all path of arbitrary (but finite) length are computed. For other approaches to formalise infinite sequences in theorem provers, consult [DGM97].

\[
\text{constdefs } \text{global_step} :: \text{global_state} \Rightarrow \text{global_state} \\
\text{global_step } p = (\text{global_transition } p \ p)
\]

\[
\text{consts } \text{global_path} :: \text{global_state} \Rightarrow \text{nat} \Rightarrow \text{global_state} \\
\text{primrec} \\
\text{global_path } p \ 0 = (\text{init_global_state } p \ p) \\
\text{global_path } p \ (\text{Suc } n) = (\text{global_step } (\text{global_path } p \ n))
\]

Fig. 4.6. Paths of an SAS System in Isabelle

In order to verify a property over this system representation, we also have to represent \( \mathcal{L}_{\text{SAS}} \) properties as properties over the global paths in Isabelle. Analogue to a system representation, this can be done in two ways by a deep or by a shallow embedding of \( \mathcal{L}_{\text{SAS}} \). A deep embedding requires providing data types for \( \mathcal{L}_{\text{SAS}} \) formulae and defining their semantics over system paths. To this end, the semantics of formulae is given by a function taking a system path, a formula representing a property and a function determining when an atomic proposition is fulfilled at a given state. The semantics function returns a truth value indicating whether the system fulfills the given property corresponding to the semantics of \( \mathcal{L}_{\text{SAS}} \). The advantage of a deep embedding is that it is also possible to reason about properties expressed in \( \mathcal{L}_{\text{SAS}} \) on a meta-level, e.g. for proving equivalence rules. For a shallow embedding, \( \mathcal{L}_{\text{SAS}} \) properties are directly expressed by semantic constructs of Isabelle over the data types used for defining system states and paths. The validity of a property is then defined by the semantics of the Isabelle constructs. For our purposes, a shallow embedding of properties in Isabelle suffices since no meta-proofs over \( \mathcal{L}_{\text{SAS}} \) formulae have to be performed. We only prove properties for concrete system instances. In our examples, we are mainly dealing with invariant properties that can be expressed in \( \mathcal{L}_{\text{SAS}} \) by \( A G \varphi \) if \( \varphi \) does not contain any temporal operators. Invariant properties can be expressed in Isabelle by universal quantification over \( n \) in the definition of a global path \( (\text{global_path } p \ n) \).

As an example for direct verification of properties of SAS models in Isabelle, we consider a safety property of the brake and gas modules. For correct operation of the system, the computed output brake and gas values
Lemma 4.4.2: States of Brake and Gas

Lemma: \[ \forall n \in \mathbb{N}. \left( (\text{brakeGas\_Input\_gas (gasT (global\_path p n)) = brakeGas\_Input\_brake (brakesT (global\_path p n))}) \land (\text{useCONFIG\_gas (gasT (global\_path p n)) \neq \text{gas\_Off})} \land (\text{useCONFIG\_brake (brakesT (global\_path p n)) \neq \text{brakes\_Off}}) \right) \implies \left( (\text{gas (gasT (global\_path p n)) > 0} \implies (\text{brake (brakesT (global\_path p n)) = 0}) \land (\text{brake (brakesT (global\_path p n)) > 0} \implies (\text{gas (gasT (global\_path p n)) = 0}) \right) \right) \]

apply (rule allI)
apply (case_tac n)
apply (simp)
apply (simp add: init_global_state_def)
apply (simp add: init_gas_def)
apply (simp add: init_brakes_def)
apply (simp add: brakeGasInv)
done

Fig. 4.7. Safety Property of Modules Brake and Gas

have to be mutually exclusive if both modules are not in their shutdown configuration \text{Off}. This means, if the output of the brake module is greater than zero, the output of the gas module has to be equal to zero, and if the gas output is greater than zero, the brake output has to be zero. As stated in Section 3.2.2, this property is expressed in \( L_{SAS} \) as

\[
(SafetyBrakeGas) \quad AG \left( (useconf_{Brake} \neq \text{Off} \land useconf_{Gas} \neq \text{Off}) \rightarrow (brake > 0 \rightarrow gas = 0) \land (gas > 0 \rightarrow brake = 0) \right)
\]

In order to establish \((SafetyBrakeGas)\) for the vehicle stability control system, we consider the modules \text{Brake} and \text{Gas} in isolation. The decomposition results in Section 6.2.1 will later allow us to lift this property to the overall vehicle stability control system. The safety property cannot be shown by standard model checking. The reason is that brake and gas inputs and also brake and gas outputs are integer variables. This makes the models of both modules as well as their combination infinite-state. As standard model checking is limited to finite-state systems, model checking of the original specification is impossible. A theorem prover, however, can deal with infinite-state systems such that the property can be verified over the original system specification.

In Isabelle, the property \((SafetyBrakeGas)\) is expressed over a path of the system containing the brake and gas modules by universal quantification over the path parameter \( n \). As additional assumption, we add that the
input to both modules brakeGas_Input_brake and brakeGas_Input_gas is equal. In the upper part of Figure 4.7, the translation of the property (SafetyBrakeGas) to an Isabelle lemma is shown. In order to prove the lemma, we first show an invariant over the global transitions of the system. The lemma BrakeGasInv in the upper part of Figure 4.8 states that if the input to both modules is the same and both modules are not in their shutdown configuration, brake and gas output are mutually exclusive in each global transition. The lower part of Figure 4.8 shows the proof script computing the proof for this lemma. A proof script is a kind of program that tells the theorem prover how to conduct a proof for a theorem. It comprises the application of several tactics (the applies) which can be regarded as subprograms in the proving process. The proof goes by simplifications using the definitions of global transition and local transition functions.

Using the invariant lemma brakeGasInv, we can verify the safety property (SafetyBrakeGas) over the brake and gas modules. The proof in the lower part of Figure 4.7 uses first universal quantification introduction with apply (rule allI) which transforms the universal quantification into a big conjunction over all n. Then, we use a case distinction apply (case_tac n) over n which splits the proof goal into one goal for n = 0, i.e. the initial states, and one for nat = Suc(n), i.e. the recursion step. For the first case, we use the definitions of the initial global state and the initial states of the brake and gas modules. For the second proof goal, the previously established invariant brakeGasInv over the global transitions is applicable and completes the proof.

```
lemma brakeGasInv: "(((brakeGas_Input_gas (gasT (global_step p)) = 
    brakeGas_Input_brake (brakesT ( global_step p))) 
  & (useCONFIG_gas (gasT (global_step p)) ˜= gas_Off) 
  & (useCONFIG_brakes (brakesT (global_step p)) ˜= brakes_Off)) 
  --> (((gas (gasT (global_step p)) > 0) 
    --> (brake (brakesT (global_step p)) = 0)) 
  & ((brake (brakesT (global_step p)) > 0) 
    --> (gas (gasT (global_step p)) = 0))))"
apply(simp add: global_step_def)
apply(simp add: global_transition_def)
apply (simp add: gas_def)
apply (simp add: brakes_def)
done
```

**Fig. 4.8.** Invariant over Global Transition of Modules Brake and Gas
4.2.2 Verification with Underspecified Functionality

If the functional behaviour of the module configurations is given by logical pre/post-condition specifications, we can use a theorem prover for inferring properties of the overall system without a complete specification. To this end, the theorem prover Isabelle/HOL [NPW02] provides an implementation of Hilbert’s $\varepsilon$-operator. Hilbert’s $\varepsilon$-operator applied to a predicate $P(x)$ by $\varepsilon x. P(x)$ returns some element satisfying the predicate if there is one. Using the $\varepsilon$-operator, the pre/post-condition specification of the functionality in a module configuration can be represented and analysed. The remaining parts of the module can be represented in Isabelle in the same way as for fully specified modules. Nonetheless, the degree of detail of the pre/post-condition specification has a significant impact on the properties that can be verified. If the specification is very detailed, we will most probably be able to prove stronger properties than if the specification is rather weak.

Figure 4.9 shows a part of the output transition function `steering_angle_delimiter` of the steering angle delimiter module in configuration `Controlled` specified in Isabelle/HOL using the $\varepsilon$-operator `Eps`. The specification for this configuration says that if the input steering angle `steering_angle/Input` in the current state `g` is larger than the maximal possible steering angle `max_Angle(v_carRef g)` at the current speed of the car `v_carRef`, the output steering angle `delimited_steering_angle` is smaller than the maximal possible steering angle `max_Angle(v_carRef g)` at the current speed of the car and that it is also smaller than the input steering angle `steering_angle/Input`. In case the input steering angle `steering_angle/Input` is smaller than the maximal possible steering angle `max_Angle(v_carRef g)` at the current speed, the output steering angle `delimited_steering_angle` is assigned to the input steering angle `steering_angle/Input`.

Using this specification of the output transition function `steering_angle_delimiter`, we can prove a safety invariant of the `Controlled` configuration. The invariant expressed as Isabelle lemma in Figure

```isar
{ if ((steering_angle/Input g) > (max_Angle (v_carRef g))) then 
  (\%f. f (| delimited_steering_angle :=
    (Eps (%x. x < max_Angle(v_carRef g) 
    & x < (steering_angle/Input g)))))))
else ( (\%f. f (| delimited_steering_angle := (steering_angle/Input g)))))
```

Fig. 4.9. Use of Epsilon Operator in Steering Angle Delimiter Module
4.2. Theorem Proving of Synchronous Adaptive Systems

lemma "max_Angle(v_carRef g) > 0 -->
   (useCONFIG (steering_angle_delimiter g g) = Controlled) -->
   (delimited_steering_angle (steering_angle_delimiter g g) <= max_Angle(v_carRef g))"

Fig. 4.10. Safety Invariant for Controlled Configuration of Steering Angle Delimiter Module

4.10 says that the output steering angle delimited_steering_angle is always smaller or equal to the maximal possible steering angle max_Angle(v_carRef g) at the current speed if the module uses configuration Controlled and the maximal steering angle is greater than zero.

The steering angle delimiter module has a second configuration Forwarded besides the shutdown configuration Off in order to degrade the functionality if the current speed of the car is unavailable. In this configuration, the output steering angle delimited_steering_angle is simply set to the input steering angle steering_angle_Input. We can show a second safety invariant depicted in Figure 4.11 for all configurations of the module except for the shutdown configuration Off. The invariant says that the output steering angle delimited_steering_angle is always smaller or equal to the input steering angle steering_angle_Input given the used configuration is not the shutdown configuration.

lemma "max_Angle(v_carRef g) > 0 -->
   (useCONFIG (steering_angle_delimiter g g) ˜= Off) -->
   (delimited_steering_angle (steering_angle_delimiter g g) <= steering_angle_Input (steering_angle_delimiter g g) )"

Fig. 4.11. Safety Invariant for Non-Shutdown Configurations of Steering Angle Delimiter Module

To complete the proofs of the two safety invariants, additional lemmata over properties of the \( \epsilon \)-operator are required besides the specification of the transition function steering_angle_delimiter. For instance, the lemma help (cf. Figure 4.12) says that if the integers \( y \) and \( z \) are greater than zero, there exists some integer \( x \) with the property that it is smaller than \( y \) and \( z \) and also smaller or equal to \( y \).

lemma help: "y::int > 0 & z::int > 0
   --> ((Eps %x::int. x < y & x < z) <= y)"

Fig. 4.12. Auxiliary Lemma for Proofs using Eps-Operator
Chapter 4. Verification of Synchronous Adaptive Systems

4.3 Model Checking of Synchronous Adaptive Systems

Model checking [CGP99, CS01] is an automatic technique for establishing temporal logic properties over transition systems. Model checking analyses properties of a system by an exhaustive exploration of the system’s state space. Starting from the set of initial states, the set of states reachable via system transitions is computed iteratively until a fixpoint is reached. The validity of the property is checked on the reachable states. An overview of model checking algorithms for different temporal logic fragments can for instance be found in [CGP99].

Model checking can be performed fully automatically not requiring user-interaction or expertise to analyse a system with respect to a property. Another advantage is that model checking can produce a counterexample if a property does not hold in a system. This computation path violating the property provides valuable insights to understand the real reasons for failure. In these two aspects, model checking is superior to theorem proving. However, model checking is in general restricted to finite state systems such that the computation of the reachable states terminates. Furthermore, model checking suffers severely from the state-explosion problem. This means that the number of system states grows exponentially in the number of system variables. Hence, standard model checking is only efficiently applicable for systems of limited sizes. In this section, we, first, present a general procedure how verification of generic and application-specific properties of SAS systems can be made amenable to model checking. Second, we describe an efficient procedure developed for verification of the important stability property of the adaptation process.

Verification Complexity Reduction for Model Checking of SAS Models

The semantics of synchronous adaptive systems is given in terms of state transition systems (Definition 3.8). Properties over SAS are specified in the temporal logic $L_{SAS}$. This allows verifying SAS model properties by model checking. In this direction, the system model and the property have to be translated in a semantics-preserving manner to the input language of the applied model checking tool. If the property can be established in the model checking run, it holds true in the SAS system. If verification fails, the model checker can produce a counterexample that retranslated to the SAS model level shows where the model violates the property. Examples for translations of SAS models and properties to the input of model checking tools are described in Chapter 7.
4.3. Model Checking of Synchronous Adaptive Systems

SAS models are designed to capture MARS models at a high-level of abstraction. Modelling concepts are not restricted with respect to verification. This induces a large complexity for verification in particular with respect to model checking. The reasons are that, first, SAS models can use infinite data domains such as integers making them infinite-state systems. Second, SAS systems can be composed from a large number of modules and can contain many variables resulting in an enormous state space. Thus, in order to make SAS models amenable to efficient model checking, we have to develop means for verification complexity reduction.

In this thesis, we present two approaches in this direction: first, property-preserving model transformations in Chapter 5, and second, compositional reasoning strategies in Chapter 6. Figure 4.13 shows a schematic view of the verification complexity reduction procedure for SAS verification tasks proposed in this thesis. Compositional reasoning strategies exploit the modular structure of SAS models in order to split a global verification task, consisting of a model and a property to be verified, into a number of less complex local verification tasks over parts of the model. The decomposition process ensures that the validity of the local properties implies the global property. Model transformations can further reduce the verification complexity of the local verification tasks by producing a simpler verification task with a transformed model and a corresponding transformed property. The transformations are property-preserving such that the validity of a transformed property over a transformed model implies the validity of the original property over the original model. In our experimental evaluation in Chapter 7, we demonstrate that for verifying properties of the vehicle stability control system the proposed techniques for verification complexity reduction indeed make model checking of SAS models much more feasible.

**Verification of Stability of the Adaptation Process**

While the previously described verification complexity reduction procedure aims at verification tasks comprising general system properties, for certain kinds of properties, special verification procedures can be designed that are more efficient than standard model checking. Such a property is the stability of the adaptation process, an important property for SAS models as adaptation is not controlled by a central authority. Stability of the adaptation process in an SAS model can be expressed in $L_{\mathcal{C}_{SAS}}$ (cf. Section 3.2.2). This allows verification by standard linear time logic model checking, but there are more efficient ways to check stability.
In [ASSV07], a special verification technique for stability of the adaptation process of SAS models is developed. Instead of analysing the original system with respect to the stability property, the approach analyses two different properties independently. First, it computes all reachable states in the original system. Second, the execution paths of the original system are restricted to all paths where the input does not change and which do not contain self-loops on states. This restricted set of paths is analysed for states that are the beginning of infinite computation paths. These are exactly the paths where the output changes despite unchanged input. If now the intersection of the set of reachable states in the original system with the set of states with infinite paths in the second system is non empty, we know that in the original system an unstable path occurs. If the intersection is empty, the system is stable and vice versa. Checking these two systems independently is more efficient than checking the linear time stability formula due to the complexity of linear time model checking.

With this special verification technique, an unstable adaptation sequence in an early version of the vehicle stability control system could be discovered. Figure 4.14 shows a sequence of configurations used by the modules V.YawCalculation and V.CarRef for computing the yaw rate and the speed of the car, respectively. Both start in a configuration where the output value is derived from the wheel speeds in a configuration Wheel-Based. But both modules also have a configuration with higher priority in which each of the two modules computes its output from the output of the
other module. This means that the module \(V_{CarRef}\) computes the speed of the car from the yaw rate in configuration \(VYawBased\), and the module \(V_{YawCalculation}\) computes the yaw rate from the car’s speed in configuration \(SteeringBased\). However, this mutual dependency is excluded by the configuration guards such that in the next cycle both modules switch back to the wheel based configuration. Then, the dependency no longer exists, and both modules upgrade again and so on. Despite the input of both modules does not change, the used configurations do not stabilise. The unstable sequence of reconfigurations could be excluded by choosing different priorities for the configurations in one of the modules.

### 4.4 Related Work

Verification of adaptive systems with existing verification techniques is a fairly recent area of research. In this section, we review approaches applying static analysis and model checking for analysis of adaptive systems. Furthermore, we discuss work on theorem proving for verification of general non-adaptive transition systems.

**Static Analysis of Adaptive Systems**

Static analysis [NNH99] is a technique originally used to derive properties from program text without actually executing the programs. Prominent static analysis approaches include type checking, data flow analysis or abstract interpretation [CC77, CC79]. In [Tra05, DST06], different static analyses on physical domain models of adaptive systems are introduced that allow discovering unreasonable realisations of quality types. In this thesis, these static analyses are adopted to synchronous adaptive systems in order to analyse properties of SAS environment models.

![Unstable Reconfiguration Sequence](image.png)

*Fig. 4.14. Unstable Reconfiguration Sequence*
Chapter 4. Verification of Synchronous Adaptive Systems

Model Checking Adaptive Systems

A first attempt to verify the adaptation behaviour of MARS models using model checking has been made in [SST06]. MARS models are directly translated to the input of a model checker in the Averest framework. This, however, limits MARS models to modelling concepts that can be directly encoded in the applied model checker. Furthermore, the analysis is restricted to models of limited sizes in order to avoid the state-explosion problem. In contrast, in our work MARS modelling concepts are captured by SAS at a high-level of abstraction and made amenable to model checking by model transformations and compositional reasoning strategies.

Verification of Transition Systems by Theorem Proving

Approaches to the verification of general non-adaptive state transition systems are similar to theorem proving of SAS models. In [Mül98], a metatheory of I/O automata [LT87] is formalised in Isabelle/HOL [NPW02]. The formalisation relies on a representation of finite and infinite sequences of states by lazy lists. In earlier work (cf. [NS94]), traces of I/O automata are formalised by a mapping from natural numbers to states as in our approach. On the lazy list representation of computation traces, a general temporal logic and a temporal logic of steps is introduced for property specification comprising similar concepts as $\mathcal{L}_{SAS}$. Instead of specifying properties by temporal logic formulae, I/O automata are also used as observers for specification purposes in [MN95]. In this case, verification of a property reduces to checking trace inclusions between implementation and specification automata.

In [Hut98], invariants of state transition systems are verified using the VSE theorem proving environment. State transition systems are specified in the VSE system very similar to our approach. States are represented as records of state variables. Transitions are given as pre/post-condition specifications denoting when a transition is enabled and what its effect is. Based on this state transition system representation, it is checked that all initial states satisfy the given invariant and that it is preserved by each transition in order to conclude that the system satisfies the invariant.

[RSS95] presents a formalisation of the $\mu$-calculus in PVS for expressing properties from universal branching time logic over state transition systems. The systems are represented in the same way as in our approach. Besides verifying properties in PVS, a model checker as an integrated proof tactic can be used discharging suitable subproof goals. To this end, the $\mu$-calculus specification is translated to the input logic of the model checker.
This usage of a model checker as a special proof tactic is very popular and applied e.g. in [MN95, BBC+00, BGL+00].

The STeP framework [BBC+00] for deductive and algorithmic verification of reactive and real-time systems allows reasoning about modular transition systems and their properties in linear temporal logic. Modular transition systems are very similar to SAS with respect to their modular structure. Using deductive verification rules for temporal logic [MP92], the verification of temporal logic is reduced to first-order verification problems which can be discharged by special decision procedures, including model checking or general-purpose theorem proving.
Chapter 5

Model Transformations for Verification
Complexity Reduction

"Simplicity is prerequisite for reliability."
(Edsger W. Dijkstra)

Verification by model checking is based on an exhaustive exploration of a model's state space. In general, this limits its applicability to finite-state systems. Additionally, model checking suffers severely from the state-explosion problem since the number of system states increases exponentially in the number of system variables. This only allows for efficient model checking for systems of restricted sizes. To counter these problems, a number of model transformation techniques [CGP99, McM99, Sha02] have been proposed to reduce the sizes of models. However, it is crucial that model transformations are property-preserving. If a property can be established over a transformed model, this has to imply that the corresponding property also holds over the original model. Then, verification of a property reduces to model checking of the transformed model which is in general less complex and makes the analysis more efficient. Figure 5.1 sketches the connections between the original and transformed models and properties for property-preservation under model transformations.

In this chapter, we present model transformations on synchronous adaptive systems as a means for verification complexity reduction. Property-preservation under transformations is justified by consistent simulations and bisimulations between original and transformed models with respect to the considered property. In order to establish the correctness of a concrete transformation, we use a translation validation approach [BSPH07] adopted from compiler verification. The proposed model transformations are, first, slicing techniques [Wei84, Tip95] that can be performed au-
Chapter 5. Model Transformations

Fig. 5.1. Property Preservation under Model Transformations

Automatically without user interaction, and, second, data domain abstractions [CGL94, DHJ+01] requiring a certain degree of user guidance. Slicing techniques remove irrelevant parts of a model with respect to a property leading to a smaller state space. Data domain abstractions map large or infinite data domains to smaller and finite abstract domains reducing complex and infinite-state models to simpler finite-state models. Other reduction techniques, such as symmetry reduction [McM99] or partial order reduction [CGP99], are not appropriate for SAS models, as those reduction techniques exploit the characteristics of asynchronous systems.

5.1 Property Preservation by System Simulation

In this section, we provide the theoretical foundations for justifying property preservation between original and transformed models and show how a translation validation infrastructure for verifying correctness of a concrete transformation can be implemented. In Section 5.2 and Section 5.3, the property preservation results are used to justify the correctness of model reductions by slicing and data domain abstractions, respectively. In Chapter 6, compositional reasoning is justified by the same means.

5.1.1 Formal Foundations for Property Preservation

In order to show that the validity of a property over a transformed system implies the validity of the property over the original system, the two systems have to have the same behaviours with respect to the property. Milner [Mil71] introduced the notion of simulation in order to capture that two systems have similar behaviours. This notion can also be transferred to SAS models. An SAS transition system $T$ is simulated by a transition system $\hat{T}$ if we can find a simulation relation $R$ between the two sets of states such that, first, for all initial states of $T$, there exists a related initial state
in $\hat{T}$, and second, that for any pair of related states with a transition in $T$, there is also a transition in $\hat{T}$ such that the resulting states are related. Intuitively, the relation between the two sets of states captures in which sense the states are considered similar. If now the relation holds over the initial states of both systems and is preserved by the transitions, the behaviours of the two systems denoted by their possible computation paths are considered similar.

**Definition 5.1 (Simulation between SAS).** Let $T = (\Sigma, \text{Init}, \sim)$ and $\hat{T} = (\hat{\Sigma}, \hat{\text{Init}}, \hat{\sim})$ be two SAS transition systems. The system $\hat{T}$ simulates the system $T$, $T \preceq \hat{T}$, iff there exists a simulation relation $R \subseteq \Sigma \times \hat{\Sigma}$ such that

1. for all initial states $\sigma_0 \in \text{Init}$, there exists $\hat{\sigma}_0 \in \hat{\text{Init}}$ such that $R(\sigma_0, \hat{\sigma}_0)$
2. for $i \geq 0$ and $\sigma_i, \sigma_{i+1} \in \Sigma$ and $\hat{\sigma}_i \in \hat{\Sigma}$ with $R(\sigma_i, \hat{\sigma}_i)$ and $\sigma_i \sim \sigma_{i+1}$, there exists $\hat{\sigma}_{i+1} \in \hat{\Sigma}$ such that $\hat{\sigma}_i \hat{\sim} \hat{\sigma}_{i+1}$ and $R(\sigma_{i+1}, \hat{\sigma}_{i+1})$.

While simulation only captures similar behaviours in one direction, bisimulation denotes an equivalence between the two system behaviours. To this end, a bisimulation relation describes in which sense two system states are considered equal. The bisimulation relation has to be preserved by the initial states and the transitions in both directions in order to induce an equivalence on the computation paths of the two systems. Thus for bisimulation, we require two additional properties of the bisimulation relation with respect to the simulation relation: First, also for each initial state of the second system, there must be an initial state of the first system such that the relation holds. Second, if there is a transition in the second system from $\hat{\sigma}_i$ to $\hat{\sigma}_{i+1}$ and there is a state $\sigma_i$ with $R(\sigma_i, \hat{\sigma}_i)$ in the first system, then there must be a state $\sigma_{i+1}$ such that there is a transition from $\sigma_i$ to $\sigma_{i+1}$ and the two resulting states are in relation $R(\sigma_{i+1}, \hat{\sigma}_{i+1})$.

**Definition 5.2 (Bisimulation between SAS).** Let $T = (\Sigma, \text{Init}, \sim)$ and $\hat{T} = (\hat{\Sigma}, \hat{\text{Init}}, \hat{\sim})$ be two SAS transition systems. The system $\hat{T}$ and the system $T$ are bisimilar, $T \simeq \hat{T}$, iff there exists a bisimulation relation $R \subseteq \Sigma \times \hat{\Sigma}$ such that

1. for all initial states $\sigma_0 \in \text{Init}$, there exists $\hat{\sigma}_0 \in \hat{\text{Init}}$ such that $R(\sigma_0, \hat{\sigma}_0)$, and for all initial states $\hat{\sigma}_0 \in \hat{\text{Init}}$, there exists $\sigma_0 \in \text{Init}$ such that $R(\sigma_0, \hat{\sigma}_0)$
2. for $i \geq 0$ and $\sigma_i, \sigma_{i+1} \in \Sigma$ and $\hat{\sigma}_i \in \hat{\Sigma}$ with $R(\sigma_i, \hat{\sigma}_i)$ and $\sigma_i \sim \sigma_{i+1}$, there exists $\hat{\sigma}_{i+1} \in \hat{\Sigma}$ such that $\hat{\sigma}_i \hat{\sim} \hat{\sigma}_{i+1}$ and $R(\sigma_{i+1}, \hat{\sigma}_{i+1})$. 
3. for $i \geq 0$ and $\hat{\sigma}_i, \hat{\sigma}_{i+1} \in \hat{\Sigma}$ and $\sigma_i \in \Sigma$ with $R(\sigma_i, \hat{\sigma}_i)$ and $\hat{\sigma}_i \sim \hat{\sigma}_{i+1}$, there exists $\sigma_{i+1} \in \Sigma$ such that $\sigma_i \sim \sigma_{i+1}$ and $R(\sigma_{i+1}, \hat{\sigma}_{i+1})$.

By definition of simulation and bisimulation, it is clear that a bisimulation induces also a simulation between the two systems where the bisimulation relation can be used as simulation relation in both directions. However, if we can show that there is a simulation between the two systems in one direction and also a simulation in the other direction, this does not imply a bisimulation between the two systems as the two simulation relations may be different.

If a transition system $T$ is simulated by a system $\hat{T}$, we can show that for each path in $T$ there is a corresponding path in $\hat{T}$. This result is important for preservation of temporal operators in an $L_{SAS}$ formula. The proof proceeds by induction on the length of a path and holds for finite paths. It can easily be lifted via a contradiction proof to infinite paths as well.

**Lemma 5.3 (Corresponding Paths in Similar Systems).** Let $T$ and $\hat{T}$ be two $SAS$ transition systems such that $T \preceq \hat{T}$ with simulation relation $R$. Then, for every path $\pi = \sigma_0\sigma_1 \ldots \in Paths(T)$ and for all $i \geq 0$, there exists a corresponding path $\hat{\pi} = \hat{\sigma}_0\hat{\sigma}_1 \ldots \in Paths(\hat{T})$ such that $R(\sigma_i, \hat{\sigma}_i)$.

**Proof.** By induction on length of path $\pi$.

- **Base case:** For a path of length one, we have $\pi = \sigma_0$ where $\sigma_0 \in Init$. Since $T \preceq \hat{T}$, for all initial states, there exists a corresponding initial state in the second system $\hat{\sigma}_0 \in \hat{Init}$ such that for all $i$ it holds that $R(\sigma_0, \hat{\sigma}_0)$.

- **Induction hypothesis:** For a path up to length $n$, it holds that if $\pi_n = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in Paths(T)$, there exists a corresponding path $\hat{\pi}_n = \hat{\sigma}_0\hat{\sigma}_1 \ldots \hat{\sigma}_{n-1} \in Paths(\hat{T})$ such that $R(\sigma_i, \hat{\sigma}_i)$.

- **Induction step:** $n \rightarrow n + 1$: Let now $\pi_{n+1} = \sigma_0\sigma_1 \ldots \sigma_n \in Paths(T)$. Suppose, there exists a transition in $T$ with $\sigma_{n-1} \sim \sigma_n$. Furthermore, by induction hypothesis, we know that there exists a corresponding path $\hat{\pi}_n = \hat{\sigma}_0\hat{\sigma}_1 \ldots \hat{\sigma}_{n-1} \in Paths(\hat{T})$ such that $R(\sigma_i, \hat{\sigma}_i)$ for $i = 0, \ldots, n - 1$. As $T \preceq \hat{T}$, there also exists a transition $\hat{\sigma}_{n-1} \sim \hat{\sigma}_n$ such that $R(\sigma_n, \hat{\sigma}_n)$. Thus, there is a corresponding path of length $(n + 1)$ with $\hat{\pi}_{n+1} = \hat{\sigma}_0\hat{\sigma}_1 \ldots \hat{\sigma}_{n-1}\hat{\sigma}_n \in Paths(\hat{T})$.

For two bisimilar $SAS$ systems, we can show the stronger property that for every path in the first system $T$ there exists a corresponding path of the second system $\hat{T}$ and vice versa.
5.1. Property Preservation by System Simulation

Lemma 5.4 (Corresponding Paths in Bisimilar Systems). Let $T$ and $\hat{T}$ be two transition systems such that $T \cong \hat{T}$ with bisimulation relation $\mathcal{R}$. Then, for every path $\pi = \sigma_0\sigma_1\ldots \in \text{Paths}(T)$, there exists a corresponding path $\hat{\pi} = \hat{\sigma}_0\hat{\sigma}_1\ldots \in \text{Paths}(\hat{T})$ such that for all $i \geq 0$, it holds that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$, and for every path $\hat{\pi} = \hat{\sigma}_0\hat{\sigma}_1\ldots \in \text{Paths}(\hat{T})$, there exists a corresponding path $\pi = \sigma_0\sigma_1\ldots \in \text{Paths}(T)$ such that for all $i \geq 0$, it holds that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$.

Proof. By induction over the length of the paths $\pi$ and $\hat{\pi}$.

- Base case: For a path of length one, we have $\pi = \sigma_0$ and $\hat{\pi} = \hat{\sigma}_0$ where $\sigma_0 \in \text{Init}$ and $\hat{\sigma}_0 \in \text{Init}$. Since $T \cong \hat{T}$, for all initial states of $T$, there exists a corresponding initial state in $\hat{T}$. This yields $\mathcal{R}(\sigma_0, \hat{\sigma}_0)$ and vice versa.

- Induction hypothesis: For a path up to length $n$, it holds that if $\pi_n = \sigma_0\sigma_1\ldots\sigma_{n-1} \in \text{Paths}(T)$, then there exists a corresponding path $\hat{\pi}_n = \hat{\sigma}_0\hat{\sigma}_1\ldots\hat{\sigma}_{n-1} \in \text{Paths}(\hat{T})$ such that for all $i$ it holds that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ and vice versa.

- Induction step (left-to-right): Let now $\pi_{n+1} = \sigma_0\sigma_1\ldots\sigma_n \in \text{Paths}(T)$. Then suppose, there exists a transition in $T$ with $\sigma_{n-1} \sim \sigma_n$. By induction hypothesis, we know that there exists a corresponding path $\hat{\pi}_n = \hat{\sigma}_0\hat{\sigma}_1\ldots\hat{\sigma}_{n-1} \in \text{Paths}(\hat{T})$ such that $\mathcal{R}(\hat{\sigma}_i, \hat{\sigma}_i)$ for $i = 0, \ldots, n-1$. As $T \cong \hat{T}$, there also exists a transition $\sigma_{n-1} \sim \sigma_n$ such that $\mathcal{R}(\sigma_n, \hat{\sigma}_n)$. Thus, there is a corresponding path of length $(n+1)$ with $\pi_{n+1} = \sigma_0\sigma_1\ldots\sigma_{n-1}\sigma_n \in \text{Paths}(T)$.

- Induction step (right-to-left): Let now $\hat{\pi}_{n+1} = \hat{\sigma}_0\hat{\sigma}_1\ldots\hat{\sigma}_n \in \text{Paths}(\hat{T})$. Then suppose, there exists a transition in $\hat{T}$ with $\hat{\sigma}_{n-1} \sim \hat{\sigma}_n$. By induction hypothesis, we know that there exists a corresponding path $\pi_n = \sigma_0\sigma_1\ldots\sigma_{n-1} \in \text{Paths}(T)$ such that $\mathcal{R}(\hat{\sigma}_i, \hat{\sigma}_i)$ for $i = 0, \ldots, n-1$. As $T \cong \hat{T}$, there also exists a transition $\sigma_{n-1} \sim \sigma_n$ such that $\mathcal{R}(\sigma_n, \hat{\sigma}_n)$. Thus, there is a corresponding path of length $(n+1)$ with $\pi_{n+1} = \sigma_0\sigma_1\ldots\sigma_n \in \text{Paths}(T)$.

Simulation or bisimulation between two systems is only one part for arguing that the validity of a property is preserved under a model transformation. The results on corresponding paths provide the necessary prerequisites to justify preservation of temporal operators in a formula. However, to complete the argument that a property is preserved, a consistency condition has to be imposed on simulation or bisimulation relations. The consistency condition requires that all atomic propositions of a property and their negations are preserved by the respective relation.
In a model transformation, e.g. in data domain abstraction, it can happen that the interpretation of values and variables is changed from one system to the other. For instance, if all values greater than zero are mapped to an abstract value \textit{high}. This also causes that the atomic propositions that can be used to express properties over both systems become different. However, by the transformation it is determined which atomic propositions in the transformed system correspond to which atomic propositions in the original system.

Thus, in order to formulate the consistency condition over simulation and bisimulation relations, we need a concretisation function \( C \) that associates corresponding atomic propositions in the original and the transformed system with each other and maps a property \( \hat{\varphi} \) over the transformed system \( \hat{T} \) to a corresponding property \( \varphi \) over the original system \( T \). The concretisation function \( C \) relies on a function \( f \) mapping expressions over \( \hat{T} \) to expressions defined in \( T \). This reflects the potentially different interpretations of variables and values in expressions over both systems and associates corresponding atomic propositions. If the interpretation of variables and values is unchanged by the transformation, \( f \) is the identity. Further, the concretisation function preserves the structure of the temporal formula. In the following definition, \( \mathcal{L}_{SAS}[T] \) denotes the set of \( \mathcal{L}_{SAS} \) formulae defined over \( T \) and \( \mathcal{A}\mathcal{L}_{SAS}[T] \) universal formulae defined over \( T \).

**Definition 5.5 (Concretisation).** Let \( T \) be an SAS transition system and \( \hat{T} \) the corresponding transformed SAS transition system. Let \( f : \hat{\text{Expr}} \rightarrow \text{Expr} \) be a mapping from expressions defined over \( \hat{\text{Var}} \) and \( \hat{\text{Val}} \) in \( \hat{T} \) to expressions defined over \( \text{Var} \) and \( \text{Val} \) in \( T \). The concretisation function \( C : \mathcal{L}_{SAS}[\hat{T}] \rightarrow \mathcal{L}_{SAS}[T] \) is defined inductively on the structure of an \( \mathcal{L}_{SAS} \) formula by:

- \( C(r(\text{expr}_1, \text{expr}_2)) = r(f(\text{expr}_1), f(\text{expr}_2)) \)
- \( C(\neg \varphi) = \neg C(\varphi) \)
- \( C(\varphi_1 \land \varphi_2) = C(\varphi_1) \land C(\varphi_2) \)
- \( C(\mathcal{X} \varphi) = \mathcal{X} C(\varphi) \)
- \( C(\varphi_1 U \varphi_2) = C(\varphi_1) U C(\varphi_2) \)
- \( C(\mathcal{E} \varphi) = \mathcal{E} C(\varphi) \)

Using the concretisation function mediating between interpretations of variables and values in original and transformed systems and associating corresponding atomic propositions, we can define the notion of a simulation and bisimulation consistent with respect to a property. A consistent
(bi-)simulation is a (bi-)simulation between two systems where for two states in the (bi-)simulation relation it holds that if a state of the transformed system satisfies an atomic proposition, the related original state satisfies the concretisation of this proposition, and vice versa. Let \( \text{Atoms}(\varphi) \) denote the atomic propositions of the property \( \varphi \).

**Definition 5.6 (Consistent Simulation and Bisimulation).** Let \( \mathcal{T} = (\Sigma, \text{Init}, \leadsto) \) and \( \mathcal{T} = (\hat{\Sigma}, \text{Init}, \leadsto) \) be two SAS transition systems and \( \hat{\varphi} \) an \( \mathcal{L}_{\text{SAS}} \) formula over \( \hat{\mathcal{T}} \). Let \( C \) denote a concretisation function \( C: \mathcal{L}_{\text{SAS}}[\hat{\mathcal{T}}] \rightarrow \mathcal{L}_{\text{SAS}}[\mathcal{T}] \) between \( \hat{\mathcal{T}} \) and \( \mathcal{T} \).

The original SAS transition system \( \mathcal{T} \) and the abstract SAS transition system \( \hat{\mathcal{T}} \) are **consistently similar** with respect to \( \hat{\varphi} \), \( \mathcal{T} \preceq_{[\hat{\varphi}]} \hat{\mathcal{T}} \), iff

1. **Simulation:** there exists a simulation relation \( R \subseteq \Sigma \times \hat{\Sigma} \) such that \( \mathcal{T} \preceq \hat{\mathcal{T}} \)
2. **Consistency:** for all \( \hat{a} \in \text{Atoms}(\hat{\varphi}) \) if \( R(\sigma, \hat{\sigma}) \) it holds that \( (\hat{\mathcal{T}}, \hat{\sigma}) \models \hat{a} \iff (\mathcal{T}, \sigma) \models C(\hat{a}) \)

The original SAS transition system \( \mathcal{T} \) and the abstract SAS transition system \( \hat{\mathcal{T}} \) are **consistently bisimilar** with respect to \( \hat{\varphi} \), \( \mathcal{T} \cong_{[\hat{\varphi}]} \hat{\mathcal{T}} \), iff

1. **Bisimulation:** there exists a relation \( R \subseteq \Sigma \times \hat{\Sigma} \) such that \( \mathcal{T} \cong \hat{\mathcal{T}} \)
2. **Consistency:** for all \( \hat{a} \in \text{Atoms}(\hat{\varphi}) \) and \( R(\sigma, \hat{\sigma}) \), we have \( (\hat{\mathcal{T}}, \hat{\sigma}) \models \hat{a} \iff (\mathcal{T}, \sigma) \models C(\hat{a}) \)

In the literature, e.g. in [CGP99], systems are often modelled as Kripke structures \( \mathcal{K} = (S, S_0, \leadsto, L, AP) \) with a set of states \( S \), a set of initial states \( S_0 \), a transition relation \( \leadsto \subseteq S \times S \) and a labelling function \( L: S \rightarrow \mathcal{P}(AP) \) labelling states \( s \in S \) with atomic propositions in \( AP \) that are valid in that state. The fact that a property is preserved under the transformation of a Kripke structure is justified by simulation. For a simulation between two Kripke structures \( \mathcal{K} \) and \( \mathcal{K}' \), it is required that there exists a simulation relation that preserves initial states and transitions. Further, the atomic propositions of \( \mathcal{K}' \) have to be a subset of the atomic propositions of \( \mathcal{K} \), i.e. \( AP' \subseteq AP \), such that for two states \( s \) and \( s' \) in the simulation relation \( R(\sigma, \hat{\sigma}) \) the labelling of state \( s \), \( L(s) \), restricted to the atomic propositions \( AP' \) of the structure \( \mathcal{K}' \) coincides with the labelling of the state \( s' \), i.e. \( L(s) \cap AP' = L'(s') \).

For SAS, these requirements on the atomic propositions are reflected by the concretisation function. In contrast to Kripke structures, states of SAS are not labelled with atomic propositions. Instead, the validity of atomic
propositions in a state is derived from the valuation of the state’s variables. Hence, if the interpretation of variables and values is changed by a model transformation, also the set of atomic propositions that can be used for expressing properties over both systems is changed. The concretisation function (via the mapping $f$) associates atomic propositions over $\text{Var}$ and $\text{Val}$ in the transformed system with atomic proposition over $\text{Var}$ and $\text{Val}$ in the original system according to the performed transformation. This reflects that the atomic propositions of similar Kripke structures have to be in subset relation $\text{AP}' \subseteq \text{AP}$. The consistency condition for consistent (bi-)simulations in Definition 5.6 corresponds to the requirement that the labelling of related states of similar Kripke structures have to coincide, $L(s) \cap \text{AP}' = L'(s')$.

**Property Preservation on System Level**

The notions of consistent simulation and bisimulation can now be used to formulate when two systems have the same behaviours with respect to the validity of a property. By consistent simulation, properties of $\text{AL}_{\text{SAS}}$ are preserved under transformations. Liveness properties (using the existential path quantifier $\exists$) are lost under consistent simulation as we point out later. The proof of the property preservation theorem by consistent simulation proceeds by induction on the structure of $\text{AL}_{\text{SAS}}$ properties given in positive normal form. The consistency condition is used for the induction base of the proof when reasoning about atomic propositions and their negations. For temporal operators in the induction step, the corresponding path lemma (cf. Lemma 5.3) is used. The result is similar to the property preservation results on similar Kripke structures shown in [CGP99].

**Theorem 5.7 (Preservation of $\text{AL}_{\text{SAS}}$ State and Path Formulae by Consistent Simulation).** Let $T = (\Sigma, \text{Init}, \leadsto)$ and $\hat{T} = (\hat{\Sigma}, \hat{\text{Init}}, \hat{\leadsto})$ be two SAS transition systems. If there exists a simulation relation $R \subseteq \Sigma \times \hat{\Sigma}$ and a concretisation $\mathcal{C} : \text{AL}_{\text{SAS}}[\hat{T}] \rightarrow \text{AL}_{\text{SAS}}[T]$ such that $T$ and $\hat{T}$ form a consistent simulation with respect to a property $\hat{\phi} \in \text{AL}_{\text{SAS}}$, i.e. $T \preceq_{[\mathcal{C}]} \hat{T}$, then

- it holds for every $\text{AL}_{\text{SAS}}$ state formula, if $R(\sigma, \hat{\sigma})$, then $\langle \hat{T}, \hat{\sigma} \rangle \models \hat{\phi}$ implies $\langle T, \sigma \rangle \models C(\hat{\phi})$.
- it holds for every $\text{AL}_{\text{SAS}}$ path formula, if $\pi$ and $\hat{\pi}$ are corresponding paths, then $\langle \hat{T}, \hat{\pi} \rangle \models \hat{\phi}$ implies $\langle T, \pi \rangle \models C(\hat{\phi})$

**Proof.** Proof of by induction on the formula structure; for state formulae:
5.1. Property Preservation by System Simulation

- \( \hat{\phi} = \hat{a} \) where \( \hat{a} \) atomic: directly by consistency condition, as for all states \( R(\sigma, \hat{\sigma}) \) it holds that \( (\hat{T}, \hat{\sigma}) \models \hat{a} \iff (T, \sigma) \models C(\hat{a}) \).

- \( \phi = \neg \hat{a} \) where \( \hat{a} \) is atomic: directly by consistency condition, as for all states \( R(\sigma, \hat{\sigma}) \) it holds that \( (\hat{T}, \hat{\sigma}) \models \hat{a} \iff (T, \sigma) \models C(\hat{a}) \).

- \( \hat{\phi} = \varphi_1 \land \varphi_2 \): \( (\hat{T}, \hat{\sigma}) \models \hat{\phi} \) implies that \( (\hat{T}, \hat{\sigma}) \models \varphi_1 \) and that \( (\hat{T}, \hat{\sigma}) \models \varphi_2 \). By induction hypothesis, \( (T, \sigma) \models C(\varphi_1) \) and \( (T, \sigma) \models C(\varphi_2) \). This implies that \( (T, \sigma) \models C(\hat{\phi}) \).

- \( \hat{\phi} = \varphi_1 \lor \varphi_2 \): \( (\hat{T}, \hat{\sigma}) \models \hat{\phi} \) implies that \( (\hat{T}, \hat{\sigma}) \models \varphi_1 \) or that \( (\hat{T}, \hat{\sigma}) \models \varphi_2 \). By induction hypothesis, \( (T, \sigma) \models C(\varphi_1) \) or \( (T, \sigma) \models C(\varphi_2) \). This implies that \( (T, \sigma) \models C(\hat{\phi}) \).

- \( \hat{\phi} = A \varphi_1 \): By the semantics of \( A \), it holds that \( (T, \sigma_0) \models C(A\varphi_1) \) if for all paths \( \pi_0 \in Paths(T) \) starting in \( \sigma_0 \), \( (T, \pi_0) \models C(\varphi_1) \) holds. By Lemma 5.3, for each path \( \pi = \sigma_0, \sigma_1, \ldots \in Paths(T) \) there exists a corresponding path \( \hat{\pi} = \sigma_0, \hat{\sigma}_1, \ldots \in Paths(\hat{T}) \) such that \( R(\sigma_i, \hat{\sigma}_i) \) for \( i \geq 0 \). By assumption, we have that if \( (\hat{T}, \hat{\sigma}_0) \models \hat{\phi} \), then for all paths \( \hat{\pi} = \sigma_0, \hat{\sigma}_1, \ldots \), we have that \( (\hat{T}, \hat{\pi}) \models \varphi_1 \). By induction hypothesis on path formulae, we have that \( (T, \pi) \models C(\varphi_1) \) which yields \( (T, \sigma_0) \models C(A\varphi_1) \). Thus, we conclude \( T \models C(\hat{\phi}) \).

For path formulae:

- \( \hat{\phi} = \psi_1 \land \psi_2 \) where \( \psi_1, \psi_2 \) path formulae: \( (\hat{T}, \hat{\pi}) \models \hat{\phi} \) implies that \( (\hat{T}, \hat{\pi}) \models \psi_1 \) and that \( (\hat{T}, \hat{\pi}) \models \psi_2 \). By induction hypothesis, \( (T, \pi) \models C(\psi_1) \) and \( (T, \pi) \models C(\psi_2) \). This implies that \( (T, \pi) \models C(\hat{\phi}) \).

- \( \hat{\phi} = \psi_1 \lor \psi_2 \) where \( \psi_1, \psi_2 \) path formulae: \( (\hat{T}, \hat{\pi}) \models \hat{\phi} \) implies that \( (\hat{T}, \hat{\pi}) \models \psi_1 \) or that \( (\hat{T}, \hat{\pi}) \models \psi_2 \). By induction hypothesis, \( (T, \pi) \models C(\psi_1) \) or \( (T, \pi) \models C(\psi_2) \). This implies that \( (T, \pi) \models C(\hat{\phi}) \).

- \( \hat{\phi} = X \psi \): \( (\hat{T}, \hat{\pi}) \models \hat{\phi} \) implies that \( (\hat{T}, \hat{\pi}_1) \models \psi \). It holds that \( (T, \pi_0) \models C(X\psi) \) if \( (T, \pi_1) \models C(\psi) \). By Path Lemma 5.3, for each path \( \pi_1 \in Paths(T) \), there exists a corresponding paths \( \hat{\pi}_1 \). If \( (\hat{T}, \hat{\pi}_1) \models \psi \), by induction hypothesis, \( (T, \pi_1) \models C(\psi) \). This yields \( T \models C(\hat{\phi}) \).

- \( \hat{\phi} = \psi_1 U \psi_2 \): By assumption, \( (\hat{T}, \hat{\sigma}_0) \models \psi_1 U \psi_2 \). By definition of the until operator, there is a \( k \) such that \( (\hat{T}, \hat{\pi}_k) \models \psi_2 \), and for all \( 0 \leq j < k \),
(\hat{T}, \hat{\pi}_j) \models \psi_1. It holds that \((T, \pi_0) \models C(\psi_1 \cup \psi_2)\) if there is a \(k\) such that \(\pi_k \models C(\psi_2)\), and for all \(0 \leq j < k\), \(\pi_j \models C(\psi_1)\). By Lemma 5.3, for each path \(\pi \in Paths(T)\), there exists a corresponding path \(\hat{\pi} \in Paths(\hat{T})\), and in particular for each \(\pi_k\), there exists a corresponding \(\hat{\pi}_k\). Hence, by induction hypotheses, \((T, \pi_k) \models C(\psi_2)\), and for all \(0 \leq j < k\), \((T, \pi_j) \models C(\psi_1)\). This yields \(T \models C(\psi_1 \cup \psi_2)\) and, thus, \((T, \pi) \models C(\hat{\phi})\).

\(\bullet \ \hat{\phi} = \psi_1 \ R \ \psi_2\): By assumption, \((\hat{T}, \hat{\pi}) \models \psi_1 \ R \ \psi_2\). By definition of the \(R\) operator, either for all \(i\) it holds that \((\hat{T}, \hat{\pi}_i) \models \psi_1\), or there exists an \(n\) such that \((\hat{T}, \hat{\pi}_n) \models \psi_1\), and for all \(i < n\), \((\hat{T}, \hat{\pi}_i) \models \psi_2\). By Lemma 5.3, for each path \(\pi \in Paths(T)\), there exists a corresponding path \(\hat{\pi} \in Paths(\hat{T})\), and in particular for each \(\pi_i\), there exists a corresponding \(\hat{\pi}_i\). By induction hypothesis, either for all \(i\), it holds that \((T, \pi_i) \models C(\psi_2)\), or there exists an \(n\) such that \((T, \pi_n) \models C(\psi_1)\), and for all \(i < n\), \((T, \pi_i) \models C(\psi_2)\). Hence, \((T, \pi) \models C(\psi_1 \ R \psi_2)\) and \((T, \pi) \models C(\hat{\phi})\). \(\square\)

This theorem can be instantiated with the initial states which immediately yields the result that for two consistently similar SAS transition systems the concretisation of properties of \(\mathcal{AL}_{SAS}\) are preserved. The property over the original system follows from an implication of the concretisation.

**Theorem 5.8 (Property-Preservation of \(\mathcal{AL}_{SAS}\) by Consistent Simulation).** Let \(T = (\Sigma, \text{Init}, \sim)\) and \(\hat{T} = (\hat{\Sigma}, \text{Init}, \hat{\sim})\) be two SAS transition systems, \(\varphi\) an \(\mathcal{AL}_{SAS}\) formula over \(T\) and \(\hat{\varphi}\) an \(\mathcal{AL}_{SAS}\) formula over \(\hat{T}\). If there exists a simulation relation \(R \subseteq \Sigma \times \hat{\Sigma}\) and a concretisation \(C : \mathcal{AL}_{SAS}[\hat{T}] \to \mathcal{AL}_{SAS}[T]\) such that the following conditions hold:

1. Consistent simulation: \(T\) and \(\hat{T}\) form a consistent simulation with respect to \(\hat{\varphi}\), i.e. \(T \preceq_{[\hat{\varphi}] \hat{T}}\).
2. Implication: The concretisation of \(\hat{\varphi}\) implies the original property \(\varphi\), \(T \models C(\hat{\varphi}) \to \varphi\).

Then it holds that

\(\hat{T} \models \hat{\varphi}\) implies \(T \models \varphi\).

**Proof.** The consistent simulation yields that \(\hat{T} \models \hat{\varphi}\) implies \(T \models C(\hat{\varphi})\) immediately from Theorem 5.7, instantiated with the initial states. From the implication of the assumption, we conclude the overall theorem. \(\square\)

In order to see that existentially quantified path formulae are not preserved under consistent simulation we construct a counterexample: We have an original system \(T_S = (\Sigma, \text{Init}, \sim)\) where \(\Sigma = \{\sigma_1, \sigma_2\}\), \(\text{Init} = \{\sigma_1\}\) and \(\sim = \{(\sigma_1, \sigma_2), (\sigma_2, \sigma_2)\}\). The set of paths of \(T_S\) is Paths(\(T_S\) =
5.1. Property Preservation by System Simulation

\( \{ \pi = \sigma_1 \sigma_2 \ldots \} \). The transformed system is defined by \( T_{S'} = (\Sigma', \text{Init}', \sim') \) where \( \Sigma' = \{ \sigma'_1, \sigma'_2, \sigma'_3 \} \), \( \text{Init}' = \{ \sigma'_1 \} \) and \( \sim' = \{ (\sigma'_1, \sigma'_2), (\sigma'_2, \sigma'_3), (\sigma'_3, \sigma'_3) \} \). The set of paths is given by \( \text{Paths}(T_{S'}) = \{ \pi' = \sigma'_1 \sigma'_2 \ldots, \pi'' = \sigma'_1 \sigma'_3 \ldots \} \). The two systems are depicted in Figure 5.2 with the valid atomic propositions shown in each state. Note, that the transformed system is non-deterministic which is the main reason that existential properties are not preserved. Non-determinism can for instance be introduced by data domain abstractions as described in Section 5.3.

![Graphical Representation of Example Systems](image)

**Fig. 5.2.** Graphical Representation of Example Systems

The concrete system is simulated by the abstract system, \( T_S \preceq T_{S'} \), which can be shown as follows: Let \( \mathcal{R} \subseteq \Sigma \times \Sigma' \) denote the simulation relation with \( \mathcal{R} = \{ (\sigma_1, \sigma'_1), (\sigma_2, \sigma'_2) \} \). For the initial states of \( T_S \), we have \( \mathcal{R}(\sigma_1, \sigma'_1) \). For the transition \( \sigma_1 \sim \sigma_2 \), we have \( \mathcal{R}(\sigma_1, \sigma'_1) \), and there is \( \sigma'_2 \) such that \( \sigma'_1 \sim' \sigma'_2 \) and \( \mathcal{R}(\sigma_2, \sigma'_2) \). For the transition \( \sigma_2 \sim \sigma_2 \), we have \( \mathcal{R}(\sigma_2, \sigma'_2) \), and there is \( \sigma'_2 \) such that \( \sigma'_2 \sim' \sigma'_2 \) and \( \mathcal{R}(\sigma_2, \sigma'_2) \). As the atomic propositions in \( (\sigma_1, \sigma'_1) \) and \( (\sigma_2, \sigma'_2) \) are the same, the simulation is also consistent with the identity as concretisation function.

The abstract system satisfies the properties \( \text{AG}(y = 2) \) and \( \text{EF}(x = 4) \). However, for the concrete system it holds that \( T_S \models \text{AG}(y = 2) \), but \( T_S \not\models \text{EF}(x = 4) \). The reason for this lies in the corresponding paths. From Lemma 5.3, we know that for each path in \( T_S \) there is a corresponding path in \( T_{S'} \). In this example, \( \pi \) corresponds to \( \pi'' \). As \( \text{AG}(y = 2) \) holds on all paths of \( T_{S'} \), it in particular holds on the corresponding path \( \pi'' \). Hence, \( \text{AG}(y = 2) \) also holds in the concrete system. For the existential path property \( \text{EF}(x = 4) \), we have that \( (T_{S'}, \pi'') \models F(x = 4) \). However, there is no path in \( T_S \) that corresponds to \( \pi'' \). Hence, \( \text{EF}(x = 4) \) does not hold in the concrete system.
If two systems are bisimilar, we can show that full $L_{SAS}$ including existential properties is preserved. As we have seen in the example the reason for loosing validity of existential properties in simulations is that there may be paths in the transformed system that do not have a corresponding path in the original system, but contribute to existential validity. Since in bisimilar systems corresponding paths exist in both directions, also existential properties are preserved.

**Theorem 5.9 (Preservation of $L_{SAS}$ State and Path Formulae by Consistent Bisimulation).** Let $T = (\Sigma, \text{Init}, \sim)$ and $\hat{T} = (\hat{\Sigma}, \hat{\text{Init}}, \hat{\sim})$ be two SAS transition systems. If there exists a bisimulation relation $R \subseteq \Sigma \times \hat{\Sigma}$ and a concretisation $C : L_{SAS}[\hat{T}] \rightarrow L_{SAS}[T]$ such that $T$ and $\hat{T}$ form a consistent bisimulation with respect to $\hat{\phi}$, i.e. $T \cong [\hat{\phi}] \hat{T}$, then

- it holds for every $L_{SAS}$ state formula, if $R(\sigma, \hat{\sigma})$, then $(\hat{T}, \hat{\sigma}) \models \hat{\phi}$ iff $(T, \sigma) \models C(\hat{\phi})$.
- it holds for every $L_{SAS}$ path formula, if $\pi$ and $\hat{\pi}$ are corresponding paths, then $(\hat{T}, \hat{\pi}) \models \hat{\phi}$ iff $(T, \pi) \models C(\hat{\phi})$

**Proof.** The proof proceeds analogue to the proof of Theorem 5.7 by induction on the structure of the formula $\hat{\phi}$ using Lemma 5.4 for bisimilar systems.

For state formulae:

- $\hat{\phi} = \hat{a}$ where $\hat{a}$ atomic: directly by consistency condition for consistent bisimulation as for all states $\hat{\sigma}$ of $\hat{T}$ and $\sigma$ of $T$, it holds that if $R(\sigma_0, \hat{\sigma}_0)$ then $(\hat{T}, \hat{\sigma}) \models \hat{\phi}$ iff $(T, \sigma) \models C(\hat{\phi})$.
- $\hat{\phi} = \neg \varphi_1$, $\hat{\phi} = \varphi_1 \land \varphi_2$: immediately from induction hypothesis.
- $\hat{\phi} = E \varphi_1$: Assume $(\hat{T}, \hat{\sigma}) \models \hat{\phi}$. By semantics of $E$, there is a path $\hat{\pi} \in Paths(\hat{T})$ starting in $\hat{\sigma}$ such that $(\hat{T}, \hat{\pi}) \models \varphi_1$. By Path Lemma 5.4, for each path $\pi \in Paths(T)$ there exists a corresponding paths $\hat{\pi} \in Paths(\hat{T})$ and vice versa. Hence, by induction hypothesis on path formulae, there must be a path $\pi \in Paths(T)$ starting in $\sigma$ such that $(T, \pi) \models C(\varphi_1)$ which yields that $(T, \sigma) \models C(\hat{\phi})$ and vice versa.

For path formulae:

- $\hat{\phi} = \varphi_1$, where $\hat{\phi}$ is a path formula and $\varphi_1$ is a state formula, $\hat{\phi} = \neg \psi_1$ where $\psi_1$ path formula, $\hat{\phi} = \psi_1 \land \psi_2$ where $\psi_1, \psi_2$ are path formulae: immediately from induction hypothesis similar to the proof of Theorem 5.7.
- $\hat{\phi} = X \psi$: $(\hat{T}, \hat{\pi}) \models \hat{\phi}$ implies that $(\hat{T}, \hat{\pi}_1) \models \psi$. It holds that $(T, \pi_0) \models C(\psi)$ if $(T, \pi_1) \models C(\psi)$. By Path Lemma 5.4, for each path $\pi_1 \in Paths(T)$, there exists a corresponding paths $\hat{\pi}_1$ and vice versa.
5.1. Property Preservation by System Simulation

Hence, by induction hypothesis, \( (\hat{T}, \hat{\pi}_1) \models \psi \) iff \( (T, \pi_1) \models C(\psi) \). This yields \( T \models C(\hat{\phi}) \) and vice versa.

- \( \hat{\phi} = \psi_1 \cup \psi_2 \): By assumption, \( (\hat{T}, \hat{\pi}_0) \models \psi_1 \cup \psi_2 \). By definition of the until operator, there is a \( k \) such that \( (\hat{T}, \hat{\pi}_k) \models \psi_2 \), and for all \( 0 \leq j < k \), \( (\hat{T}, \hat{\pi}_j) \models \psi_1 \). It holds that \( (T, \pi_0) \models C(\psi_1 \cup \psi_2) \) if there is a \( k \) such that \( \pi_k \models C(\psi_2) \), and for all \( 0 \leq j < k \), \( \pi_j \models C(\psi_1) \). By Lemma 5.4, for each path \( \pi \in Paths(T) \), there exists a corresponding path \( \hat{\pi} \in Paths(\hat{T}) \), and in particular for each \( \pi_k \), there exists a corresponding \( \hat{\pi}_k \) and vice versa. Hence, by induction hypotheses, \( (\hat{T}, \hat{\pi}_k) \models \psi_2 \) iff \( (T, \pi_k) \models C(\psi_2) \), and for all \( 0 \leq j < k \), \( (\hat{T}, \hat{\pi}_j) \models \psi_1 \) iff \( (T, \pi_j) \models C(\psi_1) \). This yields that \( T \models C(\psi_1 \cup \psi_2) \) and vice versa. \( \Box \)

Instantiating this theorem for the initial states yields that for two bisimilar systems the concretisation of full logic is preserved.

**Theorem 5.10 (Preservation of \( L_{SAS} \) by Consistent Bisimulation).**

Let \( T = (\Sigma, \text{Init}, \leadsto) \) and \( \hat{T} = (\hat{\Sigma}, \hat{\text{Init}}, \hat{\leadsto}) \) be two SAS transition systems, \( \varphi \) an \( L_{SAS} \) formula over \( T \) and \( \hat{\varphi} \) an \( L_{SAS} \) formula over \( \hat{T} \). Then it holds that

\[
\hat{T} \models \hat{\varphi} \text{ iff } T \models \varphi
\]

if there exists a bisimulation relation \( R \subseteq \Sigma \times \hat{\Sigma} \) and a concretisation function \( C : L_{SAS}[\hat{T}] \rightarrow L_{SAS}[T] \) such that the following conditions hold:

1. Consistent Bisimulation: \( T \) and \( \hat{T} \) form a consistent bisimulation with respect to \( \hat{\varphi} \), \( T \cong_{[\hat{\varphi}]} \hat{T} \).
2. Equivalence: \( T \models C(\hat{\varphi}) \leftrightarrow \varphi \).

**Proof.** The consistent bisimulation yields that \( \hat{T} \models \hat{\varphi} \) iff \( T \models C(\hat{\varphi}) \) immediately from Theorem 5.9, instantiated with the initial states. From the equivalence of the concretisation \( C(\hat{\varphi}) \) and the original property \( \varphi \), we conclude the overall theorem. \( \Box \)

**Property Preservation on Module Level**

Similar to SAS transition systems, we define module transition systems from the local semantics of SAS modules. Module transition systems are required when transformations are carried out on module level, e.g. after the decomposition of a system into the contained modules (cf. Section 6.2.1). The results on property preservation by simulation and bisimulation, we have previously established for SAS transition systems, can be transferred directly to module transition systems as module and system transition systems are structurally equivalent.
Corollary 5.11 (Property Preservation on Module Level). Let $T_m = (S, \text{init}, \rightarrow)$ and $\hat{T}_m = (\hat{S}, \text{init}, \hat{\rightarrow})$ be two module transition systems. Then it holds for $\hat{\varphi}$, an $AL_{SAS}$ property over $\hat{T}_m$, and $\varphi$, an $AL_{SAS}$ property over $T_m$, that

$$(\hat{T}_m |= \hat{\varphi} \text{ implies } T_m |= \varphi) \iff (T_m \preceq_{[\varphi]} \hat{T}_m \text{ and } C(\hat{\varphi}) \rightarrow \varphi).$$

Further, it holds for $\hat{\varphi}$ an $L_{SAS}$ property over $\hat{T}_m$ and $\varphi$ an $L_{SAS}$ property over $T_m$ that

$$(\hat{T}_m |= \hat{\varphi} \text{ iff } T_m |= \varphi) \iff (T_m \equiv_{[\varphi]} \hat{T}_m \text{ and } C(\hat{\varphi}) \leftrightarrow \varphi).$$

Proof. Analogous to the proof of property preservation on system level in Theorem 5.7 and Theorem 5.9 because module transition systems are structurally equivalent. □

Consistent simulations between modules can be lifted to a consistent simulation between systems. Assume a system $SAS$ with the set of modules $M = \{m_1, \ldots, m_n\}$, and for each module $m_i$, there exists another module $m'_i$ such that both modules are in consistent simulation with respect to a property referring to the module variables of $m'_i$. Then, the system $SAS'$ containing the modules $m'_i$ is in consistent simulation with $SAS$ for a property $\varphi$ where $\varphi$ is defined over the module variables of the $m'_i$. A module $m_i$ that is similar to a module $m'_i$ can have different input and output variables. If we want to built the system $SAS'$ by replacing each module $m_i$ in $SAS$ with its similar module $m'_i$, this affects system variables and connections in $SAS$. The system $SAS'$ built from the similar modules $m'_i$ only contains system variables from $SAS$ that are connected to module inputs and outputs of the modules $m'_i$. The connections in $SAS'$ are the connections of $SAS$ restricted to module variables of the modules $m'_i$. The definition of the modules $m'_i$ has to ensure that no variable in $SAS'$ remains unconnected such that $SAS'$ is a valid system.

In order to define the system $SAS'$ formally, we need some notation about connected variables. With $Conn_S(Var_1, Var_2)$, we denote variables from a set $Var_1$ that are connected via adaptive or functional connections $conn_a$ or $conn_d$ as source variables to variables in the set $Var_2$. With $Conn_T(Var_1, Var_2)$, we denote variables from a set $Var_1$ that are connected via adaptive or functional connections $conn_a$ or $conn_d$ as target variables to variables in the set $Var_2$. The set $Conn(Var_1, Var_2)$ is the union of both sets.

Definition 5.12 (Variables in Connections). For an SAS system $SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)$, we define the set of
source variables \(Conn_S(Var_1, Var_2)\), the set of target variables \(Conn_T(Var_1, Var_2)\) and the set of source and target variables \(Conn(Var_1, Var_2)\) of connections between two sets of variables \(Var_1, Var_2 \subseteq Var(SAS)\) by

- \(Conn_S[SAS](Var_1, Var_2) = \{v \in Var_1 | \exists y \in Var_2. (v, y) \in (conn_a \cup conn_d)\}\)
- \(Conn_T[SAS](Var_1, Var_2) = \{v \in Var_1 | \exists y \in Var_2. (y, v) \in (conn_a \cup conn_d)\}\)
- \(Conn[SAS](Var_1, Var_2) = Conn_S(Var_1, Var_2) \cup Conn_T(Var_1, Var_2)\)

The following definition captures the construction of a system \(SAS'\) from a system \(SAS\) where \(SAS'\) contains the modules \(m'_i\) consistently similar to the modules \(m_i\) in \(SAS\). Let \(Var(M')\) denote the variables of the modules \(m'_i\), i.e. \(Var(M') = \bigcup_{i=1}^{n} Var(m'_i)\).

**Definition 5.13 (System Construction from Similar Modules).** Let \(SAS = (\{m_1, \ldots, m_n\}, input_d, input_d, output_a, output_d, conn_a, conn_d)\) be a system. For each \(1 \leq i \leq n\), let \(m'_i\) be a module such that \(m_i \succeq [\varphi]^{m'} \) for a property \(\varphi\) defined over \(Var(m'_i)\). We define the system \(SAS'\) containing the similar modules \(m'_1, \ldots, m'_n\) as

\[SAS' = (\{m'_1, \ldots, m'_n\}, input'_a, input'_d, output'_a, output'_d, conn'_a, conn'_d)\]

where

- \(input'_a = input_a \cap Conn_S[SAS](input_a, Var(M'))\)
- \(input'_d = input_d \cap Conn_S[SAS](input_d, Var(M'))\)
- \(output'_a = output_a \cap Conn_T[SAS](output_a, Var(M'))\)
- \(output'_d = output_d \cap Conn_T[SAS](output_d, Var(M'))\)
- \(conn'_a = conn_a \cap ((Var(M') \cup input'_a) \times (Var(M') \cup output'_a))\)
- \(conn'_d = conn_d \cap ((Var(M') \cup input'_d) \times (Var(M') \cup output'_d))\)

if \(conn'_a\) and \(conn'_d\) are total such that no variable is unconnected.

The proof that \(SAS\) and \(SAS'\) are consistently similar with respect to \(\varphi\) proceeds by extending the simulation relations and concretisations existing on module level by assumption to a relation and a concretisation on system level and showing that it constitutes a consistent simulation with respect to a property \(\varphi\). Note, that \(\varphi\) has to be defined over the variables of \(SAS'\) such that each atomic proposition refers only to one module.

**Theorem 5.14 (Consistent Simulation by Similar Modules).** Let \(SAS = (\{M_1, \ldots, M_n\}, input_d, input_d, output_a, output_d, conn_a, conn_d)\) be a system, and for each \(1 \leq i \leq n\), let \(m'_i\) be a module such that \(T_{m_i} \succeq \T{\varphi}_i \ T_{m'_i}\) for a property \(\varphi_i \in AC_{SAS}\alue\) expressible over \(Var(m_i)\). Let \(\psi\) be an \(AC_{SAS}\alue\) property
such that for all \( a \in \text{Atoms}(\psi) \) it holds that \( a \in \bigcup_{i=1}^{n} \text{Atoms}(\varphi_i) \). Then it holds for \( \text{SAS}' \), the system containing \( \{m'_1, \ldots, m'_n\} \) as in Definition 5.13, that \( T_{\text{SAS}} \preceq_{[\psi]} T_{\text{SAS}'} \).

**Proof.** Let \( R_i \subseteq S_i \times S_i' \) be the simulation relation on the i-th module pair and \( C_i \) the concretisation on the i-th module pair. We define \( R \subseteq \Sigma \times \Sigma' \) as a relation on \( \text{SAS} \) and \( \text{SAS}' \) with \( (\sigma, \sigma') \in R \) iff \( (\sigma|_{\text{Var}(m)}, \sigma'|_{\text{Var}(m')}) \in R_i \), and for \( a \in \text{Atoms}(\psi) \), we have \( C(a) = C_i(a) \) if \( a \in \text{Atoms}(\varphi_i) \). Now, we show that \( R \) is a consistent simulation on \( \text{SAS} \) and \( \text{SAS}' \) with respect to \( \psi \).

- **Initial Simulation:** Let \( \sigma_0 \in \text{Init} \) be an initial state of \( \text{SAS} \). For each module \( m_i \), it holds by assumption that for \( \sigma_0|_{\text{Var}(m_i)} \) there is a state \( \sigma'_0 \in \text{Init}' \) such that \( \sigma'_0|_{\text{Var}(m'_i)} \) is initial and that \( (\sigma_0|_{\text{Var}(m_i)}, \sigma'_0|_{\text{Var}(m'_i)}) \in R_i \). By definition of \( R \), this yields \( (\sigma_0, \sigma'_0) \in R \).

- **Step Simulation:** Let \( \sigma_i \sim \sigma_{i+1} \) in \( \text{SAS} \) and \( (\sigma_i, \sigma'_i) \in R \). Then by definition of the SAS semantics, we have that for all \( i \) it holds \( \sigma_i|_{\text{Var}(m_i)} \rightarrow \sigma_{i+1}|_{\text{Var}(m_i)} \). By definition of \( R \), we have that for each module \( m_i \) there is a state \( \sigma'_i \) such that \( \sigma'_i|_{\text{Var}(m_i)} \rightarrow \sigma'_{i+1}|_{\text{Var}(m'_i)} \). Then we can conclude by definition of the global SAS semantics that there is a global transition \( \sigma'_i \sim \sigma'_{i+1} \) and \( (\sigma_{i+1}, \sigma'_{i+1}) \in R \).

- **Consistency:** Let \( (\sigma, \sigma') \in R \) and \( \sigma \models a \) for \( a \in \text{Atoms}(\varphi_i) \). Assume that \( \varphi_i \in A\mathcal{L}_{\text{SAS}} \). Then by definition of \( R \) and by assumption on \( R_i \), we know that \( \sigma'|_{\text{Var}(m_i)} \models a \) iff \( \sigma|_{\text{Var}(m_i)} \models C_i(a) \). Hence, \( \sigma \models C(a) \) and vice versa.

The same result also holds for bisimilar modules. A system \( \text{SAS} \) is bisimilar to a system \( \text{SAS}' \) if for each module of \( \text{SAS} \) there is exactly one module in the other system \( \text{SAS}' \) such that both modules are bisimilar. The proof is done by extending the bisimulation relation that is given for the module pairs to the system level.

**Theorem 5.15 (Consistent Bisimulation by Bisimilar Modules).** Let \( \text{SAS} = (\{M_1, \ldots, M_n\}, \text{input}, \text{output}, \text{output}_d, \text{conn}, \text{conn}_d) \) be a system, and for each \( 1 \leq i \leq n \), let \( m'_i \) be a module such that \( T_{m_i} \cong_{[\psi]} T_{m'_i} \) for a property \( \varphi_i \in A\mathcal{L}_{\text{SAS}} \) expressible over \( \text{Var}(m_i) \). Let \( \psi \) be an \( A\mathcal{L}_{\text{SAS}} \) property such that for all \( a \in \text{Atoms}(\psi) \), it holds that \( a \in \bigcup_{i=1}^{n} \text{Atoms}(\varphi_i) \). Then it holds for \( \text{SAS}' \), the system containing \( \{m'_1, \ldots, m'_n\} \) as in Definition 5.13, that \( T_{\text{SAS}} \cong_{[\psi]} T_{\text{SAS}'} \).

**Proof.** Analog to the proof of Theorem 5.14, by extending the bisimulation relation on the module pairs to system level and showing that it constitutes a bisimulation. \( \square \)
5.1.2 Translation Validation for Model Transformations

In order to use model transformations for verification complexity reduction in our integration framework, we have to make sure that for a concrete model instance and a concrete property the performed transformation is property-preserving. This allows showing the transformed property over the transformed model and inferring that the original model satisfies the original property. In principle, there are two approaches to guarantee correctness of model transformations: The transformation algorithms (and their implementations!) can be verified once and for all [Ler06]. Alternatively, the results of each distinct run of the transformation procedure can be proven correct. Our translation validation technique [BSPH07] uses the second approach and is inspired by a translation validation [PSS98] based approach for compilers [GBP06, BPH07, PHG05]. In the area of compiler verification, it has turned out that runtime verification of compilers is often the method of choice for achieving guaranteed correct compilation results. As for compilers, correctness proofs for distinct model transformations are usually less complex and easier to establish than proofs for a general transformation procedure. An additional advantage is that the transformation procedure can be tailored to a particular model and property under consideration and thus match the requirements of the concrete problem very closely. Also note, that in our approach the correctness of a transformation is proven formally using a theorem prover instead of a paper-and-pencil-proof.

The overall structure of our translation validation approach [BSPH07] for model transformations is depicted in Figure 5.3. For verifying a transformation, the transformation procedure is given an original system model and a property to be checked. As output a transformed system model with a corresponding transformed property is produced. Furthermore, a proof script is generated capturing the actual proof that the transformation preserves the considered property. In this direction, a correctness criterion for model transformations based on consistent simulation between the two systems is formalized. This criterion is checked for the considered system models and properties in a theorem prover, Isabelle/HOL [NPW02] in our implementation, using the generated proof script. Thus, the correctness of a transformation is verified for each distinct run of the transformation procedure. Note that the correctness of the technique does not depend on the proof script provided. An incorrect proof script may never lead to an incorrect proof but rather to no proof at all.
In order to implement translation validation in the theorem prover Isabelle/HOL, the following tasks have to be performed: First, we need a description of both the original and the transformed system model in Isabelle/HOL. Second, the original and the transformed property have to be represented in Isabelle. Then, we need a formalisation of the criterion when a transformation is considered as correct. This criterion is the Isabelle representation of the conditions stated in Theorem 5.8. Finally, we need a proof script that proves that the original and transformed model representations and property descriptions fulfill the correctness criterion.

System and Property Representations in Isabelle

In our translation validation approach, Isabelle representations of original and transformed system models are generated right before and after the model transformation process. In principle, transformations of SAS modules by data domain abstraction presented in Section 5.3 can lead to non-deterministic SAS modules. The non-determinism is due to configuration guards and conditions in conditional assignments evaluating to an unknown value. This results in a set of possible successor states. In our practical examples, however, we observed that in suitable abstractions the conditions in guards and conditional assignments do not evaluate to the unknown value because this leads to a loss of precision with respect to verification. Hence, it suffices to use a deterministic representation of module transition functions for translation validation of model transformations. This simplifies reasoning about simulations between systems to a great extend. Original and transformed system models are represented using the same data types as described in Section 4.2.

Properties over SAS models are expressed in $\mathcal{L}_{SAS}$ both for the original and the transformed property. A transformation of SAS models and
properties is property-preserving if the simulation between two systems is consistent with the atomic propositions of the transformed property and its concretisation. Hence, it suffices to consider the atomic propositions of \( \varphi \in L_{SAS} \) in our translation validation approach. Atomic propositions \( r(expr_1, expr_2) \in Atoms(\varphi) \) are represented in terms of Isabelle semantics as for the shallow embedding of \( L_{SAS} \) properties in Section 4.2.

**Formalising Transformation Correctness in Isabelle**

To prove that a transformed model preserves a property of an original model, we need a formalisation of property preservation in Isabelle/HOL. Figure 5.4 shows the correctness criterion formalised in Isabelle corresponding to Theorem 5.8. It depends on the transition functions of original and transformed system, on the initial states of both systems and on a simulation relation over the set of states of both systems. The simulation relation can be generated by the transformation procedure or adjusted by hand to reflect the performed transformation. The function systemequivalence evaluates to true if the considered transformation is property-preserving. The first condition in the formalisation requires that initial states are related by the simulation relation \( R \), and the second requires that \( R \) preserves the transitions of transformed and original system. Both conditions are formalised once and for all systems. The further requirements on \( R \) corresponds to the consistency of the simulation relation \( R \). The requirements for consistency differ for each concrete transformation applied and ensure that the concretisation of the transformed property implies the original property.

**Proving Transformations Correct**

To conduct the correctness proof, we need a proof script establishing that for an original system and a transformed system systemequivalence

```isabelle
constdefs systemequivalence ::
(state \Rightarrow state) \Rightarrow (state' \Rightarrow state') \Rightarrow state \Rightarrow state' \Rightarrow
(state \Rightarrow state' \Rightarrow bool) \Rightarrow bool
"systemequivalence nextstate nextstate' s0 s0' R ==
R s0 s0' \&
\forall s s'. R s s' --> R (nextstate s) (nextstate s') \&
\text{further requirements on } R"
```

Fig. 5.4. Correctness Criterion in Isabelle
evaluates to true. In our implementation, we first prove additional lemmas that imply the actual correctness theorem. Such an auxiliary lemma aux, generic for a class of systems, is depicted in Figure 5.5. The predicates funequiv and inputequiv over system states s and s’ together imply system equivalence and in general highly depend on the choice of the simulation relation. In the proof process, the theorem prover symbolically evaluates the state transition functions transition and transition’ of original and transformed system, respectively, on symbolic states s and s’. The states are only specified by the requirements of the simulation relation. In the proof of the lemma, it is checked that if the relation (funequiv s s’) between two states holds before a transition, it is still true after the transition. The predicate (funequiv’ s s’) is necessary for technical reasons. In the examples examined so far, we have developed a single highly generic proof script that establishes transformation correctness in all example scenarios. For more complicated scenarios, the proof script might need adaptation which could be done fully automatically in case of translation validation for compilers [BPH07].

```
lemma aux: "(funequiv s s') \& (inputequiv s s') \& (funequiv' s s')
          --> (funequiv (transition s s) (transition' s' s'))"
apply clarify
apply (unfold funequiv_def inputequiv_def)
apply clarify
apply (unfold transition_def)
apply (unfold transition'_def)
apply (erule subst)+
apply (unfold funequiv'_def funequiv_def inputequiv_def)
apply clarify
apply (rule conjI, simp)+
apply simp
done
```

Fig. 5.5. Proof Script for Translation Validation

5.2 Model Reduction by Slicing

In order to use model checking for automatic verification of model properties, verification complexity introduced by the employed modelling concepts has to be alleviated. A promising technique bridging the gap between intuitive modelling concepts and input for formal verification is automatic
Slicing [Wei84, Tip95] was originally introduced as static program analysis for simplifying sequential programs for debugging and program understanding. A program slice is the projection of a program to relevant parts with respect to a slicing criterion. The resulting program is therefore in general simpler and easier to understand. Slicing algorithms for programs traditionally identify a slicing criterion with a variable at a point in the program dependency graph. From this, iteratively a transitive set of statements and variables influencing (backward slicing) or being influenced by this point (forward slicing) is computed. Afterwards, the part of the program not included in the slice can be safely removed. Recently, slicing has also been transferred to model reduction countering the state-explosion problem in model checking. In these approaches, slicing has been applied mainly to low-level transition system representations, such as extended hierarchical automata [WDQ02], or the input language of a model checker [HDZ00, MT00b]. In contrast, we use slicing on SAS models, a high-level representation of the MARS modelling concepts, before the actual verification input is generated [SPH08].

For reducing SAS models to the parts relevant for a property, we perform property-directed static backward slicing. This means that the property to be verified over a system model serves as slicing criterion. We identify it with the variables contained in the property as the validity of a property is determined by the values of the referenced variables. A slice of a system is computed by analysing which system parts transitively influence the slicing criterion. While there exist several slices with respect to a given slicing criterion, we construct the smallest slice that can be obtained from the slicing criterion and that removes as much irrelevant parts of the model as possible.

Slicing of high-level SAS models allows exploiting structural system information in the analysis of irrelevant system parts with respect to a property. SAS models can be reduced by analyses in different levels of detail. This allows a fine-grained control over required analyses and possible reductions for the applied verification tools. First, slicing of SAS models can be performed on system level by considering only the module a property variable belongs to and transitively the modules influencing this module based on the system connections. Second, the behaviour inside the modules can be analysed in more detail such that also inside the modules irrelevant
parts can be removed. This analysis considers SAS function descriptions introduced in Definition 3.11. Third, if the property only refers to adaptation behaviour, the functional behaviour of all modules can be removed since in SAS adaptation and functional behaviour are clearly separated. The correctness of the slicing algorithms is justified by showing that slicing induces a consistent bisimulation with respect to the considered property. Thus, it suffices to verify the property over the reduced system in order to conclude that the original system satisfies it. Furthermore, for a concrete system and its reduction by slicing, property preservation can be established using the translation validation approach presented in Section 5.1.2. In this section, we present slicing of SAS models in all three levels of granularity, i.e. system slicing, module slicing and adaptive slicing, and discuss how these techniques can be combined for enhancing model reductions.

5.2.1 System Slicing

An SAS model is composed from a set of interconnected modules. A property over this model may only refer to a subset of these modules. Hence, all modules that are not transitively connected to the modules concerned with the property do not influence its validity and can safely be removed. In this direction, system slicing reduces the system by exploting the mere structure of the system model. For a given property, we determine the set of modules that are concerned with this property. These are the modules whose variables are referenced directly in the property. This set of modules serves as slicing criterion for system slicing. From the initial set of modules, we compute the modules of the system that influence their behaviour according to the connections in the system. Iteratively, we compute the transitive closure of the modules connected to the inputs of the modules already encountered until we reach a fixpoint. Then, we have obtained a slice of the system that is only relevant for the validity of the considered property.

For ease of presentation, we assume that the property to be verified only contains variables of modules contained in the system and no system border variables from $\text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d$. Using Lemma 3.17, any property can be transformed into an equivalent property over the module variables. For a SAS containing the modules $\{m_1, \ldots, m_n\}$, we define the set of modules concerned with a property as the smallest set $M_\varphi \subseteq \{m_1, \ldots, m_n\}$ such that $\text{Var}(\varphi) \subseteq \text{Var}(M_\varphi)$. We define the modules
transitively connected to the inputs of the modules $M_\varphi$ as a system slice. A module $m_j$ is connected to a module $m_i$ denoted by $\text{Connected}(m_j, m_i)$ if there exists a connection $(v_j, v_i) \in \text{conn}_a \cup \text{conn}_d$ such that $v_j \in \text{out}_j \cup \text{adapt}_\text{out}_j$ and $v_i \in \text{in}_i \cup \text{adapt}_\text{in}_i$.

**Definition 5.16 (System Slice).** Let $\text{SAS}$ be a system $\text{SAS} = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d)$ and $\varphi$ an $\mathcal{L}_\text{SAS}$ property over $\text{Var}(M)$. We define the system slice $\text{Slice}_\varphi(\text{SAS})$ with respect to $\varphi$ as the smallest fixpoint satisfying the following conditions:

- $M_\varphi \subseteq \text{Slice}_\varphi(\text{SAS})$
- $m_j \in \text{Slice}_\varphi(\text{SAS})$ if there exists $m_i \in \text{Slice}_\varphi(\text{SAS})$ such that $\text{Connected}(m_j, m_i)$.

As an example consider the SAS model depicted in Figure 5.6 with four modules $\{m_1, m_2, m_3, m_4\}$. The arrows show the connections between the modules, where the dashed arrows are adaptive connections. The grey area represents the system slice computed for a property referring only to $m_3$.

![Fig. 5.6. Example of System Slicing](image)

Using a system slice $\text{Slice}_\varphi$, we define a reduced system $\text{SAS}|_{\text{Slice}_\varphi}$ only containing the modules included in the slice. On system level, all modules not included in the slice are removed and also all system variables connected to the removed modules together with the respective connections. The remaining modules remain unchanged. Only for module output variables $v$ connected to modules not in the slice, a new system variable $v_{\text{sys}}$ is added to $\text{output}_a$ or $\text{output}_d$ together with a corresponding connection in $\text{conn}_a$ or $\text{conn}_d$ to maintain a valid system since no module input or output variable may be unconnected.
**Definition 5.17 (System Slicing).** Let \( \text{SAS} \) be a system and \( \varphi \) an \( \mathcal{L}_{\text{SAS}} \) property over \( \text{Var}(\text{M}) \). We define

\[
\text{SAS}|_{\text{Slice}_\varphi} = (M|_{\text{Slice}_\varphi}, \text{input}_a|_{\text{Slice}_\varphi}, \text{input}_d|_{\text{Slice}_\varphi}, \text{output}_a|_{\text{Slice}_\varphi}, \text{output}_d|_{\text{Slice}_\varphi}, \text{conn}_a|_{\text{Slice}_\varphi}, \text{conn}_d|_{\text{Slice}_\varphi})
\]

as the system reduced with respect to \( \text{Slice}_\varphi(\text{SAS}) \) with

\[
\begin{align*}
M|_{\text{Slice}_\varphi} &= \text{Slice}_\varphi(\text{SAS}) \\
\text{input}_a|_{\text{Slice}_\varphi} &= \text{input}_a \cap \text{Conn}_a[\text{SAS}](\text{Var}(\text{SAS}), \text{Var}(M|_{\text{Slice}_\varphi})) \\
\text{input}_d|_{\text{Slice}_\varphi} &= \text{input}_d \cap \text{Conn}_d[\text{SAS}](\text{Var}(\text{SAS}), \text{Var}(M|_{\text{Slice}_\varphi})) \\
\text{output}_a|_{\text{Slice}_\varphi} &= (\text{output}_a \cap \text{Conn}_a[\text{SAS}][\text{Var}(\text{SAS}), \text{Var}(M|_{\text{Slice}_\varphi})) \\
&\quad \cup \{v_{\text{sys}} \mid v \in \text{Conn}_a(\text{SAS})\} \\
\text{output}_d|_{\text{Slice}_\varphi} &= (\text{output}_d \cap \text{Conn}_d[\text{SAS}][\text{Var}(\text{SAS}), \text{Var}(M|_{\text{Slice}_\varphi})) \\
&\quad \cup \{v_{\text{sys}} \mid v \in \text{Conn}_d(\text{SAS})\} \\
\text{conn}_a|_{\text{Slice}_\varphi} &= (\text{conn}_a \cap (\text{Conn}_a|_{\text{Slice}_\varphi} \times \text{Conn}_d|_{\text{Slice}_\varphi})) \\
&\quad \cup \{(v, v_{\text{sys}}) \mid v \in \text{Conn}_a(\text{SAS})\} \\
\text{conn}_d|_{\text{Slice}_\varphi} &= (\text{conn}_d \cap (\text{Conn}_a|_{\text{Slice}_\varphi} \times \text{Conn}_d|_{\text{Slice}_\varphi})) \\
&\quad \cup \{(v, v_{\text{sys}}) \mid v \in \text{Conn}_d(\text{SAS})\}
\end{align*}
\]

where

\[
\overline{\text{Conn}_a}(\text{SAS}) = \{v \in \text{Var}_{\text{adapt, out}}(M|_{\text{Slice}_\varphi}) \mid \\
\neg \exists w \in \text{Var}(M|_{\text{Slice}_\varphi}) \cup \text{output}_a \text{ with } \text{conn}_a(v, w)\}
\]

and

\[
\overline{\text{Conn}_d}(\text{SAS}) = \{v \in \text{Var}_{\text{out}}(M|_{\text{Slice}_\varphi}) \mid \neg \exists w \in \text{Var}(M|_{\text{Slice}_\varphi}) \cup \text{output}_d \text{ with } \text{conn}_d(v, w)\}
\]

denote the sets of unconnected adaptive and functional module variables, respectively, and \( \text{Conn}|_{\text{Slice}_\varphi} = \text{Conn}[\text{SAS}][\text{Var}(\text{SAS}), \text{Var}(M|_{\text{Slice}_\varphi})] \) denotes the variables in \( \text{SAS} \) connected to module variables in the system slice \( \text{Slice}_\varphi(\text{SAS}) \).

In order establish that this reduction is property preserving, we have to show that the original \( \text{SAS} \) and the reduced \( \text{SAS}|_{\text{Slice}_\varphi} \) are consistently bisimilar with respect to \( \varphi \). For the proof, we choose a bisimulation relation where the global states of the original and of the reduced system coincide on the variables of the modules contained in the slice. The concretisation is the identity function since system slicing does not change the interpretation of variables and values. As the remaining variables of the original system cannot influence the values of the other variables by construction of the slice, initial states, transitions and atomic propositions are preserved by the relation.
Theorem 5.18 (Consistent Bisimulation for System Slicing). Let \( SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d) \) be a system and \( \varphi \) an \( \mathcal{L}_{SAS} \) property over \( \text{Var}(M) \) and \( SAS|_{\text{Slice}_\varphi} \) as defined in Definition 5.17. Then it holds that the transition systems induced by \( SAS \) and \( SAS|_{\text{Slice}_\varphi} \) form a consistent bisimulation with respect to \( \varphi \):

\[
T_{SAS} \cong [\varphi] T_{SAS|_{\text{Slice}_\varphi}}
\]

**Proof.** We have to show that \( T_{SAS} \) and \( T_{SAS|_{\text{Slice}_\varphi}} \) are consistently bisimilar with respect to \( \varphi \). Let \( R \subseteq \Sigma_{SAS} \times \Sigma_{SAS|_{\text{Slice}_\varphi}} \) denote the bisimulation relation, where \( R(\sigma, \sigma|_{\text{Slice}_\varphi}) \) iff for all \( v \in \text{Var}(\text{Slice}_\varphi(SAS)) \) it holds that \( \sigma(v) = \sigma|_{\text{Slice}_\varphi}(v) \). Let \( C(\varphi) = \varphi \) be the identity function.

1. Initial simulation: Let \( \sigma_0 \) be an initial state of \( SAS \) and \( \sigma_0|_{\text{Slice}_\varphi} \) an initial state of \( SAS|_{\text{Slice}_\varphi} \). It is clear by construction of \( SAS|_{\text{Slice}_\varphi} \) that for all \( \sigma_0 \) and \( \sigma_0|_{\text{Slice}_\varphi} \), it holds that \( R(\sigma_0, \sigma_0|_{\text{Slice}_\varphi}) \).

2. Step simulation (1): Let \( \sigma_i \leadsto_{SAS} \sigma_{i+1} \) and \( R(\sigma_i, \sigma_i|_{\text{Slice}_\varphi}) \). By construction of \( \text{Slice}_\varphi \), there exists \( \sigma_{i+1}|_{\text{Slice}_\varphi} \) such that \( \sigma_i|_{\text{Slice}_\varphi} \leadsto_{SAS} \sigma_{i+1}|_{\text{Slice}_\varphi} \) and \( R(\sigma_{i+1}, \sigma_{i+1}|_{\text{Slice}_\varphi}) \).

3. Step simulation (2): Let \( \sigma_i|_{\text{Slice}_\varphi} \leadsto_{\text{Slice}_\varphi} \sigma_{i+1}|_{\text{Slice}_\varphi} \) and \( R(\sigma_i, \sigma_i|_{\text{Slice}_\varphi}) \). By construction of \( \text{Slice}_\varphi \), there exists \( \sigma_{i+1} \) such that \( \sigma_i \leadsto_{SAS} \sigma_{i+1} \) and \( R(\sigma_{i+1}, \sigma_{i+1}|_{\text{Slice}_\varphi}) \).

4. Consistency: As \( \text{Var}(\varphi) \subseteq \text{Var}(M) \subseteq \text{Var}(\text{Slice}_\varphi) \), we have for all \( a \in \text{Atoms}(\varphi) \) if \( R(\sigma, \sigma|_{\text{Slice}_\varphi}) \) that \( \sigma \models a \) iff \( \sigma|_{\text{Slice}_\varphi} \models a \).

By Theorem 5.10, it holds that properties expressible in \( \mathcal{L}_{SAS} \) are preserved under system slicing. This allows showing a property \( \varphi \) over the system reduced by system slicing with respect to \( \varphi \) in order to establish its validity over the original system.

**Corollary 5.19 (Property Preservation under System Slicing).** Let \( SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d) \) be a system and \( \varphi \) an \( \mathcal{L}_{SAS} \) property over \( \text{Var}(M) \) and \( SAS|_{\text{Slice}_\varphi} \) as in Definition 5.17. Then it holds that:

\[
T_{SAS|_{\text{Slice}_\varphi}} \models \varphi \iff T_{SAS} \models \varphi
\]

**Proof.** Immediately from Theorem 5.18 and Theorem 5.10.

### 5.2.2 Module Slicing

Slicing can also be performed on intra-module level in order to remove irrelevant parts with respect to a property inside a module. To this end, it...
has to be analysed which assignments in the transition functions of a module influence a variable in a property. This restricts module slicing to SAS models where functionality is fully specified by functional descriptions.

The adaptive transition functions \( \text{adapt}_{\text{next\_state}} \) and \( \text{adapt}_{\text{next\_out}} \) and the functional transition functions of each configuration \( \text{next\_state}_j \) and \( \text{next\_out}_j \) are expressed by a list of unconditional and conditional assignments of variables to expressions (cf. Definition 3.11). In these syntactic function descriptions, we distinguish data dependence, control dependence and adaptive dependence between variables. Data dependence \( \mathcal{DD} \) describes that a variable in an assignment is influenced by the variables in the assigned expression. Control dependence \( \mathcal{CD} \) means that a variable assigned in a conditional assignment depends on the variables used in the conditions because those variables control which branch of the conditional assignment is selected. Adaptive dependence \( \mathcal{AD} \) is a special dependence relation for adaptive systems. A functional variable is influenced by the variables occurring in the configuration guards. Those variables determine which configuration is activated and therefore select the functional assignments to be executed. Hence, for each functional variable assigned in a configuration, also the variables of the configuration guards belong to the influencing variables. As SAS are synchronous systems, there are no other forms of dependencies with respect to synchronisation or communication. The following definition captures the notions of data and control dependencies in an SAS function description formally.

**Definition 5.20 (Data and Control Dependencies).** For a function description \( f \) given as a list of assignments \( f = [a_1, \ldots, a_n] \in \text{Func} \) and \( v \in \text{Var} \), we have:

- **Data Dependence:** \( \mathcal{DD}(v, f) = \mathcal{DD}(v, a_i) \) if \( a_i \) is an assignment to \( v \), else \( \mathcal{DD}(v, f) = \emptyset \)
- **Control Dependence:** \( \mathcal{CD}(v, f) = \mathcal{CD}(v, a_i) \) if \( a_i \) is an assignment to \( v \), else \( \mathcal{CD}(v, f) = \emptyset \)

For \( a_i = (v := \text{expr}) \) an unconditional assignment to \( v \), we have:

- **Data Dependence:** \( \mathcal{DD}(v, a_i) = \text{Var}(\text{expr}) \)
- **Control Dependence:** \( \mathcal{CD}(v, a_i) = \emptyset. \)

For \( a_i = (v := [(c_1, \text{expr}_1), \ldots, (c_m, \text{expr}_m)]) \) a conditional assignment to \( v \), we have:

- **Data Dependence:** \( \mathcal{DD}(v, a_i) = \bigcup_{i=1, \ldots, m} \text{Var}(\text{expr}_i) \)
- **Control Dependence:** \( \mathcal{CD}(v, a_i) = \bigcup_{i=1, \ldots, m} \text{Var}(c_i). \)
In an SAS module $m$, a variable $v \in \text{Var}(m)$ can be assigned only once in each cycle according to the synchronous semantics of SAS. For determining the influences for $v$, we have to distinguish whether it is a functional or an adaptive variable. A functional state variable $v \in \text{loc}$ can be assigned in the $\text{next}_\text{state}_j$ function, a functional output variable $v \in \text{out}$ in the $\text{next}_\text{out}_j$ function of a configuration $j$. For a functional state or output variable, we take the transition functions of the module configurations and determine the data, control and adaptive dependencies for obtaining the influencing variables. If $v$ is an adaptive variable, it can be assigned in the adaptive $\text{adapt}_\text{next}_\text{state}$ or $\text{adapt}_\text{next}_\text{out}$ function, respectively. Analogue to the functional variables, we add variables in data and control dependency to the set of influencing variables. Adaptive dependence is irrelevant for adaptive variables as they are set independently of the selected configuration. Any input variable is set by the environment and not influenced by the module itself. The variable for representing the used configuration $\text{useconf}$ in a module $m$ has to be treated separately as the used configuration depends on the configuration guards. For a $\text{useconf}$ variable, we consider adaptive dependencies and add all variables occurring in the configuration guards to the influences.

**Definition 5.21 (Module Dependencies).** Let $\text{Descr}[m]$ be a syntactic SAS description $\text{Descr}[m] = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{Descr}[\text{confs}], \text{Descr}[\text{adapt}])$ for a module $m$. We define the set of influencing variables $\text{infl}(v)$ for a variable $v \in \text{Var}(m)$ by:

- For an input variable $v \in \text{in} \cup \text{adapt}_\text{in}$, $\text{infl}(v) = \emptyset$.
- For a functional state variable $v \in \text{loc}$,

$$\text{infl}(v) = \bigcup_{j=1,\ldots,n} \text{DD}(v, \text{Descr}[\text{next}_\text{state}_j]) \cup \text{CD}(v, \text{Descr}[\text{next}_\text{state}_j]) \cup \text{Var}(\text{guard}_j)$$

- For a functional output variable $v \in \text{out}$,

$$\text{infl}(v) = \bigcup_{j=1,\ldots,n} \text{DD}(v, \text{Descr}[\text{next}_\text{out}_j]) \cup \text{CD}(v, \text{Descr}[\text{next}_\text{out}_j]) \cup \text{Var}(\text{guard}_j)$$

- For an adaptive state variable $v \in \text{adapt}_\text{loc}$,

$$\text{infl}(v) = \text{DD}(v, \text{Descr}[\text{adapt}_\text{next}_\text{state}]) \cup \text{CD}(v, \text{Descr}[\text{adapt}_\text{next}_\text{state}])$$

- For an adaptive output variable $v \in \text{adapt}_\text{out}$,

$$\text{infl}(v) = \text{DD}(v, \text{Descr}[\text{adapt}_\text{next}_\text{out}]) \cup \text{CD}(v, \text{Descr}[\text{adapt}_\text{next}_\text{out}])$$
• For the configuration variable, $\text{infl}(\text{useconf}) = \bigcup_{j=1,...,n} \text{Var}(\text{guard}_j)$.

The set of variables influencing a property $\varphi$ in a module $m$ is computed iteratively. As initial set, the variables of the module $m$ occurring in the property are selected. In each iteration step, the dependencies of the variables already encountered are resolved and the corresponding variables are added to the set of influences until a fixpoint is reached.

**Definition 5.22 (Module Influences).** Let $\text{Descr}[m]$ be a syntactical SAS representation for the module $m$ and $\varphi$ an $L_{SAS}$ property. The set of module influences $\text{infl}(m, \text{Var}(\varphi))$ is the smallest fixpoint such that

- $\text{infl}(m, \text{Var}(\varphi)) \subseteq \text{Var}(\varphi) \cap \text{Var}(m)$
- For $v \in \text{infl}(m, \text{Var}(\varphi))$, $\text{infl}(v) \subseteq \text{infl}(m, \text{Var}(\varphi))$.

A module can be reduced with respect to the influencing variables by removing all variables not included in $\text{infl}$ from $\text{Var}(m)$ as they are not relevant for the validity of the considered property. Furthermore, all assignments to these variables are removed from the module’s adaptive transition functions $\text{adapt}_{\text{next state}}$ and $\text{adapt}_{\text{next out}}$ and from the functional transition functions of each configuration $\text{next state}_j$ and $\text{next out}_j$. For adaptive variables occurring in configuration guards, the only possibility that such an adaptive variable is not included in the set of module influences is that only adaptive variables excluding the $\text{useconf}$ variable are contained in the considered property. If a functional variable or the $\text{useconf}$ variable occurs in a property, all variables occurring in all configuration guards will be added to the module influences by adaptive dependence. This means, however, that if a variable in a configuration guard is not included in the set of module influences, the functionality implemented in the module configurations does not influence the validity of the property. Hence, we can replace the configuration guard over adaptive variables not included in $\text{infl}$ with true. The following definition captures the reduction of a syntactic SAS description with respect to the module influences formally.

**Definition 5.23 (Module Slicing of Modules).** Let $\text{Descr}[m]$ be a syntactic SAS description $\text{Descr}[m] = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{Descr[confs]}, \text{Descr[adapt]})$ for a module $m$ and $\varphi$ an $L_{SAS}$ property. We define the module description reduced with respect to the module influences $\text{infl}(m, \text{Var}(\varphi))$ by $\text{Descr}[m]|_{\text{infl}(m, \text{Var}(\varphi))}$ where

- $\text{in}|_{\text{infl}(m, \text{Var}(\varphi))} = \text{in} \cap \text{infl}(m, \text{Var}(\varphi))$,
- $\text{out}|_{\text{infl}(m, \text{Var}(\varphi))} = \text{out} \cap \text{infl}(m, \text{Var}(\varphi))$,
- $\text{loc}|_{\text{infl}(m, \text{Var}(\varphi))} = \text{loc} \cap \text{infl}(m, \text{Var}(\varphi))$ and $\text{init}|_{\text{infl}(m, \text{Var}(\varphi))} = \text{init} |_{\text{infl}(m, \text{Var}(\varphi))}$
• for each configuration \( j \) we have:
  - \( \text{guard}_j \mid \text{infl}(m, \text{Var}(\varphi)) = \text{reduce}(\text{guard}_j, \text{infl}(m, \text{Var}(\varphi))) \)
  - \( \text{Descr}[\text{next\_state}_j] \mid \text{infl}(m, \text{Var}(\varphi)) = \text{reduce}(\text{Descr}[\text{next\_state}_j], \text{infl}(m, \text{Var}(\varphi))) \)
  - \( \text{Descr}[\text{next\_out}_j] \mid \text{infl}(m, \text{Var}(\varphi)) = \text{reduce}(\text{Descr}[\text{next\_out}_j], \text{infl}(m, \text{Var}(\varphi))) \)

• \( \text{adapt\_in} \mid \text{infl}(m, \text{Var}(\varphi)) = \text{adapt\_in} \cap \text{infl}(m, \text{Var}(\varphi)) \),
  \( \text{adapt\_out} \mid \text{infl}(m, \text{Var}(\varphi)) = \text{adapt\_out} \cap \text{infl}(m, \text{Var}(\varphi)) \) and
  \( \text{adapt\_init} \mid \text{infl}(m, \varphi) = \text{adapt\_init} \mid \text{infl}(m, \text{Var}(\varphi)) \)

• \( \text{Descr}[\text{adapt\_next\_state}] \mid \text{infl}(m, \varphi) = \text{reduce}(\text{Descr}[\text{adapt\_next\_state}], \text{infl}(m, \text{Var}(\varphi))) \)

• \( \text{Descr}[\text{adapt\_next\_out}] \mid \text{infl}(m, \varphi) = \text{reduce}(\text{Descr}[\text{adapt\_next\_out}], \text{infl}(m, \text{Var}(\varphi))) \)

where \( \text{reduce}(f, V) \) for a function description \( f \in \text{Func} \) is defined by

• \( \text{reduce}([a_1, \ldots, a_n], V) = \text{reduce}([a_2, \ldots, a_n], V) \) if \( a_1 \) is an assignment to \( v \) and \( v \notin V \)

• \( \text{reduce}([a_1, \ldots, a_n], V) = [a_1, \text{reduce}([a_2, \ldots, a_n], V)] \) if \( a_1 \) is an assignment to \( v \) and \( v \in V \)

• \( \text{reduce}([], V) = [] \) where \( [] \) denotes the empty list

and for \( c \in \text{Constraint} \), \( \text{reduce}(c, V) = c \) if \( \text{Var}(c) \subseteq V \) else \( \text{reduce}(c, V) = \text{true} \)

Now, we can show that for a property \( \varphi \) only containing variables of a module \( m \), the transition system induced by the original module \( m \) and the transition system induced by the module reduced with respect to the module influences are consistently bisimilar with respect to \( \varphi \). The proof proceeds by choosing a bisimulation relation such that two states of the original and the reduced transition system are in relation if they coincide on the values of the variables included in the module influences.

**Theorem 5.24 (Consistent Bisimulation for Module Slicing on Module).** Let \( m \) be an SAS module \( m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt}) \) given by the syntactic description \( \text{Descr}[m] \) and \( \varphi \) an \( L_{\text{SAS}} \) property such that \( \text{Var}(\varphi) \subseteq \text{Var}(m) \). Then it holds that the induced transition systems for the module \( m \) and for the module reduced with respect to the module influences \( \text{infl}(m, \text{Var}(\varphi)) \) are consistently bisimilar with respect to \( \varphi \), i.e. it holds that

\[
T_m \cong_{\varphi} T_{m|\text{infl}(m, \text{Var}(\varphi))}
\]
proof. Let \( m' := m|_{\text{infl}(\mu, \text{Var}(\phi))} \). As bisimulation relation, we choose \((s, s') \in R \subseteq S \times S'\) iff for all \( v \in \text{infl}(m, \text{Var}(\phi))\) it holds that \( s(v) = s'(v) \) and the concretisation is the identity function.

1. Initial simulation: Let \( s_0 \) be an initial state of \( T_m \) and \( s'_0 \) an initial state of \( T_{m'} \). By construction of \( T_{m'} \) it holds that for all \( s_0 \) and \( s'_0 \), \( R(s_0, s'_0) \).

2. Step simulation (1): Let \( s_i \rightarrow_m s_{i+1} \) using configuration \( j \) and \( R(s_i, s'_i) \). Assume that there exists \( s'_{i+1} \) such that \( s'_i \rightarrow_{m'} s'_{i+1} \), but \((s_{i+1}, s'_{i+1}) \notin R\).

By construction of \( m' \), the only possibility for differing resulting states is that there is a configuration \( k \) with \( j \neq k \) such that \( s'_i \rightarrow_{m'} s'_{i+1} \) using configuration \( k \). Without loss of generality, assume that \( j > k \), then \( \text{eval}(\text{guard}_j)[s] = \text{true} \), whereas \( \text{eval}(\text{guard}_j)[s'] = \text{false} \). This can only hold if \( \text{Var}(\text{guard}_j) \notin \text{infl}(m, \text{Var}(\phi)) \) which is a contradiction to the construction of \( \text{infl}(m, \text{Var}(\phi)) \). This yields that \((s_{i+1}, s'_{i+1}) \in R\).

3. Step simulation (2): Let \( s'_i \rightarrow_{m'} s'_{i+1} \) and \( R(s_i, s'_i) \). By construction of \( m' \), it holds that there exists \( s_{i+1} \) such that \( s_i \rightarrow_m s_{i+1} \) and \((s_{i+1}, s'_{i+1}) \in R\).

4. Consistency: As \( \text{Var}(\phi) \subseteq \text{Var}(\text{infl}(m, \text{Var}(\phi))) \), we have for all \( a \in \text{Atoms}(\phi) \) if \( R(s, s') \) that \( s \models a \) iff \( s' \models a \).

By the consistent bisimulation between a module and the module reduced with respect to the module influences for a property \( \phi \) only containing variables of \( m \), we can establish the result that if the property \( \phi \) holds over the reduced module it is also true over the original module. For a property only affecting \( m \), it suffices to show the property over the reduced module in order to infer the validity over the original module. The proof uses the fact that properties for modules are preserved by consistent bisimulation in Corollary 5.11.

Corollary 5.25 (Property-Preservation for Module Slicing on Module). Let \( m \) be an SAS module \( m = (in, out, loc, init, confs, adapt) \) given by the syntactic description \( \text{Descr}[m] \) and \( \phi \) an \( L_{\text{SAS}} \) property such that \( \text{Var}(\phi) \subseteq \text{Var}(m) \). It holds that

\[ T_{m|_{\text{infl}(m, \phi)}} \models \phi \iff T_m \models \phi \]

Proof. Immediately by the consistent bisimulation with respect to \( \phi \) in Theorem 5.24 and property preservation by consistent bisimulation on module level in Corollary 5.11.

The idea of module slicing can be lifted to the system level. This means that a set of variables, the system influences, is computed containing all
variables of the system influencing the validity of a property \( \varphi \). Then, the system can be reduced with respect to the system influences resulting in a system which comprises only the variables relevant for the validity of the considered property. The initial set of influences on system level \( \text{infl}(SAS, \varphi) \) is the set of variables occurring in the property. For modules \( m_i \) in the system directly referenced by the property, \( \text{infl}(m_i, \text{Var}(\varphi)) \) is computed. For module input and system output variables contained in \( \text{infl}(SAS, \varphi) \), connected output variables of other modules are added. For each module \( j \) with variables in \( \text{infl}(SAS, \varphi) \), the module influences \( \text{infl}(m_j, \text{Var}(\varphi)) \) are computed again and so on, until fixpoint is reached.

**Definition 5.26 (System Influences).** Let \( SAS \) be an SAS system \( SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) containing the modules \( M = \{m_1, \ldots, m_n\} \) given by the syntactic descriptions \( \text{Descr}[m_i] \) for \( i = 1, \ldots, n \) and \( \varphi \) an \( L_{SAS} \) property over \( SAS \). The set of system influences \( \text{infl}(SAS, \varphi) \) is the smallest fixpoint such that

- \( \text{Var}(\varphi) \subseteq \text{infl}(SAS, \varphi) \).
- For \( V \subseteq \text{infl}(SAS, \varphi) \cap \text{Var}(m_i) \), \( \text{infl}(m_i, V) \subseteq \text{infl}(SAS, \varphi) \).
- If \( v \in \text{infl}(SAS, \varphi) \) and \( w \in \text{Var}_{m_j} \) and \( (w, v) \in \text{conn}_a \cup \text{conn}_d(w, v) \), \( w \in \text{infl}(SAS, \varphi) \).

The system reduced with respect to the system influences \( SAS|_{\text{infl}(SAS, \varphi)} \) is computed by reducing all modules as previously described with respect to the module variables included in the set of system influences. If no variable of a module is included in the system influences, the module is removed completely. Thus, module slicing subsumes system slicing. On system level, all system variables not contained in the system influences are removed from \( \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d \) as well as all system connections to deleted module variables. For module variables \( v \) that are now unconnected, a new system variable \( v_{\text{sys}} \) is added together with a corresponding connection such that the result of the reduction is a valid system.

**Definition 5.27 (Module Slicing).** Let \( SAS \) be an SAS system \( SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) containing the modules \( M = \{m_1, \ldots, m_n\} \) given by the syntactic descriptions \( \text{Descr}[m_i] \) for \( i = 1, \ldots, n \) and \( \varphi \) an \( L_{SAS} \) property over \( SAS \). The system reduced with respect to the system influences \( \text{infl}(SAS, \varphi) \) is defined by \( SAS|_{\text{infl}(SAS, \varphi)} = (M|_{\text{infl}(SAS, \varphi)}, \text{input}_a|_{\text{infl}(SAS, \varphi)}, \text{input}_d|_{\text{infl}(SAS, \varphi)}, \text{output}_a|_{\text{infl}(SAS, \varphi)}, \text{output}_d|_{\text{infl}(SAS, \varphi)}, \text{conn}_a|_{\text{infl}(SAS, \varphi)}, \text{conn}_d|_{\text{infl}(SAS, \varphi)}) \) where
Chapter 5. Model Transformations

- $M|_{\text{infl}(SAS, \varphi)} = \{ m_i |_{\text{infl}(SAS, \varphi)} \cap \text{Var}(m_i) \mid \text{infl}(SAS, \varphi) \cap \text{Var}(m_i) \neq \emptyset \}$
- $\text{input}_a|_{\text{infl}(SAS, \varphi)} = \text{input}_a \cap \text{infl}(SAS, \varphi)$
- $\text{input}_d|_{\text{infl}(SAS, \varphi)} = \text{input}_d \cap \text{infl}(SAS, \varphi)$
- $\text{output}_a|_{\text{infl}(SAS, \varphi)} = (\text{output}_a \cap \text{infl}(SAS, \varphi)) \cup \{ v_{\text{sys}} \mid v \in \overline{\text{Conn}_a(SAS)} \}$
- $\text{output}_d|_{\text{infl}(SAS, \varphi)} = (\text{output}_d \cap \text{infl}(SAS, \varphi)) \cup \{ v_{\text{sys}} \mid v \in \overline{\text{Conn}_d(SAS)} \}$
- $\text{conn}_a|_{\text{infl}(SAS, \varphi)} = (\text{conn}_a \cap (\text{infl}(SAS, \varphi) \times \text{infl}(SAS, \varphi)))$
  $\cup \{ (v, v_{\text{sys}}) \mid v \in \overline{\text{Conn}_a(SAS)} \}$
- $\text{conn}_d|_{\text{infl}(SAS, \varphi)} = (\text{conn}_d \cap (\text{infl}(SAS, \varphi) \times \text{infl}(SAS, \varphi)))$
  $\cup \{ (v, v_{\text{sys}}) \mid v \in \overline{\text{Conn}_d(SAS)} \}$

where $\overline{\text{Conn}_a(SAS)} = \{ v \in \text{Var}_{\text{adapt, out}}(M|_{\text{infl}(SAS, \varphi)}) \mid$
  $\neg \exists w \in \text{infl}(SAS, \varphi) \text{ with conn}_a(v, w) \}$

and $\overline{\text{Conn}_d(SAS)} = \{ v \in \text{Var}_{\text{out}}(M|_{\text{infl}(SAS, \varphi)}) \mid$
  $\neg \exists w \in \text{infl}(SAS, \varphi) \text{ with conn}_d(v, w) \}$

denote the sets of unconnected adaptive and functional module output variables, respectively.

The transition system induced by the original system $SAS$ and the transition system induced by the system reduced with respect to the system influences $SAS|_{\text{infl}(SAS, \varphi)}$ form a consistent bisimulation with respect to the property $\varphi$. For the proof, we take two states in bisimulation relation if they coincide on all variables in $\text{infl}(SAS, \varphi)$. This relation preserves initial states, transitions and atomic propositions as variables not included in the system influences cannot alter the values of the included variables by construction of the system influences.

**Theorem 5.28 (Consistent Bisimulation for Module Slicing).** Let $SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d)$ be a system and $\varphi$ an $L_{SAS}$ property over $SAS$ and $SAS|_{\text{infl}(SAS, \varphi)}$ as defined in Definition 5.27. Then, the transition systems induced by $SAS$ and by its reduction with respect to the system influences $SAS|_{\text{infl}(SAS, \varphi)}$ are consistently bisimilar with respect to $\varphi$:

$$T_{SAS} \cong_{[\varphi]} T_{SAS|_{\text{infl}(SAS, \varphi)}}$$

**Proof.** Let $SAS' := SAS|_{\text{infl}(SAS, \text{Var}(\varphi))}$. As bisimulation relation, we choose $(\sigma, \sigma') \in R \subseteq \Sigma \times \Sigma'$ iff for all $v \in \text{infl}(SAS, \text{Var}(\varphi))$ it holds that $\sigma(v) = \sigma'(v)$. The concretisation function is the identity function.

1. Initial simulation: Let $\sigma_0$ be an initial state of $SAS$ and $\sigma'_0$ an initial state of $SAS'$. It is clear by construction of $SAS'$ that for all $\sigma_0$ and $\sigma'_0$ it holds that $R(\sigma_0, \sigma'_0)$. 

5.2. Model Reduction by Slicing

2. Step simulation (1): Let \( \sigma_i \leadsto_{SAS} \sigma_{i+1} \) and \( R(\sigma_i, \sigma'_i) \). Assume that there exists \( \sigma'_{i+1} \) such that \( \sigma'_{i+1} \leadsto_{SAS'} \sigma'_{i+1} \). By construction of \( SAS' \) and preservation of the transitions in all modules by Theorem 5.24, we must have \( (\sigma_{i+1}, \sigma'_{i+1}) \in R \).

3. Step simulation (2): Let \( \sigma'_i \leadsto_{SAS'} \sigma'_{i+1} \) and \( R(\sigma_i, \sigma'_i) \). By construction of \( SAS' \), there exists \( \sigma_{i+1} \) such that \( \sigma_i \leadsto_{SAS} \sigma_{i+1} \) and \( (\sigma_{i+1}, \sigma'_{i+1}) \in R \).

4. Consistency: As \( \text{Var}(\phi) \subseteq \text{Var}(\text{infl}(SAS, \text{Var}(\phi))) \), we have for all \( a \in \text{Atoms}(\phi) \) if \( R(\sigma, \sigma') \) that \( \sigma \models a \iff \sigma' \models a \). \( \square \)

The consistent bisimulation between original system \( SAS \) and the system reduced with respect to the system influences \( SAS|_{\text{infl}(SAS,\phi)} \) allows us to conclude that module slicing preserves the validity of all properties from \( L_{SAS} \) by Theorem 5.10. Thus, it suffices to verify a property \( \phi \) over the reduced system in order to establish its validity for the original system.

Corollary 5.29 (Property Preservation for Module Slicing). Let \( SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) be a system and \( \phi \) an \( L_{SAS} \) property over \( SAS \) and \( SAS|_{\text{infl}(SAS,\phi)} \) as in Definition 5.27. Then it holds that:

\[ T_{SAS|_{\text{infl}(SAS,\phi)}} \models \phi \iff T_{SAS} \models \phi \]

Proof. Immediately from Theorem 5.28 and Theorem 5.10. \( \square \)

5.2.3 Adaptive Slicing

In synchronous adaptive system, functional and adaptive system parts are completely syntactically separated. The adaptation behaviour is specified without any reference to functionality since the adaptation aspect encapsulates it entirely. The adaptive next_state and next_out functions only depend on adaptive variables. Configuration guards only contain adaptive variables. Hence, the selection of a configuration never depends on functional values. Instead, the configurations provide interfaces to the respective functionality. A main advantage of this separation is that for verification of purely adaptive properties, functional descriptions in a model can be removed without any further analysis. A property is purely adaptive according to Definition 3.18 if it only contains adaptive variables and useconf variables for expressing the configurations used in modules.

To this end, adaptive slicing is a special slicing technique for adaptive systems. Based on the definition of SAS, adaptive slicing reduces the system to the adaptation behaviour simply by projecting out functional system
parts. On module level, functional variables and their initialisation are removed together with the `next_state` and `next_out` functions of the single configurations. The configuration guards are kept as they influence the used configuration of a module. Adaptive slicing of a system reduces all modules to their adaptation behaviour, removes all functional system variables in `input_d ∪ output_d` and deletes all functional connections in `conn_d`.

**Definition 5.30 (Adaptive Slicing).** Let `SAS` be a system `SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)`. We define the adaptively sliced system as

\[
SAS|_{\text{adapt}} = (M|_{\text{adapt}}, input_a, output_a, conn_a, \varnothing)
\]

where \( M|_{\text{adapt}} = \{m|_{\text{adapt}} \mid m \in M\} \).

For a module `m = (in, out, loc, init, confs, adapt)`, we define the adaptively sliced module by

\[
m|_{\text{adapt}} = (\varnothing, \varnothing, \varnothing, \varnothing, confs|_{\text{adapt}}, \varnothing)
\]

where \( confs|_{\text{adapt}} = \{conf_j|_{\text{adapt}} \mid conf_j \in confs\} \) and \( conf_j|_{\text{adapt}} = (guard_j, \varnothing, \varnothing) \).

For a purely adaptive property \( \varphi \), we can show that `SAS` and `SAS|_{\text{adapt}}` form a consistent bisimulation with respect to \( \varphi \). The proof proceeds by giving a bisimulation relation requiring that two states in relation agree on the adaptive variables. The concretisation is the identity as the interpretation of variables and values is not changed by adaptive slicing. By design of SAS models, the bisimulation relation preserves initial states, transitions and atomic propositions over adaptive variables.

**Theorem 5.31 (Consistent Bisimulation for Adaptive Slicing).** Let `SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)` be a system and `SAS|_{\text{adapt}}` the adaptively sliced system. Let \( \varphi \) denote a purely adaptive property over `SAS` such that `Var(\varphi) ⊆ Var_a(SAS)`.

Then it holds that the induced transition systems are consistently bisimilar with respect to \( \varphi \),

\[
T_{SAS} \cong_{[\varphi]} T_{SAS|_{\text{adapt}}}
\]

**Proof.** We define a bisimulation relation on \( \Sigma \) and \( \Sigma|_{\text{adapt}} \) as \( R(\sigma, \sigma|_{\text{adapt}}) \) iff for all \( v \in Var|_{\text{adapt}}(SAS) \) it holds that \( \sigma(v) = \sigma|_{\text{adapt}}(v) \). The concretisation \( C \) is the identity function.
• Initial Bisimulation: For \( \sigma_0 \in Init \), we have \( (\sigma_0, \sigma_0|_{\text{adapt}}) \in \mathcal{R} \) where \( \sigma_0|_{\text{adapt}} \in Init|_{\text{adapt}} \). Similarly, for \( \sigma_0|_{\text{adapt}} \in Init \), we have \( (\sigma_0, \sigma_0|_{\text{adapt}}) \in \mathcal{R} \) where \( \sigma_0 \in Init \).

• Step Simulation (1): Let \( \sigma_i \xrightarrow{\text{SAS}} \sigma_{i+1} \) and \( \mathcal{R}(\sigma_i, \sigma_i|_{\text{adapt}}) \). By definition of the separation of functionality and adaptation behaviour in SAS, there exists \( \sigma_{i+1}|_{\text{adapt}} \) such that \( \sigma_i|_{\text{adapt}} \xrightarrow{\text{SAS}} \sigma_{i+1}|_{\text{adapt}} \) and \( \mathcal{R}(\sigma_{i+1}, \sigma_{i+1}|_{\text{adapt}}) \).

• Step Simulation (2): Let \( \sigma_i|_{\text{adapt}} \xrightarrow{\text{SAS}} \sigma_{i+1}|_{\text{adapt}} \) and \( \mathcal{R}(\sigma_i, \sigma_i|_{\text{adapt}}) \). By definition of the separation of functionality and adaptation behaviour in SAS, there exists \( \sigma_{i+1} \) such that \( \sigma_i \xrightarrow{\text{SAS}} \sigma_{i+1} \) and \( \mathcal{R}(\sigma_{i+1}, \sigma_{i+1}|_{\text{adapt}}) \).

• Consistency: Let \( \mathcal{R}(\sigma, \sigma|_{\text{adapt}}) \) and for \( a \in \text{Atoms}(\varphi) \), let \( \sigma \models a \). As \( \text{Var}(a) \subseteq \text{Var}_{\text{adapt}}(\text{SAS}) \), for all \( v \in \text{Var}_{\text{adapt}}(\text{SAS}) \), it holds that \( \sigma(v) = \sigma|_{\text{adapt}}(v) \) such that \( \sigma|_{\text{adapt}} \models a = C(a) \) and vice versa. \( \square \)

As a corollary, we can formulate that adaptive slicing preserves purely adaptive properties. This result allows verifying purely adaptive properties on the adaptively sliced system to conclude that the property also holds on the original system.

**Corollary 5.32 (Property Preservation for Adaptive Slicing).** Let \( \text{SAS} = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) be a system and \( \varphi \) an adaptive property, i.e. \( \text{Var}(\varphi) \subseteq \text{Var}_a(\text{SAS}) \). Then it holds that

\[ T_{\text{SAS}}|_{\text{adapt}} \models \varphi \iff T_{\text{SAS}} \models \varphi \]

**Proof.** Immediately from Theorem 5.31 and in Theorem 5.10 \( \square \)

Adaptive Slicing can also be performed on single modules, e.g. after a decomposition step (cf. Section 6.2.1). We can reformulate Theorem 5.31 on module level for an adaptive property only containing adaptive variables of this module.

**Theorem 5.33 (Consistent Bisimulation for Adaptive Module Slicing).** Let \( m \) be an SAS module \( m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{conf}s, \text{adapt}) \) and \( m|_{\text{adapt}} \) the adaptively sliced module. For a purely adaptive property \( \varphi \) over \( m \) with \( \text{Var}(\varphi) \subseteq \text{Var}_a(m) \), it holds that

\[ T_m \approx_{[\varphi]} T_m|_{\text{adapt}} \]

**Proof.** Analogue to the proof of Theorem 5.31 for the induced module transition system. \( \square \)
This result allows us to conclude, analogue to Corollary 5.32, that adaptive properties referring to one module only are preserved under adaptive slicing of modules in isolation. Hence, for an adaptive property over a module it suffices to show the property over the adaptively reduced module. This offers additional reduction potential in combination with compositional reasoning strategies presented in Chapter 6.

**Corollary 5.34 (Property Preservation for Adaptive Module Slicing).** Let \( m \) be an SAS module \( m = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt}) \) and \( \varphi \) an adaptive property, i.e. \( \text{Var}(\varphi) \subseteq \text{Var}_a(m) \). Then it holds that

\[
T_{m|\text{adapt}} \models \varphi \iff T_m \models \varphi
\]

**Proof.** Immediately from Theorem 5.33 and Corollary 5.11.

### 5.2.4 Combining Reductions by Slicing

In order to reduce verification complexity of adaptive system models, slicing is a valuable technique as it is fully automatic not requiring any user-interaction. Furthermore, slicing of high-level intermediate model representations allows exploiting the internal system structure for the analysis of irrelevant system parts. To this end, the proposed slicing techniques, system, module and adaptive slicing, use different levels of detail for their analyses. All techniques are modular in that they produce a valid system if they are given a valid system as input. This facilitates combining slicing techniques of different granularity to obtain a fine-grained control over the analysis effort for computing reductions. If, for example, system slicing was the only reduction technique available, it would not be possible to weigh these different interests against each other. The correctness of a combination of different slicing techniques can be justified by the correctness of each transformation step.

The ordering in which slicing techniques are applied determines the analysis complexity. A general guideline is to start with the slicing technique removing the largest part of a system with the least analysis effort. This avoids that a system part is analysed but later removed by another slicing step. Figure 5.7 shows the procedure how slicing techniques are best combined. First, adaptive slicing is performed, if applicable, since it removes all functional system parts without any further analysis of the model. Second, if the system contains more than one module and if the property considered only refers to a subset of these, system slicing can
remove all modules from the system that do not influence the modules directly affected by the property. Third, module slicing can be applied if the transition functions are given as SAS function descriptions. For a system where functionality is provided by pre/post-condition specifications, it is possible to use system and adaptive slicing, if applicable. If the property to be verified is an adaptive property, adaptive slicing removes the pre/post-condition specifications among the other functional system parts. Then, also module slicing on the reduced system is possible as now all transition functions, i.e. the adaptive ones, are syntactically specified.

As we have seen, module slicing subsumes system slicing. If module slicing is to be carried out, it is not necessary to perform system slicing beforehand. Nonetheless, in certain examples system slicing before module slicing can speed up the time for computing the module slice. In Section 7.2.2, we evaluate at the example of the adaptive vehicle stability control system how a combination of different slicing techniques influences the times required for computing different model reductions by slicing and how this relates to the reduction in verification time.

5.3 Data Domain Abstractions

Abstraction [Sha02] is widely used concept in order to reduce verification complexity for model checking. Since model checking can in general only be applied to finite-state systems, abstraction reduces infinite-state sys-
tems to finite-state systems while preserving the validity of the considered properties. Furthermore, abstraction is used to counter the state-explosion problem by minimising the model size from concrete to abstract systems. For SAS models, abstraction offers the possibility both to reduce infinite-state SAS systems to finite-state systems and also to reduce their sizes in general. With respect to their control structures, SAS models are finite-state systems as the number of modules, the connections and also the number of configurations inside a module is finite and fixed at design time. However, SAS models can use arbitrary data types, such as integers or real numbers, making them infinite-state.

In this direction, we propose a data domain abstraction approach [CGL94, DHJ+01] for SAS models. Data domain abstraction maps infinite concrete data domains to abstract finite domains in order to reduce a system model to finite-state or to reduce a large concrete data domain to a smaller abstract domain. Each value in the concrete domain is assigned a value of the abstract domain. Also operators in expressions and predicates in conditions are changed to be interpreted over the abstract values. The behaviour of an SAS model should nonetheless be captured as precisely as necessary for verification of desired properties. Hence, the concrete abstraction to be carried out on an SAS system always depends on the system itself and the property to be verified.

Abstractions can lead to the introduction of non-determinism into a system. Consider the abstraction of the integer domain to its sign, i.e. the domain \{pos, zero, neg\} where all integers greater than zero are mapped to pos, all integers smaller than zero are mapped to neg, and 0 is mapped to zero. The computation of an addition of two values in the abstract domain can have different results. Adding two positive integers is certainly positive, but addition of a negative and a positive integer can result in a positive value, a negative value or zero. Hence, instead of computing single abstract values, an abstract system operates on the powerset of abstract values. This means, that binary operators in expressions have to take sets of values as arguments.

By abstraction, uncertainty is also introduced into the evaluation of constraints in conditional assignments and into the evaluation of the configuration guards. Comparing whether an negative integer is smaller than a positive integer is true, but a positive integer may or may not be smaller than another positive integer depending on the value it corresponds to in the concrete. However, as this information is lost under abstraction, we introduce a third truth value unknown = \{true, false\} representing that the
5.3. Data Domain Abstractions

Evaluation of a predicate may be true or false such that also constraints can evaluate to true, false or unknown. As a consequence, conditions in SAS function descriptions can evaluate to the unknown value. This allows more than one possible variable assignment in a successor state as the condition can either be true such that the assignment comes into effect or false such that another conditional is selected. Therefore in a system abstraction, transition functions are replaced by relations in order to capture multiple successor states. Also configuration guards may evaluate to an unknown value. This leads to non-deterministic SAS systems formalised in Section 5.3.1.

In order to abstract a synchronous adaptive system, we have to provide the following definitions for a concrete abstraction procedure:

- an abstract domain of values \( \hat{Val} \)
- an abstraction mapping \( \alpha : Val \rightarrow \hat{Val} \) assigning each concrete value to an abstract counterpart
- an interpretation of the predicate symbols \( r \in Rel \) over the abstract domain \( \hat{r}_I : P(\hat{Val}) \times P(\hat{Val}) \rightarrow \{true, false, unknown\} \)
- an interpretation of the operation symbols \( b \in BinOp \) over the abstract domain \( \hat{b}_I : P(\hat{Val}) \times P(\hat{Val}) \rightarrow P(\hat{Val}) \)

It is important that the abstraction is safe meaning that the validity of an abstract property over an abstract system implies the validity of the concrete property over the concrete system. Then, it suffices to show the abstract property over the abstract system in order to establish that the original system satisfies the original property. For a safe abstraction the abstract operations must satisfy the following safety condition:

\[
\alpha(b(v_1, v_2)) \in \hat{b}(\alpha(v_1), \alpha(v_2))
\]

Equally, the predicates must satisfy:

\[
r(v_1, v_2) \in \hat{r}(\alpha(v_1), \alpha(v_2))
\]

Intuitively, the safety conditions denote that if the result of an operation is calculated in the concrete system and afterwards abstracted, it is contained in the result of first computing the abstraction and then applying the abstract operation. The same holds for predicates where the evaluation of the concrete predicate on concrete values must be included in the evaluation of the abstract predicate over the abstract values, e.g. if the abstract predicate evaluates to unknown = \{true, false\}. If the safety conditions on abstract operations and predicates hold, we can show that the
abstraction of a module as well as the abstraction of an SAS system are property-preserving.

This section is structured as follows: First, we formalise non-deterministic SAS semantics in order to capture abstract SAS systems formally. Second, we explain how to transform a syntactic SAS representation to an abstract SAS representation, given an abstract domain, an abstraction mapping and abstract operations and predicates, and prove that concrete and abstracted modules form a simulation. Third, we show how to abstract a $L_{SAS}$ property with respect to a system and justify that this abstraction is property-preserving. Finally, we present how data domain abstractions can be used in SAS verification by means of a library of pre-computed abstractions.

### 5.3.1 Non-Deterministic SAS Semantics

The main difference between deterministic and non-deterministic SAS models is that in the non-deterministic case a state can have multiple successors in a transition. This is first due to the fact that in an non-deterministic SAS module the configuration selected for computing the next functional state is no longer uniquely determined. By allowing that the configuration guards evaluate to an unknown value besides true and false, there can be several potentially active configurations. Second, the transition functions of the modules are replaced by transition relations. So, also on this level several successor states become possible.

A non-deterministic SAS module consists, such as a deterministic module, of a set of functional configurations with an adaptation aspect on top. Furthermore, it comprises functional input, output and local variables and their adaptive counterparts. The next state and next out functions of the configurations as well as adapt next state and adapt next out functions are replaced by relations.

**Definition 5.35 (Non-Deterministic Module).** A non-deterministic SAS module $m$ is defined as $m = (in, out, loc, init, confs, adapt)$ with

- $in \subseteq \text{Var}$, a set of input variables, $out \subseteq \text{Var}$, a disjoint set of output variables, $loc \subseteq \text{Var}$, a disjoint set of local variables and $init : in \cup loc \cup out \rightarrow Val$ their initial values
- $confs = \{(\text{guard}_j, \text{next}_\text{state}_j, \text{next}_\text{out}_j) \mid j = 1, \ldots, n\}$ the configurations of the module, where
  - $\text{guard}_j$: an SAS constraint over $\{\text{adapt}_\text{in}, \text{adapt}_\text{loc}\}$ with $\text{adapt}_\text{in}$ and $\text{adapt}_\text{loc}$ as defined below
5.3. Data Domain Abstractions

- \( \text{next}_\text{state}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \times (\text{loc} \rightarrow \text{Val}) \) the next state relation for configuration \( j \)
- \( \text{next}_\text{out}_j : (\text{in} \cup \text{loc} \rightarrow \text{Val}) \times (\text{out} \rightarrow \text{Val}) \) the output relation for configuration \( j \)

The adaptation aspect is defined as a tuple \( \text{adapt} = (\text{adapt}_\text{in}, \text{adapt}_\text{out}, \text{adapt}_\text{loc}, \text{adapt}_\text{init}, \text{adapt}_\text{next}_\text{state}, \text{adapt}_\text{next}_\text{out}) \) where

- \( \text{adapt}_\text{in} \subseteq \text{Var} \), a disjoint set of adaptation input variables,
- \( \text{adapt}_\text{out} \subseteq \text{Var} \), a disjoint set of adaptation output variables,
- \( \text{adapt}_\text{loc} \subseteq \text{Var} \), a disjoint set of adaptation local state variables and \( \text{adapt}_\text{init} : \text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \cup \text{adapt}_\text{out} \rightarrow \text{Val} \) their initial values
- \( \text{adapt}_\text{next}_\text{state} : (\text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \rightarrow \text{Val}) \times (\text{adapt}_\text{loc} \rightarrow \text{Val}) \) the adaptation next state relation
- \( \text{adapt}_\text{next}_\text{out} : (\text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \rightarrow \text{Val}) \times (\text{adapt}_\text{out} \rightarrow \text{Val}) \) the adaptation output relation

A local state of a non-deterministic SAS module is an assignment of the module’s variables to values. A local state is initial if the variables, both adaptive and functional, are assigned to their initial values, as in the deterministic case. The local transition relation is defined by the set of possible local transitions \( s \rightarrow_{\text{loc}} s' \). In a local transition, the functional and the adaptive input is set by the environment. For the adaptive local and output variables, the values of \( s' \) must be in adaptive next state and adaptive next output relation with the adaptive input and adaptive local variables of the previous state \( s \). The functional values must be related by the next state and next out relations of the active configuration.

However, determining the active configuration in a non-deterministic SAS module is more complicated than in the deterministic case. While in the deterministic case, the configuration guard either evaluates to true or false, in the non-deterministic case configuration guards can also evaluate to an unknown value meaning either true or false. A configuration guard evaluating to false means that the configuration cannot be activated. Then a configuration with lower priority becomes active. If a configuration guard evaluates to true, the configuration is definitely activated, and no other configuration with a lower priority becomes active. If a guard evaluates to unknown, the transition induced by this configuration constitutes one pair of states in the local transition relation. But also other configurations with lower priority and a guard evaluating to true or unknown are considered in the local transition relation. For two states to be included in the local
transition relation, there must exist a configuration \( j \) whose guard evaluates to unknown or true, while all configuration guards with higher priority evaluate to false or unknown, and the functional output and local variables of the states must be in the \( \text{next}_\text{state}_j \) and \( \text{next}_\text{out}_j \) relation. The special adaptive variable \( \text{useconf} \) storing the name of the used configuration in a transition is assigned accordingly to the configuration \( j \).

**Definition 5.36 (Non-Deterministic Local States and Transitions).** A \( \text{local state} \) \( s \) of a non-deterministic SAS module \( m \) is defined as variable assignment:

\[
s : \text{in} \cup \text{out} \cup \text{loc} \cup \text{adapt}_\text{in} \cup \text{adapt}_\text{out} \cup \text{adapt}_\text{loc} \rightarrow \text{Val}.
\]

A \( \text{local state} \) \( s \) is \( \text{initial} \) if \( s|_{\text{in} \cup \text{loc} \cup \text{out}} = \text{init} \) and \( s|_{\text{adapt}_\text{in} \cup \text{adapt}_\text{loc} \cup \text{adapt}_\text{out}} = \text{adapt}_\text{init} \).

A \( \text{local transition} \) between two states \( s \) and \( s' \) is defined as \( s \rightarrow_{\text{loc}} s' \) iff the following conditions hold:

\[
\begin{align*}
\text{adapt}_\text{next}_\text{state}(s'|_{\text{adapt}_\text{in}} \cup s|_{\text{adapt}_\text{loc}}, s'|_{\text{adapt}_\text{loc}}) \\
\text{adapt}_\text{next}_\text{out}(s'|_{\text{adapt}_\text{in}} \cup s|_{\text{adapt}_\text{loc}}, s'|_{\text{adapt}_\text{out}}) \forall 0 < j < i \cdot \text{eval}(\text{guard}_j)[s'|_{\text{adapt}_\text{in}} \cup s|_{\text{adapt}_\text{loc}}] \in \{\text{unknown, false}\} \text{ and } \text{eval}(\text{guard}_i)[s'|_{\text{adapt}_\text{in}} \cup s|_{\text{adapt}_\text{loc}}] \in \{\text{unknown, true}\} \\
\text{next}_\text{state}_i(s'|_{\text{in}} \cup s|_{\text{loc}}, s'|_{\text{loc}}) \text{ and } \text{next}_\text{out}_i(s'|_{\text{in}} \cup s|_{\text{loc}}, s'|_{\text{out}})
\end{align*}
\]

A path of \( m \) is defined as a sequence of states \( s_0s_1\ldots \) where \( s_0 \) is \( \text{initial} \), and for all \( i \geq 0 \), we have \( s_i \rightarrow_{\text{loc}} s_{i+1} \).

A non-deterministic SAS system is defined by composing non-deterministic SAS modules as in Definition 3.3. It does not make a difference whether non-deterministic or deterministic SAS modules are composed as long as the result is a valid system. The semantics of a non-deterministic SAS system is analogue to the semantics of a deterministic SAS system. A global state of a non-deterministic SAS system is a valuation of the modules’ variables and of the system variables. The global transition relation is defined in three steps: First, all inputs either from the system inputs or from other module outputs are read, then each local state of a module performs a local transition according to its non-deterministic local transition relation, and third, the computed outputs are written to the system outputs. As there are several successor states for each module local state, there are also multiple global successor states.

**Definition 5.37 (Non-Deterministic Synchronous Adaptive System).** A system \( \text{SAS} = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) is a non-deterministic SAS system if there is a module \( m \in M = \{m_1, \ldots, m_n\} \) such that \( m \) is a non-deterministic module as in Definition 5.35.
Global and initial states of SAS are defined as for deterministic SAS in Definition 3.7, where a global state $\sigma$ is defined as

$$\sigma = s_1 \cup \ldots \cup s_n \cup ((input_a \cup input_d \cup output_a \cup output_d) \rightarrow Val)$$

Two states $\sigma$ and $\sigma'$ perform a global transition, $\sigma \rightarrow_{glob} \sigma'$, iff the following conditions hold:

1. For all $x, y \in Var(SAS) \setminus (input_d \cup input_a)$ with $(x, y) \in conn_d$ or $(x, y) \in conn_a$, it holds that $\sigma'(y) = \sigma(x)$, for all $x \in input_a$ and $y \in Var(SAS)$ with $(x, y) \in conn_{\text{a}}$, it holds that $\sigma'(y) = \sigma'(x)$, and for all $x \in input_d$ and $y \in Var(SAS)$ with $(x, y) \in conn_{\text{d}}$, it holds that $\sigma'(y) = \sigma'(x)$.
2. For all $s_j \in \sigma$ and for all $s'_j \in \sigma'$, it holds that the local states are in the local transition relation $s_j \rightarrow_{\text{loc}} s'_j$.
3. For all $x \in Var(SAS)$ and $y \in output_d$ with $(x, y) \in conn_{\text{d}}$, it holds that $\sigma'(y) = \sigma'(x)$, and for all $x \in Var(SAS)$ and $y \in output_a$ with $(x, y) \in conn_{\text{a}}$, it holds that $\sigma'(y) = \sigma'(x)$.

The induced transition systems for non-deterministic SAS systems and non-deterministic SAS modules can be defined as in Definitions 3.8 and 3.10, respectively. The transition relations in these transition systems are given by the local and global transition relations defined in Definitions 5.36 and 5.37. The computation paths of non-deterministic SAS modules and non-deterministic SAS systems are defined exactly as in Definition 3.9. The only difference is that in the non-deterministic case the transition relation is a real relation, while in the deterministic case the transition relation reduces to a function. In this sense, deterministic SAS systems are a special case of non-deterministic SAS systems. Instead of a collecting semantics, non-deterministic SAS systems are used to formalise uncertainty in system computations because this allows applying the formal arguments for property preservation established in Section 5.1 also in case of non-determinism. These results hold for non-deterministic systems because the property that the transition relation of a deterministic SAS transition system is a function in not required in any proof.

5.3.2 Abstracting System Models

A syntactic SAS representation can be abstracted in a purely syntactical manner. Using the definitions of the abstract domain $\bar{Val}$, the abstraction mapping $\alpha$ and the abstract predicates and operators, syntactic SAS representations are abstracted by extending the abstraction mapping $\alpha$ from...
single values to syntactic constructs, such as expressions, constraints and functions. This reduces verification complexity enormously as the state space of the original transition system does not have to be constructed explicitly. Moreover, original and abstract SAS systems have the same syntactic structure which leads to increased traceability.

For the abstraction of an SAS expression, values are mapped to their abstract counterparts by \( \alpha \) whereas variables remain unchanged. A binary operation \( b \) over two expressions remains syntactically unchanged, but is later interpreted abstractly by the abstract operation \( \hat{b} \). Abstractions are carried out inductively on the structure of expressions. In SAS constraints, the constant \( \text{true} \) is mapped to the abstract constant \( \text{true} \), and so is \( \text{false} \). Predicates \( r \) over expressions are not changed syntactically, but later interpreted by the abstract interpretation \( \hat{r} \). This also includes that they can evaluate to the unknown value. The Boolean operators in a constraint are abstracted in order to work with the unknown value. Table 5.1 gives the value tables for the abstract Boolean conjunction \( \hat{\land} \) and negation \( \hat{\neg} \) satisfying the safety conditions for operators stated above. The symbol \( * \) denotes the unknown value. Constraints are further abstracted inductively on their structure. The same holds for SAS function descriptions. An unconditional assignment is abstracted by abstracting the assigned expression. In a conditional assignment, each conditional is abstracted by abstracting constraints and assigned expressions.

**Definition 5.38 (Abstracting Syntactic Constraints, Expressions and Functions).** For SAS expressions, we define the abstraction \( \hat{\alpha} : \text{Expr} \rightarrow \hat{\text{Expr}} \) where \( \hat{\text{Expr}} \) denotes the set of abstract expressions over the abstract domain by:

- \( \hat{\alpha}(x) = x \), and for \( v \in \text{Val} \), \( \hat{\alpha}(v) = \alpha(v) \)
- \( \hat{\alpha}(b(expr_1, expr_2)) = b(\hat{\alpha}(expr_1), \hat{\alpha}(expr_2)) \)

For SAS constraints, we define the abstraction \( \hat{\alpha} : \text{Constraint} \rightarrow \hat{\text{Constraint}} \) where \( \hat{\text{Constraint}} \) are constraints over the abstract domain by:

- \( \hat{\alpha}(\text{true}) = \text{true} \) and \( \hat{\alpha}(r(expr_1, expr_2)) = r(\hat{\alpha}(expr_1), \hat{\alpha}(expr_2)) \)

<table>
<thead>
<tr>
<th>( \land )</th>
<th>true</th>
<th>false</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>*</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>false</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \neg )</th>
<th>true</th>
<th>false</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.1.** Value Tables for Abstract Boolean Operators
5.3. Data Domain Abstractions

- $\hat{\alpha}(-c_1) = \hat{\alpha}(c_1)$
- $\hat{\alpha}(c_1 \land c_2) = \hat{\alpha}(c_1) \land \hat{\alpha}(c_2)$

For SAS functions, we define the abstraction $\hat{\alpha} : Func \to \hat{\text{Func}}$ where $\hat{\text{Func}}$ denotes function descriptions over the abstract domain by:

- for an unconditional assignment $\hat{\alpha}(x := \text{expr}) = x := \hat{\alpha}(\text{expr})$
- for a conditional assignment $\hat{\alpha}(x := [\text{cond}_1, \ldots, \text{cond}_n]) = x := \hat{\alpha}([\text{cond}_1, \ldots, \text{cond}_n])$ where $\hat{\alpha}([[c, \text{expr}], \text{cond}_2, \ldots, \text{cond}_n]) = ([\hat{\alpha}(c), \hat{\alpha}(\text{expr})], \hat{\alpha}([\text{cond}_2, \ldots, \text{cond}_n]))$ and $\hat{\alpha}([]) = []$ where $[]$ denotes the empty list
- for a function list $\hat{\alpha}([a_1, a_2, \ldots, a_n]) = [\hat{\alpha}(a_1), \hat{\alpha}([a_2, \ldots, a_n])$ and $\hat{\alpha}([]) = []$

A syntactic SAS module representation is abstracted by abstracting the configuration guards as well as all functions descriptions according to $\hat{\alpha}$. Initial values of adaptive and functional variables are mapped to abstract values by the abstraction mapping $\alpha$.

**Definition 5.39 (Abstract SAS Representation).** For a syntactic SAS module description $\text{Descr}[m] = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{Descr}[\text{confs}], \text{Descr}[\text{adapt}])$ for a module $m$, we define the abstract representation by

$\hat{\alpha}(\text{Descr}[m]) = (\text{in}, \text{out}, \text{loc}, \hat{\alpha}(\text{init}), \hat{\alpha}(\text{Descr}[\text{confs}]), \hat{\alpha}(\text{Descr}[\text{adapt}]))$

where

- $\hat{\alpha}(\text{Descr}[\text{adapt}]) = (\text{adapt}_\text{in}, \text{adapt}_\text{out}, \text{adapt}_\text{loc}, \hat{\alpha}(\text{adapt}_\text{init}), \hat{\alpha}(\text{Descr}[\text{adapt}_\text{next}\text{state}]), \hat{\alpha}(\text{Descr}[\text{adapt}_\text{next}\text{out}]))$
- $\hat{\alpha}(\text{Descr}[\text{confs}]) = \{(\hat{\alpha}(\text{guard}_j), \hat{\alpha}(\text{Descr}[\text{next}\text{state}_j]), \hat{\alpha}(\text{Descr}[\text{next}\text{out}_j]) | j = 1, \ldots, n\}$

The variable mappings $\text{init}$ and $\text{adapt}_\text{init}$ are abstracted such that for all $x \in \text{Var}_d(m)$, we have $\hat{\alpha}(\text{init})(x) = \alpha(\text{init}(x))$, and for all $x \in \text{Var}_a(m)$, we have $\hat{\alpha}(\text{adapt}_\text{init})(x) = \alpha(\text{adapt}_\text{init}(x))$.

The abstract representations for constraints, expressions and functions are interpreted over SAS states $\hat{\sigma} : \text{Var} \to \mathcal{P}(\hat{\text{Val}})$ where variables are assigned to the powerset of abstract values. Expressions and constraints are interpreted as denoted in Definition 3.5 using the interpretations over the abstract powerset domain $\hat{\mathcal{B}}_r$ and $\hat{\mathcal{R}}_r$ for operators and predicates, respectively. A constraint $c \in \text{Constraint}$ is interpreted over the truth values $\text{true}$, $\text{false}$ and $\text{unknown}$. This also applies to the configuration guards leading to a non-deterministic abstract system.
An abstract function representation is interpreted by the abstract evaluation $\text{eval} : \text{Func} \times \hat{\Sigma} \rightarrow \mathcal{P}(\hat{\Sigma})$ where the evaluation of an abstract function description in a state results in a set of states, all possible successor states. For a simple assignment, the expression is evaluated abstractly, and the resulting set of abstract values is assigned to the variable. In a conditional assignment, a case distinction is made whether the constraint in the first conditional evaluates to true, false or unknown. If it is true, the expression is abstractly evaluated, and the result is assigned to the respective variable. If it evaluates to false, the following conditionals in the conditional assignment are considered. If a constraint evaluates to the unknown value, a case split is performed such that the result of the evaluation is a set of states. In one state, the constraint is assumed as true, and the respective variable is set accordingly, in the other, the constraint is taken as false, and the subsequent conditionals are considered. If no more conditionals exist, the resulting state is set to the input state. In a function list, the first function is evaluated resulting in a set of states. In each of the resulting states, the remainder of the function list is evaluated. If the function list is empty, the result is a singleton containing the unchanged input state.

**Definition 5.40 (Evaluation of Abstract Function Descriptions).** The semantics of an abstract function description $f \in \hat{\text{Func}}$ in a state $\sigma$ is defined by $\text{eval}(f)[\sigma] : \text{Func} \times \hat{\Sigma} \rightarrow \mathcal{P}(\hat{\Sigma})$ where

- **for an unconditional assignment**, $\text{eval}(x := \text{expr})[\sigma] = \{\sigma'\}$ with $
\sigma'(x) = \text{eval}(\text{expr})[\sigma], \text{ and for all } y \neq x, \sigma'(y) = \sigma(y).$

- **for a conditional assignment**, $\text{eval}(x := [(c, \text{expr}), \text{conds}])[\sigma]$ is defined as
  - if $\text{eval}(c)[\sigma] = \text{true}$, then
    $$\text{eval}(x := [(c, \text{expr}), \text{cond}_2, \ldots, \text{cond}_n])[\sigma] = \{\sigma'\}$$
    with $\sigma'(x) = \text{eval}(x := \text{expr})[\sigma], \text{ and for all } y \neq x, \sigma'(y) = \sigma(y)$
  - if $\text{eval}(c)[\sigma] = \text{false}$, then
    $$\text{eval}(x := [(c, \text{expr}), \text{cond}_2, \ldots, \text{cond}_n])[\sigma] = \{\sigma'\}$$
    with $\sigma'(x) = \text{eval}(x := [\text{cond}_2, \ldots, \text{cond}_n])[\sigma], \text{ and for eval}(x := [])[\sigma], \text{ it holds that } \sigma(x) = \sigma'(x), \text{ and for all } y \neq x, \sigma'(y) = \sigma(y).$
  - if $\text{eval}(c)[\sigma] = \text{unknown}$, then
    $$\text{eval}(x := [(c, \text{expr}), \text{cond}_2, \ldots, \text{cond}_n])[\sigma] = \{\sigma_1', \sigma_2'\}$$
    with $\sigma_1' = \text{eval}(x := \text{expr})[\sigma] \text{ and } \sigma_2' = \text{eval}(x := [\text{cond}_2, \ldots, \text{cond}_n])[\sigma]$

- **for a function list** $\text{eval}([a_1, \ldots, a_n])[\sigma] = \Sigma'$ where
  - $\text{eval}([a_1, \ldots, a_n])[\sigma] = \Sigma'$ where $\Sigma' = \bigcup_{\sigma \in \Sigma_1} \text{eval}([\text{cond}_2, \ldots, \text{cond}_n])[\sigma]$
  - $\text{with } \Sigma_1 = \text{eval}(a_1)[\sigma]$
  - $\text{eval}([])[\sigma] = \{\sigma\}$
Having defined the evaluation of an abstract function description, we use abstract function descriptions for defining transition relations for non-deterministic SAS modules. As a state can have multiple successors, the abstract syntactic SAS representation describes a non-deterministic SAS module defined in Definition 5.35.

**Definition 5.41 (Abstract Transition Relations).** An abstract syntactic SAS module description for a module \( m \) \( \hat{\alpha}(\text{Descr}[m]) = \langle \text{in}, \text{out}, \text{loc}, \hat{\alpha}(\text{init}), \hat{\alpha}(\text{Descr[confs]}), \hat{\alpha}(\text{Descr[adapt]}) \rangle \) defines the following transition relations on states \( s, s' \):

- \( \text{next\_state}_j : (\text{in} \cup \text{loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \times (\text{loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \) the next state relation for configuration \( j \) where \( \text{next\_state}_j(s, s') \iff s' \in \mathcal{E}(\hat{\alpha}(\text{Descr[next\_state]}))[s] \)
- \( \text{next\_out}_j : (\text{in} \cup \text{loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \times (\text{out} \rightarrow \mathcal{P}(\hat{\text{Val}})) \) the output relation for configuration \( j \) where \( \text{next\_out}_j(s, s') \iff s' \in \mathcal{E}(\hat{\alpha}(\text{Descr[next\_out]}))[s] \)
- \( \text{adapt\_next\_state} : (\text{adapt\_in} \cup \text{adapt\_loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \times (\text{adapt\_loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \) the adaptation next state relation where \( \text{adapt\_next\_state}(s, s') \iff s' \in \mathcal{E}(\hat{\alpha}(\text{Descr[adapt\_next\_state]}))[s] \)
- \( \text{adapt\_next\_out} : (\text{adapt\_in} \cup \text{adapt\_loc} \rightarrow \mathcal{P}(\hat{\text{Val}})) \times (\text{adapt\_out} \rightarrow \mathcal{P}(\hat{\text{Val}})) \) the adaptation output function where \( \text{adapt\_next\_out}(s, s') \iff s' \in \mathcal{E}(\hat{\alpha}(\text{Descr[adapt\_next\_out]}))[s] \)

As an example for a data domain abstraction reducing an infinite-state system to finite-state, consider the brake module already introduced in Section 4.2. This module is infinite-state as the input value for brake force and gas brakeGasInput and the output value brake are integer variables. However, for checking that brake and gas outputs computed from the brake and gas input are mutually exclusive, it suffices to consider the signs of the inputs and outputs. As abstract domain, we chose the set \( \hat{\text{Val}} = \{\text{pos, neg, zero}\} \). For the abstraction, we map positive integers to \( \text{pos} \), negative to \( \text{neg} \) and 0 to zero. The transition function determining the brake output in the concrete operates as shown in Figure 3.4. If the input brakeGasInput is less or equal to zero, the output brake is assigned to the absolute value of the input by negating the input. If the input is greater than zero, the output is set to zero as a positive input is interpreted as gas value. For the abstraction, we have to find abstract operators that comply to the safety conditions. Negating a value in the abstract domain means that a positive integer becomes negative, i.e. \( -\text{pos} = \text{neg} \), a negative integer becomes positive, i.e. \( -\text{neg} = \text{pos} \), and 0 remains 0, i.e. \( -\text{zero} = \text{zero} \). Using
this definition, we obtain the following abstract transition function for the brake output. Note, that in this example the transition function remains deterministic as the abstraction is chosen to reflect the concrete behaviour as precisely as possible avoiding the powerset of abstract values.

\[
\begin{align*}
\text{brake} &:= \begin{cases} 
(\text{brakeGas\_Input} = \text{pos}, \text{zero}) , \\
(\text{brakeGas\_Input} = \text{zero}, \text{zero}), \\
(\text{brakeGas\_Input} = \text{neg}, \text{pos}) 
\end{cases}
\end{align*}
\]

As a first step to prove that the introduced data domain abstraction is property-preserving, we have to show that a concrete SAS module and its abstraction simulate each other. So, we have to find a relation between states of an original module \(m\) and its abstraction \(\hat{m}\) that preserves initial states and transitions. We consider an original state \(s\) and an abstract state \(\hat{s}\) as similar if the abstraction of a value that a variable in the original state \(s\) assumes is contained in the set of values this variable assumes in the abstract state \(\hat{s}\). In this sense, the abstraction is an overapproximation of the concrete behaviour.

**Definition 5.42 (Simulation Relation for Data Domain Abstraction).**

Let \(T_m\) be an SAS module transition system \(T_m = (S, \text{Init}, \rightarrow)\) for a module \(m\) given by the syntactic representation \(\text{Descr}[m]\) and \(T_{\hat{m}} = (\hat{S}, \text{Init}, \hat{\rightarrow})\) the transition system of the abstracted SAS module \(\hat{m}\) given by the abstract syntactic representation \(\hat{\alpha}(\text{Descr}[m])\). The simulation relation for data domain abstraction \(R \subseteq S \times \hat{S}\) is defined by

\[
R(s, \hat{s}) \quad \text{iff} \quad \text{for all } v \in \text{Var}(m), \text{ it holds that } \alpha(s(v)) \in \hat{s}(v)
\]

In order to prove that local transitions preserve the similarity between abstract and concrete states, we first have to verify auxiliary lemmata that the syntactic abstraction is correct assuming that the safety conditions on abstract operations and predicates are satisfied. The following lemma formulates that the syntactic abstraction of expressions is correct. This means that given similar concrete and abstract states, first evaluating the concrete expression in the concrete state and then abstracting the result is in the set of values resulting when evaluating the abstract expression in the abstract state. The proof is by induction on the structure of expressions and uses the safety condition for the abstract operations.

**Lemma 5.43 (Correctness of Syntactic Expression Abstraction).** Let \(s \in S\) be a state of the original module \(m\) with \(T_m = (S, \text{Init}, \rightarrow)\) and \(\hat{s} \in \hat{S}\) a state of the abstract module \(\hat{m}\) with \(T_{\hat{m}} = (\hat{S}, \text{Init}, \hat{\rightarrow})\) such that \(R(s, \hat{s})\). For an expression, \(expr \in \text{Expr}\), it holds that \(\alpha(\text{eval}(\text{expr})[s]) \in \text{eval}(\hat{\alpha}(\text{expr}))[\hat{s}]\).
5.3. Data Domain Abstractions

Proof. The proof is by induction on the structure of the expression \( expr \).

- Induction Base: For variables and values, the result is clear by assumption that \( \mathcal{R}(s, \hat{s}) \).
- Induction Step: For operations, the result follows from induction hypothesis and the safety condition on the abstract operations \( \alpha(b(expr_1, expr_2)) \in \hat{b}(\alpha(expr_1), \alpha(expr_2)) \).

\[ \square \]

The next lemma formulates that the abstraction of constraints is correct. This means that the result of evaluating a concrete constraint over a state of a concrete module is contained in the set of values when evaluating the abstracted constraint over a state of the abstract module if the concrete and abstract state are similar. The proof is by induction on the structure of the constraint. It uses the previous lemma that the syntactic abstraction of expressions is correct and the safety condition on predicates. Furthermore, we take the unknown value as an abbreviation for the set \{true, false\}.

Lemma 5.44 (Correctness of Syntactic Constraint Abstraction). Let \( s \in S \) be a state of the original module \( m \) with \( T_m = (S, Init, \rightarrow) \) and \( \hat{s} \in \hat{S} \) a state of the abstract module \( \hat{m} \) with \( T_{\hat{m}} = (\hat{S}, Init, \hat{\rightarrow}) \) such that \( \mathcal{R}(s, \hat{s}) \). For a constraint \( c \in \text{Constraint} \), it holds that \( \text{eval}(c)[s] \in \text{eval}(\hat{\alpha}(c))[\hat{s}] \).

Proof. The proof is by induction on the structure of the constraint \( c \).

- Induction Base: For atomic constraints, the result follows from the safety condition for predicates \( \mathcal{R}(expr_1, expr_2) \in \hat{\mathcal{R}}(\alpha(expr_1), \alpha(expr_2)) \) and the correctness of abstraction for the enclosed expressions \( expr_1 \) and \( expr_2 \) by Lemma 5.43.
- Induction Step: For negations and conjunctions, the result follows from the definition of the abstract Boolean operators where unknown represents the set \{true, false\}. The abstract Boolean operators satisfy the safety condition on operations by definition.

\[ \square \]

The next lemma formulates that the syntactic abstraction of functions is correct. This means that if we have a concrete state \( s \) and an abstract state \( \hat{s} \) that are similar and in the concrete there is a transition defined by a syntactic function representation \( \text{func} \) leading to a resulting state \( s_{i+1} \), then there is an abstract state \( \hat{s}_{i+1} \) in the evaluation of the abstracted function \( \hat{\text{func}} \) such that the resulting states are also similar. The proof is by induction on the structure of functions.

Lemma 5.45 (Correctness of Syntactic Function Abstraction). Let \( m \) denote an original module with \( T_m = (S, Init, \rightarrow) \) and \( \hat{m} \) an abstract module
with $T_m = (\hat{S}, \hat{\text{Init}}, \hat{\rightarrow})$. For all $s_i, s_{i+1} \in S$ and $\hat{s}_i \in \hat{S}$ such that $R(s_i, \hat{s}_i)$, if $s_{i+1} = \text{eval}(func)[s_i]$, then there exists $\hat{s}_{i+1} \in \hat{S}$ such that $\hat{s}_{i+1} \in \text{eval}(func)[\hat{s}_i]$ and $R(s_{i+1}, \hat{s}_{i+1})$.

**Proof.** We show this lemma by induction on the structure of the function $func$.

- **Simple Assignment:** For a variable $x \in \text{Var}(m)$, $func = x := expr$ and $\hat{\alpha}(func) = x := eexpr$. We have to show that $\alpha(\text{eval}(expr)[s_i]) \in \text{eval}(\hat{\text{expr}})[\hat{s}_i])$ which immediately follows from Lemma 5.43. Hence, $\alpha(s_{i+1}(x)) \in \hat{s}_{i+1}(x)$ and thus $R(s_{i+1}, \hat{s}_{i+1})$.

- **Conditional Assignment:** Let $func = x := [\text{cond}_1, \ldots, \text{cond}_n]$ be a conditional assignment to a variable $x \in \text{Var}$. We have to show that the first conditional in the concrete whose constraint is true has an equivalent conditional in the abstract whose constraint is true or unknown and all earlier conditional constraints are unknown or false. This means, we have to show for a constraint $c$ that $\text{eval}(c)[s_i] \in \text{eval}(\hat{\alpha}(c))[\hat{s}_{i+1}]$. This follows from Lemma 5.44 stating that the abstraction of constraints is correct. The correctness of the evaluation of the corresponding abstracted expression follows from Lemma 5.43.

- **Function Lists:** $func = [a_1, \ldots, a_n]$. The proof for this case is by induction on the length of the function list. The induction base is one of the previous two cases. The induction step follows from the definition of the evaluation of function lists. \qed

In order to show that an original SAS module and its abstraction are similar, we have to establish that for each initial state of the original system, there exists a similar initial state in the abstract system. Secondly, we have to show that the simulation relation is preserved by the transition relation.

**Theorem 5.46 (Simulation for Abstract Modules).** Let $T_m = (S, \text{Init}, \rightarrow)$ denote an SAS module transition system for the module $m$ given by the syntactic representation $\text{Descr}[m]$ and $T_{\hat{m}} = (\hat{S}, \text{Init}, \hat{\rightarrow})$ the transition system for the abstracted SAS module $\hat{m}$ given by the abstracted syntactic representation $\hat{\alpha}(\text{Descr}[m])$. Then it holds that

$$T_m \preceq T_{\hat{m}}$$

**Proof.** We take the simulation relation $R \subseteq S \times \hat{S}$ as in Definition 5.42 where for all $v \in \text{Var}(m)$ it holds that $\alpha(s(v)) \subseteq \hat{s}(v)$. 

• Initial States: For an initial state $s_0 \in \text{Init}$, it holds for $v \in \text{Var}_d(m)$ that $s(v) = \text{init}(v)$ and for $v \in \text{Var}_a(m)$ that $s(v) = \text{adapt}_\text{init}(v)$. For $\hat{s}_0 \in \hat{\text{Init}}$, it holds that for $v \in \text{Var}_d(\hat{m})$ that $\hat{s}_0(v) = \alpha(\text{init}(v))$ and for $v \in \text{Var}_a(\hat{m})$ that $\hat{s}_0(v) = \alpha(\text{adapt}_\text{init}(v))$. Hence, for an initial state $s_0 \in S$ there exists $\hat{s}_0 \in \hat{S}$ such that $R(s_0, \hat{s}_0)$.

• Step Simulation: Let $(s_i, \hat{s}_i) \in R$ and $s_i \rightarrow s_{i+1}$. We have to show that there exists $\hat{s}_{i+1} \in \hat{S}$ such that $\hat{s}_i \rightarrow^{\hat{s}} \hat{s}_{i+1}$. We assume that $\alpha(s_{i+1}(v)) \in \hat{s}_{i+1}(v)$ for $v \in \text{in} \cup \text{adapt}_\text{in}$. 
  – First, we have to show that the adaptive variables are in relation, i.e. $\alpha(s_{i+1}(v)) \in \hat{s}_{i+1}(v)$ for $v \in \text{adapt}_\text{loc}$ and for $v \in \text{adapt}_\text{out}$. This follows from the above Lemma 5.45 that the syntactic function abstraction of $\text{adapt}_\text{next}_\text{state}$ and $\text{adapt}_\text{next}_\text{out}$ is correct on similar states where the input variables are interpreted in the successor state.
  – Second, we have to show that there is at least one transition in the abstract module such that the same configuration activated in the concrete module is also activated in the abstract module. Let $k$ denote the index of the configuration chosen in the concrete module. For all $0 < j < k$, it holds that $\text{eval}(\text{guard}_j)[s_{i+1}|\text{adapt}_\text{in} \cup s_i|\text{adapt}_\text{loc}] = \text{false}$ and $\text{eval}(\text{guard}_i)[s_{i+1}|\text{adapt}_\text{in} \cup s_i|\text{adapt}_\text{loc}] = \text{true}$. From the previous Lemma 5.45, we know that the abstract evaluation of constraints is correct on similar states, thus in the abstract module, it holds that $\text{false} \in \hat{\text{eval}}(\text{guard}_j)[\hat{s}_{i+1}|\text{adapt}_\text{in} \cup \hat{s}_i|\text{adapt}_\text{loc}]$ and that $\text{true} \in \hat{\text{eval}}(\text{guard}_k)[\hat{s}_{i+1}|\text{adapt}_\text{in} \cup \hat{s}_i|\text{adapt}_\text{loc}]$. So there is one state in which the same configuration is chosen.
  – Third, we have to show that the functional variables are in relation, i.e. $\alpha(s_{i+1}(v)) \in \hat{s}_{i+1}(v)$ for $v \in \text{loc}$ and $v \in \text{out}$. This follows from above Lemma 5.45 that the syntactic function abstraction of $\text{next}_\text{state}_k$ and $\text{next}_\text{out}_k$ is correct and the previous consideration that there exists a state such that the same configuration in the abstract is chosen as in the concrete. Combining the three results completes the proof. □

5.3.3 Abstracting Model Properties

In order to verify properties using data domain abstraction, we must not only abstract a module (or a system), but also the property to be verified in a property-preserving way. If we can verify an abstract property over an abstract module, we want to be able to infer that the original module satisfies the original property. Hence, property abstraction has to construct under-approximations of concrete properties with respect to atomic propositions.
expressible in the abstract module. In Section 5.3.4, we lift abstractions of modules and their properties to the system level. Abstraction of properties over SAS systems proceeds in the same way as abstraction for properties over modules.

The selection of an appropriate abstraction in a particular verification task strongly depends on the property to be verified. An abstraction must be chosen such that the property is expressible in the abstract domain. If a property is abstracted to false, as false is the best possible underapproximation, the abstraction is too coarse. Instead, the abstraction should ensure that every atomic proposition in the concrete property is expressible by a disjunction of abstract atomic propositions.

For a concrete module \( m \) over a set of variables \( \text{Var}(m) \) and a set of values \( \text{Val} \), the properties over this module refer to variables \( v \in \text{Var}(m) \) and values \( \text{val} \in \text{Val} \). An abstract module \( \hat{m} \) over a finite abstract domain \( \hat{\text{Val}} \) with the same set of variables \( \text{Var}(\hat{m}) = \text{Var}(m) \) can be checked for properties over variables \( v \in \text{Var}(\hat{m}) \) and abstract values \( \hat{\text{val}} \in \hat{\text{Val}} \). As atomic propositions for properties over an abstract module, we only allow equalities between variables and abstract values \( v = \hat{\text{val}} \) along the lines of [KP98] for a uniform property abstraction. Additionally, we do not allow negation in an abstract property as negation is not preserved by abstraction. Instead, negation of an atomic proposition \( \neg(v = \hat{\text{val}}) \) is interpreted as the disjunction over the abstract atomic propositions expressible over \( v \), \( \hat{\text{Atoms}}(v) = \{ v = \text{val} \mid \text{val} \in \hat{\text{Val}} \} \), except from the negated atomic proposition. This construction is possible if the abstract domain \( \hat{\text{Val}} \) is finite. This means that \( \neg(v = \hat{\text{val}}) \) is used as an abbreviation for

\[
\neg(v = \hat{\text{val}}) := \bigvee \hat{\text{Atoms}}(v) \setminus \{ v = \hat{\text{val}} \}
\]

As the interpretation of values changes by abstraction, we need a concretisation of abstract properties mapping them to properties of the concrete module. The concretisation has to ensure that the abstraction is property-preserving. Thus, an abstract atomic proposition \( v = \text{val} \) for a variable \( v \in \text{Var}(\hat{m}) \) and an abstract value \( \hat{\text{val}} \in \hat{\text{Val}} \) is concretised to a proposition of a concrete module by a big disjunction of equalities between the variable \( v \) and all concrete values \( \text{val} \in \text{Val} \) that are mapped to the abstract value \( \text{val} \) by the abstraction function \( \alpha \), i.e. \( \text{val} = \alpha(\text{val}) \). For example, assume as abstract domain the signs of the integers, i.e. \( \hat{\text{Val}} = \{ \text{pos, neg, zero} \} \). The set of abstract atomic propositions over this abstract domain is \( \{ v = \text{val} \mid v \in \text{Var}(\hat{m}), \text{val} \in \{ \text{pos, neg, zero} \} \} \). The con-
cretisation of the abstract atomic proposition $x = \text{pos}$ to the integers is $C(x = \text{pos}) = \bigvee_{\text{val}>0} (x = \text{val})$.

In order to obtain an underapproximation of the concrete property $\varphi$, an atomic proposition $a = r(expr_1, expr_2)$ in $\varphi$ is abstracted to a disjunction over all abstract atomic propositions $\hat{a}$ whose concretisations $C(\hat{a})$ imply the concrete atomic proposition, i.e. $C(\hat{a}) \rightarrow a$. If there is no atomic proposition $\hat{a}$ whose concretisation implies an atomic proposition, $a$ is abstracted to false. False is the smallest underapproximation of a concrete atomic proposition. In this case, precision is lost by the abstraction. In order to prevent underapproximating properties by false, the abstraction has to be chosen such that it allows representing a concrete property as precisely as possible. Restricting the abstract atomic propositions to equalities of variables and abstract values disallows expressing relations between variables. For example, the atomic proposition $x = y$ for two variables $x$ and $y \in Var(m)$ is abstracted to false. This is due to the fact that by abstraction we loose track of the concrete values a variable can assume. However, with sign abstraction we can for instance express the property that the signs of the variables $x$ and $y$ coincide as

$$(x = \text{pos} \leftrightarrow y = \text{pos}) \land (x = \text{zero} \leftrightarrow y = \text{zero}) \land (x = \text{neg} \leftrightarrow y = \text{neg})$$

where $\text{pos}$ denotes positive integers, $\text{neg}$ negative integers, and $\text{zero}$ encodes the value 0.

A concrete property $\varphi \in \mathcal{AL}_{\text{SAS}}$ over a concrete domain $\text{Val}$ is abstracted to an abstract property $\hat{\varphi}$ over an abstract domain $\widehat{\text{Val}}$ by abstracting positive atomic propositions as described above. Negated atomic propositions are abstracted to a disjunction over all abstract atomic propositions that imply the negated atomic proposition or false. For the remaining operators, the property structure is preserved. This ensures that the concretisation of an abstract property implies the original property. The process of abstracting properties can be automatised using a decision procedure for checking implications (cf. [GS97]).

**Definition 5.47 (Property Abstraction).** For a set of variables $\text{Var}$ and an abstract domain $\widehat{\text{Val}}$, the set of abstract atomic propositions $\widehat{\text{Atoms}}$ is defined by

$$\widehat{\text{Atoms}} = \{ v = \hat{\text{val}} \mid v \in \text{Var}, \hat{\text{val}} \in \widehat{\text{Val}} \}$$
The concretisation of an abstract atomic proposition \( \hat{a} = (v = \hat{val}) \in \hat{Atoms} \) is defined by
\[
C(\hat{a}) = \bigvee_{\alpha(val) = val} (v = val)
\]

Let \( \varphi \) be a concrete property in \( \mathcal{AL}_{SAS} \) with a set of atomic propositions \( \text{Atoms}(\varphi) \). We define the property abstraction function \( \alpha \) inductively on the formula structure of \( \mathcal{AL}_{SAS} \). For \( a \in \text{Atoms}(\varphi) \), \( \alpha(a) \) is defined by
\[
\alpha(a) = \bigvee \{ \hat{a} \in \hat{Atoms} \mid C(\hat{a}) \rightarrow a \} \lor \text{false}
\]

For \( \varphi = \neg a \) with \( a \in \text{Atoms}(\varphi) \), \( \alpha(\varphi) \) is defined by
\[
\alpha(\varphi) = \bigvee \{ \hat{a} \in \hat{Atoms} \mid C(\hat{a}) \rightarrow \neg a \} \lor \text{false}
\]

A property \( \varphi \in \mathcal{AL}_{SAS} \) is further abstracted inductively by \( \alpha \)
- \( \alpha(\varphi_1 \land \varphi_2) = \alpha(\varphi_1) \land \alpha(\varphi_2) \)
- \( \alpha(\varphi_1 \lor \varphi_2) = \alpha(\varphi_1) \lor \alpha(\varphi_2) \)
- \( \alpha(\mathsf{X}\varphi) = \mathsf{X} \alpha(\varphi) \)
- \( \alpha(\varphi_1 \mathsf{U} \varphi_2) = \alpha(\varphi_1) \mathsf{U} \alpha(\varphi_2) \)
- \( \alpha(\varphi_1 \mathsf{R} \varphi_2) = \alpha(\varphi_1) \mathsf{R} \alpha(\varphi_2) \)
- \( \alpha(\mathsf{A}\varphi) = \mathsf{A} \alpha(\varphi) \)

For a set of variables \( \text{Var} = \{x, y\} \) and the set of integers \( \text{Val} \), consider as example the \( \mathcal{L}_{SAS} \) property \( \varphi \equiv \mathsf{A} \mathsf{G} (x \geq 0 \rightarrow \mathsf{X}(y < 0)) \). For sign abstraction, we use the abstract domain \( \hat{\text{Val}} = \{\text{pos}, \text{zero}, \text{neg}\} \) and the abstraction function \( \alpha \) where
\[
\alpha(val) = \begin{cases} 
\text{pos} : & val > 0 \\
\text{zero} : & val = 0 \\
\text{neg} : & val < 0 
\end{cases}
\]

The set of abstract atomic propositions over the abstract domain is \( \hat{\text{Atoms}} = \{x = \text{pos}, x = \text{zero}, x = \text{neg}, y = \text{pos}, y = \text{zero}, y = \text{neg}\} \). Abstraction the atomic propositions of \( \varphi \) yields

\[
\alpha(x \geq 0) = (x = \text{pos} \lor x = \text{zero})
\]
since \( C(x = \text{pos}) = \bigvee_{v>0}(x = v) = (x > 0) \rightarrow x \geq 0 \) and \( C(x = \text{zero}) = (x = 0) \rightarrow x \geq 0 \) and
\[
\alpha(y < 0) = (y = \text{neg})
\]
since \( C(y = \text{neg}) = \bigvee_{v<0}(y = v) \rightarrow y < 0 \).
Hence, the abstraction of $\varphi$ is $\alpha(\varphi) = A\ G((x = \text{pos} \lor x = \text{zero}) \rightarrow X(y = \text{neg}))$. The concretisation of this abstraction implies $\varphi$:

$$C(\alpha(\varphi)) = A\ G((\bigvee_{v>0}(x = v) \lor (x = 0)) \rightarrow X(\bigvee_{v<0}(y = v)))$$

$$\rightarrow A\ G(x \geq 0 \rightarrow X(y < 0))$$

Now consider the $L_{SAS}$ property $\psi \equiv A\ G(x \geq 2)$. The property $\psi$ cannot be expressed by atomic propositions in $\hat{\text{Atoms}}$. The reason is that there is no atomic proposition that implies $x \geq 2$. Hence, $\psi$ will be abstracted to $\text{false}$ as this is the weakest underapproximation. Certainly, $\text{false}$ implies $\psi$, but as $\text{false}$ is not satisfied by any module the abstraction is not suitable for verifying $\psi$. For the property $\psi$, a range abstraction is more appropriate. We take as abstract domain the set $\hat{\text{Val}} = \{\text{LE2}, \text{GEQ2}\}$ where

$$\alpha(val) = \begin{cases} 
\text{LE2} : & val < 2 \\
\text{GEQ2} : & val \geq 2
\end{cases}$$

Then $\psi$ can be abstracted to

$$\alpha(\psi) = A\ G(x = \text{GEQ2}) \text{ since } C(x = \text{GEQ2}) = \bigvee_{v \geq 2} (x = v) \rightarrow x \geq 2$$

At this example, we can observe that choosing the right abstraction for a property to be verified is crucial to be able to establish the property over an abstract module. The next lemma states that the concretisation of an abstract property implies the original property. This is an important result for property preservation of data domain abstractions.

**Lemma 5.48 (Implication of Original Property by Concretisation).** Let $\varphi \in AL_{SAS}$ be a property over a concrete module (or system) $T$ over the variables $\text{Var}$ and let $\hat{\varphi} = \alpha(\varphi)$ be the abstraction of $\varphi$ with respect to an abstract domain $\hat{\text{Val}}$. Then it holds that

$$C(\alpha(\varphi)) \rightarrow \varphi$$

with $C$ defined as in Definition 5.47

**Proof.** The proof is by induction on the formula structure of $\varphi$.

- **Induction Base:**
  - $\varphi = a$ for a atomic:
    By definition of $\alpha(a) = \bigvee\{\hat{a} \in \text{Atoms} | C(\hat{a}) \rightarrow a\} \lor \text{false}$, we obtain the result that $C(\alpha(a)) \rightarrow a$. 

\( \varphi = \neg a \) for a atomic:

By definition of \( \alpha(\varphi) = \bigvee\{\hat{a} \in \text{Atoms} \mid C(\hat{a}) \rightarrow \neg a\} \lor \text{false} \), we obtain that \( C(\alpha(\varphi)) \rightarrow \varphi \).

- Induction Step:
  - \( \varphi = (\varphi_1 \land \varphi_2) \): By definition of \( \alpha \), we have that \( \alpha(\varphi) = \alpha(\varphi_1) \land \alpha(\varphi_2) \), and by induction hypothesis and structure preservation of \( C \), that \( C(\alpha(\varphi_1) \land \alpha(\varphi_2)) \rightarrow (\varphi_1 \land \varphi_2) \).
  - \( \varphi = (\varphi_1 \lor \varphi_2) \): By definition of \( \alpha \), we have that \( \alpha(\varphi) = \alpha(\varphi_1) \lor \alpha(\varphi_2) \), and by induction hypothesis and structure preservation of \( C \), that \( C(\alpha(\varphi_1) \lor \alpha(\varphi_2)) \rightarrow (\varphi_1 \lor \varphi_2) \).
  - \( \varphi = X\varphi \): By definition of \( \alpha \), we have that \( \alpha(\varphi) = X\alpha(\varphi) \), and by induction hypothesis, that \( C(X\alpha(\varphi)) \rightarrow X\varphi \).
  - \( \varphi = (\varphi_1 U \varphi_2) \): By definition of \( \alpha \), we have that \( \alpha(\varphi) = \alpha(\varphi_1) U \alpha(\varphi_2) \), and by induction hypothesis and structure preservation of \( C \), that \( C(\alpha(\varphi_1) U \alpha(\varphi_2)) \rightarrow (\varphi_1 U \varphi_2) \).
  - \( \varphi = (\varphi_1 R \varphi_2) \): By definition of \( \alpha \), we have that \( \alpha(\varphi) = \alpha(\varphi_1) R \alpha(\varphi_2) \), and by induction hypothesis and structure preservation of \( C \), that \( C(\alpha(\varphi_1) R \alpha(\varphi_2)) \rightarrow (\varphi_1 R \varphi_2) \).
  - \( \varphi = A\varphi \): By definition of \( \alpha \), we have that \( \hat{\varphi} = A\alpha(\varphi_1) \). By induction hypothesis and structure preservation of \( C \), \( C(A\alpha(\varphi_1)) \rightarrow A\varphi \). \( \square \)

Now, it remains to show that properties abstracted as defined above are preserved under abstraction. Therefore, we have to show that property abstraction is consistent with the previously defined simulation between concrete and abstract modules. However, abstraction does not preserve negation of atomic propositions between similar states. Thus, we can only show that abstraction induces a positive consistent simulation denoted by \( \equiv^{+}_{\{\hat{a}\}} \).

For a positive consistent simulation \( \hat{T} \equiv^{+}_{\{\hat{a}\}} T \), the consistency condition requires that for all abstract atomic propositions \( \hat{a} \in \text{Atoms}(\hat{\varphi}) \), if \( R(\sigma, \hat{\sigma}) \), it holds that \( (\hat{T}, \hat{\sigma}) \models \hat{a} \) implies \( (T, \sigma) \models C(\hat{a}) \). This yields that under positive consistent simulation only \( A\mathcal{L}_{\text{SAS}} \) properties without negated atomic propositions are preserved, since in the proof of Theorem 5.7, the reverse direction of the consistency condition is required in order to establish preservation of negated atomic propositions. However, for property preservation by data domain abstraction, this limitation is not relevant as by definition of property abstraction, abstract properties do not contain negations.

**Theorem 5.49 (Positive Consistent Simulation for Data Domain Abstraction).** Let \( T_m \) denote an SAS module transition system \( T_m = (S, \text{Init}, \rightarrow) \) for the module \( m \) given by the syntactic representation.
5.3. Data Domain Abstractions

Let $\text{Descr}[m]$, and let $\varphi$ be an $\mathcal{AL}_{\text{SAS}}$ property over $m$. Let $T_m = (\hat{S}, \hat{\text{Init}}, \hat{\to})$ be the abstract SAS module $\hat{m}$ given by the abstract syntactic representation $\hat{\alpha}(\text{Descr}[m])$, and let $\hat{\varphi}$ be the abstraction of $\varphi$, i.e. $\hat{\varphi} = \alpha(\varphi)$. Then, it holds that $T_m$ and $T_{\hat{m}}$ are in positive consistent simulation with respect to $\hat{\varphi}$, i.e.

$$T_m \preceq^+_{[\hat{\varphi}]} T_{\hat{m}}$$

**Proof.** We use the result from Theorem 5.46 that $(s, \hat{s}) \in R \subseteq S \times \hat{S}$ iff for all $v \in \text{Var}(m)$ we have that $\alpha(s(v)) \in \hat{s}(v)$ constitutes a simulation relation. We now have to show that the relation $R$ is positively consistent with respect to $\hat{\varphi}$, i.e. for all $\hat{a} \in \text{Atoms}(\hat{\varphi})$, if $R(s, \hat{s})$ and $\hat{s} \models \hat{a}$, then $s \models C(\hat{a})$.

Let $\mathcal{R}(s, \hat{s})$ and $\hat{s} \models (v = \text{val})$. The concretisation of $v = \text{val}$ is defined by $C(v = \text{val}) = \bigvee_{\alpha(\text{val}) = \text{val}}(v = \text{val})$. As we have $\mathcal{R}(s, \hat{s})$, we know that for all $v \in \text{Var}(m)$, it holds that $\alpha(s(v)) \in \hat{s}(v)$. So there must be a $\text{val} \in \text{Val}$ such that $s \models (v = \text{val})$ and hence $s \models C(\hat{a})$. □

From the positive consistent simulation and property preservation by positive consistent simulation for positive $\mathcal{AL}_{\text{SAS}}$ properties (cf. Theorem 5.8 and Corollary 5.11 without negation of atomic propositions), we can conclude that the proposed data domain abstraction preserves $\mathcal{AL}_{\text{SAS}}$ properties. Hence, it suffices to verify an abstract property over an abstract module in order to establish the validity of the original property over the original module.

**Corollary 5.50 (Property Preservation for Data Domain Abstraction).** Let $T_m = (S, \text{Init}, \to)$ denote an SAS module transition system for a module $m$ given by the syntactic representation $\text{Descr}[m]$, and let $\varphi$ be an $\mathcal{AL}_{\text{SAS}}$ property over $m$. Let $T_{\hat{m}} = (\hat{S}, \hat{\text{Init}}, \hat{\to})$ be the abstracted SAS module given by the abstract syntactic representation $\hat{\alpha}(\text{Descr}[m])$, and let $\hat{\varphi}$ be the abstraction of $\varphi$, i.e. $\hat{\varphi} = \alpha(\varphi)$. Then

$$T_{\hat{m}} \models \hat{\varphi} \text{ implies } T_m \models \varphi$$

**Proof.** By Theorem 5.49, we know that $T_m \preceq^+_{[\hat{\varphi}]} T_{\hat{m}}$. Since $\hat{\varphi} = \alpha(\varphi)$, we know that $\hat{\varphi}$ does not contain negated atomic propositions. Hence, by Corollary 5.11 for properties without negation, we know that $T_{\hat{m}} \models \hat{\varphi}$ implies $T_m \models C(\hat{\varphi})$. From the implication of $C(\hat{\varphi}) \rightarrow \varphi$ in Lemma 5.48, we obtain the result. □

5.3.4 Abstraction for Verification of System Properties

In order to use data domain abstraction for verification of SAS system properties, the results on abstractions of SAS modules have to be transferred
to the system level. In this section, we extend data domain abstraction to SAS systems and discuss when an abstraction is optimal with respect to a property. Additionally, we present a library of abstractions for SAS systems in order to alleviate selecting abstractions for concrete verification tasks.

**System Abstractions**

Up to now, we have only considered data domain abstractions of SAS modules. However, a verification task can also refer to an SAS system consisting of more than one module. Then, it is necessary to abstract the complete system in order to use model checking efficiently for verification. In this direction, all modules \( \{m_1, \ldots, m_n\} \) contained in the original system \( SAS \) are transformed to their abstractions \( \{\hat{m}_1, \ldots, \hat{m}_n\} \) using the same abstraction \( \hat{\alpha} \) on the syntactic descriptions \( \text{Descr}[m_i] \). The abstract system \( \hat{SAS} \) is built from the original system \( SAS \) by replacing the concrete modules with their abstractions. This construction follows Definition 5.13. As the abstraction only changes the data domains of variables and not the variables themselves, connections between modules and system border variables are unchanged such that the resulting system \( \hat{SAS}' \) is valid.

The correctness of abstracting a system \( SAS \) containing the modules \( \{m_1, \ldots, m_n\} \) by replacing each module by its abstraction follows from Theorem 5.14. It allows lifting the consistent simulation relations between the concrete module \( m_i \) and the abstract module \( \hat{m}_i \) with respect to an abstract property \( \hat{\varphi}_i \) to the system \( \hat{SAS}' \) built from the abstract modules \( \hat{m}_i \). This guarantees that if the abstract property \( \hat{\varphi} = \alpha(\varphi) \) holds over the abstract system \( \hat{SAS}' \), the concretisation \( C(\hat{\varphi}) \) holds over the original system \( SAS \). The validity of the concrete property \( \varphi \) on \( SAS \) follows from the fact that \( C(\alpha(\varphi)) \) implies \( \varphi \) by Lemma 5.48. Theorem 5.14 requires that the abstract property \( \hat{\varphi} \) may only contain atomic propositions \( \hat{a} \in \bigcup_{i=1}^n \text{Atoms}(\hat{\varphi}_i) \) for a consistent simulation where \( \hat{\varphi}_i \) refers only to module \( m_i \). This restriction is satisfied by design of abstract properties because any abstract atomic proposition is an equality of a variable and an abstract value such that it only refers to one module.

**Corollary 5.51 (Property-Preservation for System Abstraction).** Let \( SAS = (M, \text{input}_a, \text{input}_d, \text{output}_a, \text{output}_d, \text{conn}_a, \text{conn}_d) \) be a system containing the modules \( M = \{m_1, \ldots, m_n\} \) and let \( \{\hat{m}_1, \ldots, \hat{m}_n\} \) be the abstractions of the modules \( \{m_1, \ldots, m_n\} \) using the abstraction \( \hat{\alpha} \) on the syntactic descriptions \( \text{Descr}[m_i] \) such that \( \hat{SAS}' \) is the system constructed from \( SAS \) by replacing the concrete modules \( m_i \) by their abstractions \( \hat{m}_i \) according to
5.3. Data Domain Abstractions

**Definition 5.13.** Let \( \varphi \) be an ALC\(_{\text{SAS}}\) property over \( \text{SAS} \) and \( \hat{\varphi} \) the abstraction of \( \varphi \) with respect to \( \alpha \). Then

\[
T_{\text{SAS}'} \models \hat{\varphi} \implies T_{\text{SAS}} \models \varphi
\]

**Proof.** By Theorem 5.14 with the consistency condition restricted to the positive direction, we can lift the positive consistent simulations with respect to \( \hat{\varphi} \) from module to system level because all atomic propositions of \( \hat{\varphi} \) only refer to one module and \( \text{SAS}' \) is a valid system as abstraction does not change module interfaces. This yields that \( T_{\text{SAS}} \preceq_{\hat{\varphi}} T_{\text{SAS}'} \). By instantiating Theorem 5.7 to the initial states and the fact that \( \hat{\varphi} \) by construction does not contain negated atomic propositions, we know that \( T_{\text{SAS}'} \models \hat{\varphi} \implies T_{\text{SAS}} \models C(\hat{\varphi}) \). The implication \( C(\hat{\varphi}) \rightarrow \varphi \) follows from 5.48 which completes the proof. \( \square \)

In the brake and gas module example, sign abstraction of the property that brake and gas values are mutually exclusive shown in Section 3.2.2 yields the following abstract property.

\[
\forall G ((\text{useconf}_{\text{Brake}} \neq \text{Off} \land  \text{useconf}_{\text{Gas}} \neq \text{Off}) \rightarrow (\text{brake} = \text{pos} \rightarrow \text{gas} = \text{zero}) \land (\text{gas} = \text{pos} \rightarrow \text{brake} = \text{zero}))
\]

Note that the implication \((v = \hat{v}_1) \rightarrow (w = \hat{v}_2)\) is a short hand notation for \( \neg (v = \hat{v}_1) \lor (w = \hat{v}_2) \) where \( \neg (v = \hat{v}_1) \) is used as an abbreviation for a disjunction over all abstract atomic propositions over \( v \) without \( v = \hat{v}_1 \), i.e. \( \lor \text{Atoms}(v) \setminus \{v = \hat{v}_1\} \). This is admissible as the abstract domain in sign abstraction is finite. The concretisation of this property implies the original property in the original model. If the abstract property holds over the abstract brake and gas modules with respect to sign abstraction, we can infer that also the original property holds over the concrete modules. In Section 4.2, it is verified that brake and gas values are mutually exclusive by theorem proving in the concrete model. However, sign abstraction allows reducing the model to finite-state making it amenable to model checking.

**Optimal Abstractions**

The selection of an abstraction appropriate for a particular verification task depends on the system and the property to be verified. An abstraction must be chosen such that the property is expressible in the abstract domain. Furthermore, an abstraction should ensure that if the original property is valid in the original module, the abstract property can be established over the
abstraction. An *optimal abstraction* of an SAS module is an abstract SAS module in which the abstraction of the original property holds if the original property holds in the original module and which simulates all other modules for which the abstract property holds. This means that an optimally abstract module is the largest module with respect to simulation.

**Definition 5.52 (Optimal Abstraction).** Let $T_m$ be a concrete module transition system for a module $m$ given by the syntactic representation $\text{Descr}[m]$. Let $\varphi$ be a universal $L_{SAS}$ property over $m$ and $\hat{\varphi}$ the abstract property, i.e. $\hat{\varphi} = \alpha(\varphi)$. The abstract model transition system $T_{\hat{m}}$ for a module $\hat{m}$ given by the syntactic representation $\hat{\alpha}(\text{Descr}[m])$ is an *optimal abstraction* if the following conditions hold:

- if $T_m \models \varphi$ then $T_{\hat{m}} \models \hat{\varphi}$.
- for all other abstract module transition systems $T_{m'}$ for modules $m'$ with $T_{m'} \models \hat{\varphi}$, it holds that $T_{m'} \preceq_{\hat{\varphi}} T_{\hat{m}}$ with respect to the identity function as concretisation $C$.

The definition of an optimal abstraction can also be extended to SAS systems in a straightforward manner. An SAS system $\hat{SAS}$ is an optimal abstraction for a concrete system $SAS$ with respect to a property $\varphi$ and its abstraction $\hat{\varphi}$, if first $\hat{SAS} \models \hat{\varphi}$, and furthermore, $\hat{SAS}$ is the largest system satisfying this property in the simulation ordering.

**Finding Abstractions**

A property-preserving data domain abstraction on SAS systems is defined by an abstract domain $\hat{\text{Val}}$, an abstraction mapping $\alpha$ assigning concrete values to abstract values from the abstract domain $\hat{\text{Val}}$ and abstract interpretations for the predicates and operation symbols over the power set of the abstract values satisfying the safety conditions. An abstraction can be described by a tuple $\text{Abs} = (\hat{\text{Val}}, \alpha, \hat{\text{Rel}}, \hat{\text{BinOp}})$. For a specific abstraction $\text{Abs}$, these items can be stored in an abstraction library and instantiated for a concrete system to be abstracted. An abstraction library contains popular abstractions for integers or real numbers to be applied on SAS such that the user does not have to define an abstraction from scratch, but can reuse predefined items. This alleviates the usability of abstractions.

Data domain abstractions can be classified into different families: range, set, modulo, point and k-ordered abstractions [DHJ+01, HDPR02]:

- **Range Abstraction** tracks concrete values between upper and lower bounds, but abstracts values smaller than the lower bound and greater
than the upper bound. Sign abstraction is a special case of the range abstraction where the lower and upper bound are zero. The abstract domain for sign abstraction is the set \{neg, zero, pos\}

- **Set Abstraction** can be used instead of range abstraction if no operations on the abstract values have to be performed and only equality is used as predicate in atomic propositions. For instance, the integers can be mapped to two sets \{zero, non_zero\} where non_zero comprises the non zero integer values and zero the value 0.

- **Modulo-K Abstraction** maps integers having the same remainder modulo k into the same equivalence classes represented by an abstract value.

- **In Point Abstraction**, all values are collapsed to one token called unknown. This abstraction can be useful if only control flow in a system is to be considered and the property under consideration is data-independent.

- **By K-Ordered Data Abstraction**, the knowledge about the concrete data values is abstracted away. The only information that is kept is that there are k different values in an abstract system.

Recently, there have also been attempts in the literature to discover abstractions automatically. Counterexample-guided abstraction refinement [CGJ+03] is based on Boolean or predicate abstraction [GS97] of a system. Instead of abstracting data domains of variables in a state, the state as a whole is abstracted with respect to a set of assertions. The choice of the assertions depends on the property to be verified. The behaviour of a system is described by the evaluations of the assertions as system states and by transitions between these abstract states. A predicate abstraction of a system can be computed automatically by a decision procedure for the logic used to express the assertions. Counterexample-guided abstraction refinement [CGJ+03] starts with an initial set of assertions and constructs the abstract system using these assertions. Spurious counterexamples produced by model checking the abstract system give hints for assertions to be added in order to refine the abstraction. Then, the refined abstraction of the system is model checked again hoping that either the property can be verified or that a real counterexample is found. This process is performed iteratively, but it is not guaranteed to terminate. Counterexample-guided abstraction refinement would be an valuable approach for SAS systems as the system designer is assisted in finding a suitable system abstraction.
The prerequisite for this, however, is to use predicate abstraction for SAS systems instead of data domain abstraction.

5.4 Related Work

Verification complexity reduction by model transformations is used in order to make system models amenable to formal verification by model checking. In this section, we first review related work on justifying correctness of model transformations by simulation arguments, translation validation or general theorem proving techniques. Second, we consider similar approaches applying slicing techniques and abstractions to reduce verification complexity with respect to model checking.

Property-Preservation by Simulation and Translation Validation

Simulation for program correctness was originally introduced by Milner in [Mil71]. Since, property preservation by simulation has been studied for different fragments of CTL* [CGP99] and the \( \mu \)-calculus [BBLS93]. The underlying system models are Kripke structures where either states are labelled with atomic propositions or atomic propositions are labelled with states. This reduces the consistency condition to checking that the labelling of two states in simulation is a subset relation. However, this system model complicates the treatment of systems where states are defined by valuations of variables such as in SAS. In [CGL94], a system model similar to ours is used, but only applied for data domain abstractions while our technique can be applied for general system transformations.

In order to show that a system transformation on a concrete system instance is correct, we adopted the notion of translation validation [PSS98] originally used for guaranteeing correctness of compiler runs. After a compiler has translated a source program into a target program, a checker compares the two programs and decides whether they are equivalent. In our setting, we replace the compiler by the transformation procedure, the source program by the original system and the target program by the transformed system. Isabelle/HOL [NPW02] serves as checker in our case. In the original translation validation approach [PSS98], the checker derives the equivalence of source and target via static analysis while the compiler is regarded as a black box. In subsequent works, the compiler is extended to generate hints for the checker, e.g. proof scripts or a simulation relation as in our case, in order to simplify the derivation of equivalence.
of source and target programs. Translation validation in general is not limited to simulation-based correctness criteria, but for compilers as well as for model transformations simulation-based correctness criteria can be used (see e.g. [BGG05] for work using a similar Isabelle formalisation of simulation).

With respect to verification of system abstraction correctness using theorem provers, [DHS99] proposes an approach to verify abstract interpretation based abstractions with the INKA theorem prover. The theorem prover establishes safety conditions on the abstract operators in order to infer abstraction correctness from abstract interpretation theory. [SBLS99] uses the PVS theorem prover to show that an abstraction induces a simulation between original and abstract system. In contrast to our translation validation approach, in this work the verification of abstraction correctness has to be carried out manually after abstracting a system. Similarly, [RSS95] uses PVS to verify the correctness conditions for abstractions from [CGL94] for a concrete abstraction interactively. Müller and Nipkow [MN95] propose rules for proving correctness of abstractions of I/O automata in Isabelle/HOL using a meta-theory formalisation of I/O automata. The approach relies on checking that the set of traces of a concrete automaton is included in the set of traces of its abstraction.

Slicing for Model Reduction

Slicing has been introduced in [Wei84] as an effective technique for debugging and program comprehension in sequential programs. A survey of techniques can be found in [Tip95]. Recently, slicing has also been used for concurrent state-based system models, e.g. extended hierarchical automata [WDQ02, HW97, KSTV03] or input/output transition systems [LGP07]. However, most of these approaches work on relatively low-level model representations in contrast to SAS models capturing the high-level intuition of adaptive modelling concepts. In [CCIP06], slicing is used to reduce software architecture models expressed by UML state charts. However, the component-based structure of software architecture models is not exploited for determining irrelevant parts with respect to the considered property.

With respect to model reduction for model checking, cone of influence reduction, e.g. for Boolean circuits [CGP99], is applied to the internal model representations similar to static backward slicing. In the BAN-DERA [HDZ00] model checking environment for Java, program slicing techniques for concurrent Java programs are applied. Slicing for Promela,
the concurrent input language of the Spin model checker, is proposed in [MT00b].

In SAL [BGL+00], slicing on a low-level intermediate transition system representation is used for model reduction. Although SAL specifications have a modular structure, the modularity is not exploited for slicing. The IF [BGO+04] framework applies static analysis techniques for model reduction on a more high-level intermediate model representation. However, the main focus lies on property independent reductions such as live variable analysis and dead code elimination. Irrelevant variable abstraction, a form of property-directed slicing, does not make use of the modular system structure present in the IF intermediate representation. In contrast, slicing of SAS intermediate models as proposed in this work heavily exploits the internal structure of the models.

**Abstraction**

The foundations of abstraction go back to the abstract interpretation approach introduced by Cousot and Cousot in [CC77, CC79]. Abstract interpretation was originally used for static analysis of programs. Abstract interpretation operates on a concrete partial order $C$ and an abstract partial order $A$ of values. The partial orders $A$ and $C$ are related by a Galois connection which is a pair of mappings $\langle \alpha, \gamma \rangle$ where $\alpha$ maps the concrete partial order to the abstract and $\gamma$ the abstract partial order to the concrete. Correctness arguments in this approach exploit the properties of the Galois connection between the pair of abstraction mappings and the fact that a partial order induces a lattice. The data domain abstractions introduced in [LGS+95, DGG97] use $\langle \alpha, \gamma \rangle$ abstractions and the theory of abstract interpretation for justification of correctness.

A data domain abstraction using a homomorphic abstraction mapping from concrete to abstract values similar to the abstraction function $\alpha$ on SAS models is presented in [CGL94]. Using simulation arguments, it is shown that the universal fragment of $\text{CTL}^*$ is preserved. However, abstraction of expressions including operations and abstraction of predicates other than equality are not considered. In [DHJ+01], data abstraction for Java programs is proposed that also extends abstractions to operations and predicates. Java programs are syntactically abstracted using a mapping from concrete to abstract domains and abstract interpretations for operations and predicates. The correctness of the abstraction is based on safety conditions for abstract operations and predicates. This approach is very similar to data domain abstraction in SAS models. Kesten and Pnueli
[KP98] apply data abstraction to fair transition systems and LTL properties in a way that also liveness properties are preserved. But the abstraction is assumed to be given by a mapping from a concrete to an abstract structure and by a mapping from a concrete to an abstract property without providing means to constructively find the abstraction. Data type reduction [McM99] can be seen as a special case of data domain abstraction. Data type reduction keeps a finite number of values explicit while collapsing the other values of a data type to an unknown value. This is similar to range or set abstraction as presented in [DHJ+01]. In [SBLS99], a point abstraction of a program’s data to an unknown value is proposed. The approach relies on data-independence [Wol86] where the behaviour of a system to be analysed does not depend on concrete data, but only on control structures. In order to assist users in finding abstractions for efficient verification, Bandera [DHJ+01, HDPR02] and SMV [McM99] use pre-computed abstractions to syntactically abstract a concrete transition system into an abstract one. In the same direction, [GM99] proposes a syntactic abstraction of Promela programs, the input for the SPIN model checker.
Chapter 6

Compositional Reasoning

"Inside every well-written large program is a well-written small program."
(C.A.R. Hoare)

Besides reductions by model transformation presented in Chapter 5, compositional reasoning [Pnu85, GL94, FMS97, KV97, BCC97] is a second major attempt to overcome the state-explosion problem in model checking. Compositional reasoning is a divide-and-conquer approach that splits the overall global verification task into a number of local verification subtasks that are smaller and, therefore, easier to complete (cf. Figure 6.1). In this direction, the modular structure inherently present in a system is exploited in order to simplify reasoning.

In SAS verification, a global verification task over the whole model can be decomposed into verification tasks over single modules or subsystems since SAS models are built from a set of interconnected modules. If it

Fig. 6.1. Compositional Reasoning for Verification Complexity Reduction
is possible to complete the local verification tasks and the conjunction of the local results implies the global verification goal, the complete system satisfies the global property as well. This naive decomposition technique may, however, not always be feasible because of mutual dependencies between modules. In these cases, for verifying the behaviour of one system part, assumptions about the behaviour of the remaining system parts have to be made. The assumptions can later be discharged when the overall property is established. This strategy is called assume-guarantee reasoning [Pnu85]. Finding an assumption for the behaviour of the remaining system parts can be hard in some cases. Instead of employing an assumption, the remaining system parts can also be abstracted such that the observable behaviour at the interface remains the same [CLM89]. In this interface abstraction approach, a usually less complex abstraction of the remaining system parts is used for verification of the considered property reducing overall analysis effort. Furthermore, the separation of functional and adaptation behaviour in SAS models alleviates verification complexity. Orthogonal to adaptive slicing for verification of adaptive properties, functional behaviour can be verified without considering adaptation behaviour under certain conditions.

In this chapter, we first provide the formal basis for reasoning about compositions and decompositions of SAS models. Second, we present decomposition, assume-guarantee reasoning and interface abstraction as compositional reasoning strategies. Further, we develop compositional techniques focussing on the special features of adaptive systems. In embedded systems, the characteristics of the controlled system have to be considered for verification of properties of the controller that is developed. In order to deal with this problem, we propose an approach analogue to assume-guarantee reasoning expressing controlled system characteristics as environment assumptions. Finally, we show how compositional reasoning strategies can benefit from model transformations in order to obtain the best possible reduction of verification complexity.

6.1 Foundations for Compositional Reasoning

In order to develop compositional reasoning strategies for verification of SAS model properties, we have to provide basic notions about combinations of transformations and about composition and decomposition of SAS models. First, we show that consistent simulation relations between SAS
transition systems are transitive. This result is necessary for the correctness of assume-guarantee reasoning rules. For justifying that a combination of transformations is property-preserving, we show that the combination of simulation and bisimulation relations again constitute simulation or bisimulation relations. Second, we define projection and composition operators for SAS models. The projection operator provides a formal account of decompositions of SAS models. Its converse, the composition operator, is defined with respect to a larger super system defining the connections in the composed system. Based on projection and composition operators, results about the validity of model properties under composition of subsystems are derived. Third, for assume-guarantee reasoning we define the notion of maximal models consistent with a property. A model \( SAS \) for a property \( \varphi \) is a maximal model if all other models satisfying \( \varphi \) are simulated by the model \( SAS \). A maximal model can be used to encode assumptions in assume-guarantee reasoning.

### 6.1.1 Property-Preservation for Combined Transformations

If we perform a series of transformations, property preservation for each transformation step can be justified in isolation using a simulation or bisimulation argument. However, we can also show more generally that a combination of property-preserving transformations with respect to the same property is property-preserving. In [CGP99], simulation is shown to be a preorder on models. Further, it is argued that bisimulation induces an equivalence relation. Similar for SAS models, combining a transformation constituting a consistent simulation with another transformation also constituting a consistent simulation for the same property has the effect that the original and the final resulting system also form a consistent simulation. Combining two bisimulations yields a bisimulation. The combination of bisimulation and simulation results in a simulation.

The following theorem captures that the combination of two transformations, each forming a consistent simulation with respect to the same property, leads to a consistent simulation regarding the resulting system. Figure 6.2 illustrates the notions used in the theorem. The assumption that the same property is considered for the overall transformation leads to the restriction that the concretisation of the property from final to intermediate system is the same as the property on the intermediate system. In this theorem, it suffices to assume that the atomic propositions are in subset relation, which is implied by considering the same property. The proof pro-
ceeds by using both simulation relations existing by assumption in order to define a relation between the original and the final system. Then the newly defined relation is again shown to be a consistent simulation with respect to the final property. From this, it follows that properties of the universal fragment of $L_{SAS}$ are preserved under combinations of transformations constituting consistent simulations.

**Theorem 6.1 (Transitivity of Simulations).** Let $T = (\Sigma, \text{Init}, \sim)$ and $T' = (\Sigma', \text{Init}', \sim')$ be two transition systems such that both systems form a consistent simulation $T \preceq_\varphi T'$ with respect to an $AL_{SAS}$ property $\varphi$ and a concretisation function $C : AL_{SAS}[T'] \rightarrow AL_{SAS}[T]$. Further, let $T'' = (\Sigma'', \text{Init}'', \sim'')$ be a third SAS system such that $T' \preceq_\varphi T''$ are consistently similar for an $AL_{SAS}$ property $\varphi'$ over $T''$ and a concretisation function $C' : AL_{SAS}[T''] \rightarrow AL_{SAS}[T]$ such that $\text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi)$. Then it holds that $T$ and $T''$ are consistently similar with respect to $\varphi'$, i.e.

$$T \preceq_{\varphi'} T''$$

**Proof.** We know that if $T \preceq_{\varphi} T'$ and $T' \preceq_{\varphi} T''$ there must be two simulation relations $R \subseteq \Sigma \times \Sigma'$ and $R' \subseteq \Sigma' \times \Sigma''$. We define the relational composition $R'' \subseteq \Sigma \times \Sigma''$ by

$$R'' = \{(s, s'') \mid \exists s' \in \Sigma', (s, s') \in R \land (s', s'') \in R'\}$$

Furthermore, we define $C'' : AL_{SAS}[T''] \rightarrow AL_{SAS}[T]$ as $C''(a'') = C(C'(a''))$. Now, we have to show that $R''$ forms a consistent simulation between $T$ and $T''$ with respect to the property $\varphi'$ defined on $T''$ and the concretization function $C''$. 

---

**Fig. 6.2.** Composition of Simulations
6.1. Foundations for Compositional Reasoning 161

1. Initial: Let \( s_0 \) be an initial state of \( T \). As \( R \) is a simulation relation, there exists an initial state \( s_0' \) of \( T' \) such that \((s_0, s_0') \in R \) and, as \( R' \) is a simulation relation, there exists an initial state \( s_0'' \) of \( T'' \) such that \((s_0'', s_0') \in R' \). Thus, it holds that \((s_0, s_0'') \in R'' \).

2. Step Simulation: Let \( s_i \leadsto s_{i+1} \) be a transition in \( T \) and \((s_i, s_i') \in R'' \). Then there exists \( s_i' \in \Sigma' \) with \((s_i, s_i') \in R \). As \( R \) is a simulation relation, there exists \( s_i' \in \Sigma' \) such that \( s_i \leadsto s_{i+1} \) and \((s_{i+1}, s_{i+1}') \in R' \). Furthermore, as \( R' \) is a simulation relation, there exists \( s_{i+1}'' \in \Sigma'' \) such that \( s_i'' \leadsto s_{i+1}'' \) and \((s_{i+1}, s_{i+1}'') \in R'' \) which yields that \((s_{i+1}, s_{i+1}'') \in R'' \).

3. Consistency: For consistency, we have to show that for \((s, s'') \in R'' \) and \( a' \in \text{Atoms}(\varphi') \), it holds that \( s'' \models a' \iff s \models C''(a') \). By definition, there exists \( s' \in \Sigma' \) such that \((s, s') \in R \) and \((s', s'') \in R' \). As \( R' \) is consistent with respect to \( \varphi' \), we have that for \((s', s'') \in R' \) it holds that \( s'' \models a' \iff s' \models C'(a') \). As \( R \) is consistent with respect to \( \varphi \), it holds that for all \( b \in \text{Atoms}(\varphi) \) it holds that \( s' \models b \iff s \models C(b) \). As \( \text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi) \), it holds in particular that for all \( c \in \text{Atoms}(C'(\varphi')) \), that \( s' \models c \iff s \models C(c) \), which yields that \( s'' \models a' \iff s \models C(C'(a')) = C''(a') \). □

An analogue result can be proven for the composition of two bisimulations. A composition of two consistent bisimulations results in a consistent bisimulation between the original and the final system with respect to the same property. This yields the preservation of full \( \mathcal{L}_{SAS} \) under combination of two bisimilar transformations. For the following theorem, we have to assume that the atomic propositions of the concretisation of the property to the intermediate system are equal to the atomic propositions of the intermediate property in order to show the consistency condition. The proof proceeds analogue to the proof of the previous theorem by showing that the two given consistent bisimulation relations can be combined such that there is a consistent bisimulation relation between original and final system.

**Theorem 6.2 (Composition of Bimulations).** Let \( T = (\Sigma, \text{Init}, \leadsto) \) and \( T' = (\Sigma', \text{Init}', \leadsto') \) be two transition systems such that both systems form a consistent bisimulation \( T \cong_{[\varphi]} T' \) with respect to a \( \mathcal{L}_{SAS} \) property \( \varphi \) and a concretisation function \( C : \mathcal{L}_{SAS}[T'] \rightarrow \mathcal{L}_{SAS}[T] \). Further, let \( T'' = (\Sigma'', \text{Init}'', \leadsto'') \) be a third \( \text{SAS} \) system such that \( T' \cong_{[\varphi]} T'' \) are consistently bisimilar for an \( \mathcal{L}_{SAS} \) property \( \varphi' \) over \( T'' \) and a concretisation function \( C' : \mathcal{L}_{SAS}[T''] \rightarrow \mathcal{L}_{SAS}[T'] \) such that \( \text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi) \). Then it holds that \( T \) and \( T'' \) are consistently bisimilar with respect to \( \varphi' \), i.e.

\[
T \cong_{[\varphi']} T''
\]
Proof. Analogue to the proof of Theorem 6.1 with respect to consistent bisimulations: We know that if $T \cong[\varphi] T'$ and $T' \cong[\varphi'] T''$, there must be two bisimulation relations $R \subseteq \Sigma \times \Sigma'$ and $R' \subseteq \Sigma' \times \Sigma''$. We define the relational composition $R'' \subseteq \Sigma \times \Sigma''$ by

$$R'' = \{(s, s'') \mid \exists s' \in \Sigma', (s, s') \in R \land (s', s'') \in R'\}$$

Furthermore, we define $C'' : L_{SAS}[T''] \rightarrow L_{SAS}[T]$ as $C''(a'') = C(C'(a''))$. Now, we have to show that $R''$ forms a consistent bisimulation between $T$ and $T''$ with respect to the property $\varphi'$ defined on $T''$ and the concretization function $C''$.

1. Initial: Let $s_0$ be an initial state of $T$. As $R$ is a bisimulation relation, there exists an initial state $s_0'$ of $T'$ such that $(s_0, s_0') \in R$, and for each initial state $s_0$ of $T'$, there exists an initial state $s_0$ of $T$ such that $(s_0, s_0') \in R$. As $R'$ is a bisimulation relation, for each initial state $s_0'$ of $T''$ there exists an initial state of $s_0'$ of $T'$ such that $(s_0', s_0'') \in R'$ and vice versa. Thus, for each initial state $s_0$ of $T$, there is an initial state $s_0''$ of $T''$ such that $(s_0, s_0'') \in R''$ and vice versa.

2. Step Simulation 1: Let $s_i \rightsquigarrow s_{i+1}$ be a transition in $T$ and $(s_i, s_i'') \in R''$. Then there exists $s_i' \in \Sigma'$ with $(s_i, s_i') \in R$. As $R$ is a bisimulation, there exists $s_{i+1}' \in \Sigma'$ such that $s_i' \rightsquigarrow' s_{i+1}'$ and $(s_i, s_{i+1}') \in R$. Furthermore, as $R'$ is a bisimulation, there exists $s_{i+1}'' \in \Sigma''$ such that $s_i'' \rightsquigarrow'' s_{i+1}''$ and $(s_{i+1}', s_{i+1}'') \in R'$ which yields that $(s_{i+1}, s_{i+1}'') \in R''$.

3. Step Simulation 2: Let $s_i'' \rightsquigarrow'' s_{i+1}''$ be a transition in $T''$ and $(s_i, s_i'') \in R''$. Then there exists $s_i' \in \Sigma'$ with $(s_i', s_i'') \in R'$. As $R$ is a bisimulation, there exists $s_{i+1}' \in \Sigma'$ such that $s_i' \rightsquigarrow' s_{i+1}'$ and $(s_i', s_{i+1}') \in R'$. Furthermore, as $R$ is a bisimulation, there exists $s_{i+1} \in \Sigma$ such that $s_i \rightsquigarrow s_{i+1}$ and $(s_{i+1}, s_{i+1}) \in R$ which yields that $(s_{i+1}, s_{i+1}'') \in R''$.

4. Consistency: We have to show that for $(s, s'') \in R''$ it holds that $s'' \models a'$ iff $s \models C''(a')$ for $a' \in Atoms(\varphi')$. Let $s' \in \Sigma'$ such that $(s, s') \in R$ and $(s', s'') \in R'$. As $R'$ is consistent with respect to $\varphi'$, we have that for $(s', s'') \in R'$ it holds that $s'' \models a'$ iff $s \models C'(a')$. As $R$ is consistent with respect to $\varphi$, it holds that for all $b \in Atoms(\varphi)$ it holds that $s' \models b$ iff $s \models C(b)$. As $Atoms(C(\varphi')) \subseteq Atoms(\varphi)$, it holds in particular that for all $b \in Atoms(C'(\varphi'))$ that $s' \models b$ iff $s \models C(b)$ which is equivalent to $s \models C(C'(a')) = C''(a')$. 

As a consequence of the previous two theorems, we can show that a combination of two transformations where one is a consistent bisimulation and the other is a consistent simulation results in a consistent simulation with
6.1. Foundations for Compositional Reasoning

respect to the same property. This is an immediate consequence of Theorem 6.1 as a consistent bisimulation is also a consistent simulation in both directions. For the intermediate property, we have to require that the atomic propositions of the concretisation of the property to the intermediate system are a subset of the atomic propositions of the intermediate property as in Theorem 6.1.

Corollary 6.3 (Composition of Bimulation and Simulation). Let \( T = (\Sigma, \text{Init}, \sim) \) and \( T' = (\Sigma', \text{Init}', \sim') \) be two transition systems such that both systems form a consistent bisimulation \( T \cong_{[\varphi]} T' \) with respect to an \( \mathcal{AL}SAS \) property \( \varphi \) and a concretisation function \( C : \mathcal{AL}SAS[T'] \rightarrow \mathcal{AL}SAS[T] \). Further, let \( T'' = (\Sigma'', \text{Init}'', \sim'') \) be a third SAS system such that \( T' \preceq_{[\varphi']} T'' \) are consistently similar for an \( \mathcal{AL}SAS \) property \( \varphi'' \) over \( T' \) and a concretisation function \( C' : \mathcal{AL}SAS[T''] \rightarrow \mathcal{AL}SAS[T'] \) such that \( \text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi) \). Then it holds that \( T \) and \( T'' \) are consistently similar with respect to \( \varphi' \), i.e.

\[
T \preceq_{[\varphi']} T''
\]

Proof. As a bisimulation is also a simulation, the result follows immediately from Theorem 6.1.

Using the results on property preservation for consistent simulation and bisimulation as formulated in Theorems 5.8 and 5.10, we can now formulate which classes of properties are preserved under combinations of model transformations. The following theorems are applied in Section 6.3 in order to argue that a combination of different model transformations is also property-preserving and can be used for further verification complexity reduction.

The following theorem states that a combination of simulations preserves the universal fragment of \( \mathcal{L}_{SAS} \). This is for instance the case, when combining two different data domain abstractions. If the concretisation of the abstract property from the final system to the intermediate system is equal to the intermediate property, we can conclude that a combination of both abstractions preserves the abstract property on the final system.

Theorem 6.4 (Property-Preservation under Combination of Simulations). Let \( T = (\Sigma, \text{Init}, \sim) \) and \( T' = (\Sigma', \text{Init}', \sim') \) be two transition systems such that both systems form a consistent simulation \( T \preceq_{[\varphi]} T' \) with respect to a \( \mathcal{AL}SAS \) property \( \varphi \) and a concretisation function \( C : \mathcal{AL}SAS[T'] \rightarrow \mathcal{AL}SAS[T] \). Further, let \( T'' = (\Sigma'', \text{Init}'', \sim'') \) be a third
Chapter 6. Compositional Reasoning

SAS system such that \( T' \preceq_{[\varphi]} T'' \) are consistently similar with respect to an \( \mathcal{L}_{\text{SAS}} \) property \( \varphi' \) over \( T'' \) and a concretisation function \( C' : \mathcal{L}_{\text{SAS}}[T''] \rightarrow \mathcal{L}_{\text{SAS}}[T'] \) such that \( C'(\varphi') = \varphi \). Then, it holds that

\[
T'' \models \varphi' \quad \text{implies} \quad T \models C(C'(\varphi'))
\]

**Proof.** From \( C'(\varphi') = \varphi \), it follows that \( \text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi) \). Then, Theorem 6.1 is applicable such that it holds that \( T \preceq_{[\varphi]} T'' \) with respect to \( C''(\varphi') = C(C'(\varphi')) \). Now let \( T'' \models \varphi' \). From Theorem 5.7 instantiated for the initial states, we obtain \( T \models C(C'(\varphi')) \). \( \square \)

As the combination of two transformations constituting consistent bisimulations results in a consistent bisimulation (cf. Theorem 6.2), we can conclude that full \( \mathcal{L}_{\text{SAS}} \) is preserved. This result is necessary to derive that a combination of two different slicing algorithms such as adaptive slicing followed by system slicing preserves all properties expressible in \( \mathcal{L}_{\text{SAS}} \).

**Theorem 6.5 (Property-Preservation under Combination of Bisimulations).** Let \( T = (\Sigma, \text{Init}, \leadsto) \) and \( T' = (\Sigma', \text{Init}', \leadsto') \) be two transition systems such that both systems form a consistent bisimulation \( T \cong_{[\varphi]} T' \) with respect to an \( \mathcal{L}_{\text{SAS}} \) property \( \varphi \) and a concretisation function \( C : \mathcal{L}_{\text{SAS}}[T'] \rightarrow \mathcal{L}_{\text{SAS}}[T] \). Further, let \( T'' = (\Sigma'', \text{Init}'', \leadsto'') \) be a third SAS system such that \( T' \cong_{[\varphi']} T'' \) are consistently bisimilar for an \( \mathcal{L}_{\text{SAS}} \) property \( \varphi' \) over \( T'' \) and a concretisation function \( C' : \mathcal{L}_{\text{SAS}}[T''] \rightarrow \mathcal{L}_{\text{SAS}}[T'] \) such that \( \varphi = C'(\varphi') \). Then, it holds that

\[
T'' \models \varphi' \quad \text{iff} \quad T \models C(C'(\varphi'))
\]

**Proof.** From \( C'(\varphi') = \varphi \), it follows that \( \text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi) \). Then, Theorem 6.2 is applicable with yields \( T \cong_{[\varphi']} T'' \) with respect to \( C''(\varphi') = C(C'(\varphi')) \). Now let \( T'' \models \varphi' \). From Theorem 5.9 instantiated for the initial states, we obtain \( T \models C(C'(\varphi')) \). \( \square \)

In Corollary 6.3, we have stated that a combination of a consistent bisimulation and a consistent simulation leads to a consistent simulation with respect to the same property such that the universal fragment of \( \mathcal{L}_{\text{SAS}} \) is preserved. This result can be applied if a reduction by slicing is followed by an abstraction. Then the resulting system is similar to the original system which yields that universal properties are preserved. Note, that both properties must be from the universal fragment of \( \mathcal{L}_{\text{SAS}} \).
Theorem 6.6 (Property-Preservation under Combination of Bisimulations and Simulations). Let $T = (\Sigma, \text{Init}, \sim)$ and $T' = (\Sigma', \text{Init}', \sim')$ be two transition systems such that both systems form a consistent bisimulation $T \cong_{[\varphi]} T'$ with respect to an $\text{AL}_{\text{SAS}}$ property $\varphi$ and a concretisation function $C : \text{AL}_{\text{SAS}}[T'] \rightarrow \text{AL}_{\text{SAS}}[T]$. Further let, $T'' = (\Sigma'', \text{Init}'', \sim'')$ be a third SAS system such that $T' \preceq_{[\varphi']} T''$ are consistently similar for an $\text{AL}_{\text{SAS}}$ property $\varphi'$ over $T''$ and a concretisation function $C' : \text{AL}_{\text{SAS}}[T''] \rightarrow \text{AL}_{\text{SAS}}[T']$ such that $C'(\varphi') = \varphi$. Then, it holds that

\[ T'' \models \varphi' \implies T \models C(C'(\varphi')) \]

Proof. From $C'(\varphi') = \varphi$, it follows that $\text{Atoms}(C'(\varphi')) \subseteq \text{Atoms}(\varphi)$. Then, Corollary 6.3 is applicable with yields $T \preceq_{[\varphi]} T''$ with respect to $C''(\varphi') = C(C'(\varphi'))$. Now let $T'' \models \varphi'$. From Theorem 5.7 instantiated for the initial states, we obtain $T \models C(C'(\varphi'))$. □

Furthermore, we can transfer the results for property-preservation under combination of different transformations to the module level. These results are applicable in the context of decompositions presented in Section 6.2.1. Decomposing a system enables reasoning about single modules in isolation. For further reduction of verification complexity, these modules can be successively transformed before verifying the respective properties. As on system level, a combination of two bisimulations leads to a bisimulation of original and resulting system, a combination of two simulations gives a simulation, and a combination of a simulation and a bisimulation results in a simulation with respect to the same property. By structural equality of system and module transition systems, the proof of the following corollary is analogue to the proofs of the theorems on system level.

Corollary 6.7 (Combination of Module Transformations). Let $T_M = (S, \text{init}, \rightarrow)$ and $T_{M'} = (S', \text{init}', \rightarrow')$ and $T_{M''} = (S'', \text{init}'', \rightarrow'')$ be three module transition systems. Then it holds that

- If $T_M \preceq_{[\varphi]} T_{M'}$ are consistently similar with respect to an $\text{AL}_{\text{SAS}}$ property $\varphi$ and concretisation function $C : \text{AL}_{\text{SAS}}[T_{M'}] \rightarrow \text{AL}_{\text{SAS}}[T_M]$ and $T_{M'} \preceq_{[\varphi']} T_{M''}$ are consistently similar with respect to an $\text{AL}_{\text{SAS}}$ property $\varphi'$ and concretisation function $C' : \text{AL}_{\text{SAS}}[T_{M''}] \rightarrow \text{AL}_{\text{SAS}}[T_{M'}]$ then $C'(\varphi') = \varphi$ then

\[ T_{M''} \models \varphi' \implies T_M \models C(C'(\varphi')) \]
• If $T_M \cong_{[\varphi]} T_M'$ are consistently bisimilar with respect to an $L_{SAS}$ property $\varphi$ and concretisation function $C : L_{SAS}[T_M'] \rightarrow L_{SAS}[T_M]$ and $T_{M''} \cong_{[\varphi']} T_{M'''}$ are consistently bisimilar with respect to an $L_{SAS}$ property $\varphi'$ and concretisation function $C' : L_{SAS}[T_{M'''}] \rightarrow L_{SAS}[T_{M''}]$ where $C' (\varphi') = \varphi$ then

$$T_{M''} \models \varphi' \text{ iff } T_M \models C(C' (\varphi'))$$

• If $T_M \cong_{[\varphi]} T_M'$ are consistently bisimilar with respect to an $AL_{SAS}$ property $\varphi$ and concretisation function $C : AL_{SAS}[T_M'] \rightarrow AL_{SAS}[T_M]$ and $T_{M''} \cong_{[\varphi']} T_{M'''}$ are consistently similar with respect to an $AL_{SAS}$ property $\varphi'$ and concretisation function $C' : AL_{SAS}[T_{M'''}] \rightarrow AL_{SAS}[T_{M''}]$ where $C' (\varphi') = \varphi$ then

$$T_{M''} \models \varphi' \text{ implies } T_M \models C(C' (\varphi'))$$

Proof. As module transition systems are structurally equivalent to system transition systems, the proofs are analogue to the proofs of Theorems 6.4, 6.5 and 6.6.

6.1.2 Composition and Decomposition

First, compositional reasoning for SAS models requires to decompose a system into subsystems in order to consider parts of a system in isolation. Decomposition of an SAS is defined by means of a projection. A projection operator selects a set of SAS modules and produces a valid system containing

\[
\text{SAS}
\]

Fig. 6.3. Original SAS System
only the selected modules. We call the result of a projection an SAS subsystem. In the subsystem, the connections of the input system are projected onto the variables of the contained modules. This means that connections to modules no longer contained in the subsystem are removed. The same holds for system variables that are connected to removed variables. These are also deleted. If by projection module variables are unconnected because they were connected to removed modules, new system variables and the respective connections are inserted. In this way, a new system border is created around the subsystem. The system contains connections between modules already present in the original system and new system border variables and connections for the unconnected module variables. It is important that the resulting SAS is valid in order to be able to handle it by the same means as a global system before, for instance for model transformations. As example of a projection, consider the SAS system depicted in Figure 6.3. The projection onto the modules $M_1$ and $M_2$ and the projection onto the modules $M_3$ and $M_4$ are shown in Figure 6.4(a) and Figure 6.4(b), respectively. Note, that the newly created system variables are uniformly named with the module variable name and the suffix $sys$.

**Definition 6.8 (Projection).** Let $SAS$ be a system containing the modules $M = \{m_1, \ldots, m_n\}$. We define the system containing only the modules $M_I = \{m_i \mid i \in I\}$ with $I \subseteq \{1, \ldots, n\}$ as

$$\biguplus_{i \in I} M_i = \biguplus_{i \in I} m_i = (M_I, \text{inputs}_{a_I}, \text{inputs}_{d_I}, \text{output}_{a_I}, \text{output}_{d_I}, \text{conn}_{a_I}, \text{conn}_{d_I})$$

where
Second, for compositional reasoning subsystems have to be combined to create one system in order to lift local results over subsystems back to a global system. The system resulting from the composition of two subsystems contains the modules of the two subsystems and a new system border. The two old system borders are removed. The connections between module variables already contained in one of the two subsystems to be composed remain unchanged. For variables connected to the system border in one of the two subsystems inserting connections is more complex. This is because in SAS connected variables are not identified by matching names. Connections are not specified in a generic way, but have to be defined by the system designer. However, in order to be able to define composition of SAS subsystems in a general way, composition is defined with respect to a global SAS system. From this global system, connections between the subsystems to be composed are derived. The intuition is that variables in the two subsystems matching two connected variables in the global SAS will be connected. All variables that cannot be connected in this way are con-
6.1. Foundations for Compositional Reasoning

connected to the new system border. In order to define when a variable of a
global SAS matches a variable of one of the subsystems, we use projections
of the global SAS. If we can project the global system onto two subsystems
such that the interface variables, i.e. the system inputs and outputs, of the
subsystems obtained by the projection and the interface variables of the
subsystems to be composed are equal, equal variables are considered as
matching. If in the global SAS there is a connection between two variables
having matching variables in the subsystems, variables in the subsystems
are connected in the composition.

In order to capture this intuition formally, we introduce the notion of
compatibility between interfaces and the notion of composability with re-
spect to a global system. We call two systems $SAS_1$ and $SAS_2$ compatible, if
their interfaces are equal. This means that all input and output variables
at the system border have to have the same names. Figure 6.5 shows a sys-
tem that is compatible with $SAS_1$. It can be easily seen that the projection
onto the modules $M_1$ and $M_2$ and $SAS_3$ have the same interfaces. Note that
the variables inside the system and also the number of modules may differ
as long as the variables at the system border are equal.

**Definition 6.9 (Compatible Interfaces).** Let $SAS_1$ and $SAS_2$ be two sys-
tems with $SAS_1 = (M_1, inputs_{a1}, inputs_{d1}, output_{a1}, outputs_{d1}, conn_{a1}, conn_{a1})$
and $SAS_2 = (M_2, inputs_{a2}, inputs_{d2}, output_{a2}, outputs_{d2}, conn_{a2}, conn_{a2})$. $SAS_1$
and $SAS_2$ are compatible, denoted by $SAS_1 \sim SAS_2$, if

- $inputs_{a1} = inputs_{a2}$
- $inputs_{d1} = inputs_{d2}$
- $output_{a1} = outputs_{a2}$
- $output_{d1} = outputs_{d2}$

Using the definition of compatibility, we can define when two SAS sub-
systems are composable with respect to a global SAS. Two SAS subsystems $SAS_1$ and $SAS_2$ are composable with respect to a global SAS if there
are two subsystems $SAS_3$ and $SAS_4$ as projections from $SAS$ such that
the interfaces of the subsystems are compatible, i.e. $SAS_1 \sim SAS_3$ and
$SAS_2 \sim SAS_4$. Two projections from one SAS system are always compos-
able, i.e. if $SAS_1 = \bigcup_{i \in I} M_i$ for $I \subseteq \{1, \ldots, n\}$ and $SAS_2 = \bigcup_{j \in J} M_j$ and
$J \subseteq \{1, \ldots, n\}$, $SAS_1$ and $SAS_2$ are composable. In the definition of com-
posability, the projections do not have to be disjoint. In our example, $SAS_3$
and $SAS_2$ are composable with respect to $SAS$ because $SAS_3$ is compatible
with the projection $SAS_1$ and $SAS_2$ is a projection from $SAS$ itself. Note,
Chapter 6. Compositional Reasoning

**Fig. 6.5.** SAS System $SAS_3$ Compatible with $SAS_1$

the variable names of the systems to be composed must be disjoint in order to construct a valid system where all variables have to be disjoint.

**Definition 6.10 (Composability).** Let $SAS$ be an SAS system containing the modules $M = \{m_1, \ldots, m_n\}$. The SAS systems $SAS_1$ and $SAS_2$ are composable with respect to SAS, if there exist $M_J = \bigcup_{j \in J} m_j$ and $M_I = \bigcup_{i \in I} m_i$ such that $SAS_1 \sim M_J$ and $SAS_2 \sim M_I$ and $\text{Var}(SAS_1) \cap \text{Var}(SAS_2) = \emptyset$.

If we have two SAS systems $SAS_1$ and $SAS_2$ that are composable with respect to a global SAS, then the composition of those two subsystems is defined by removing the original system borders of the two SAS systems and adding a new system border for the composed system. The composed system contains the union of the modules of both subsystems. The connections inside the two subsystems to be composed remain the same. Only the variables connected to the system border in each system are connected according to the connections in the global SAS system. These connections are constructed as follows: Between each two variables of the subsystems connected to system variables, a connection is introduced if there exist connected matching system variables in the projections of the global system required for composability. As an example, consider $SAS_3$ in Figure 6.5. It is compatible with $SAS_1$ in Figure 6.4. In the original SAS depicted in Figure 6.3, there is a connection from $b$ to $h$. Variable $b_{sys}$ in $SAS_1$ and $SAS_3$ match. As $b_{sys}$ in $SAS_1$ is connected to $b$, $h_{sys}$ in $SAS_2$ is connected to $h$, and $b$ and $h$ are connected in $SAS_3$, there will be a connection between the variables that are connected to $b_{sys}$ and $h_{sys}$ in the composition. Thus, in the composition of $SAS_2$ with $SAS_3$, there will be a connection from $y$ in $SAS_3$ to $h$ in $SAS_2$ as $y$ is connected to $b_{sys}$ in $SAS_3$ and $h$ is connected to
6.1. Foundations for Compositional Reasoning

$h_{sys}$ in $SAS_2$. The next definition captures this notion of the connections introduced by a composition. Furthermore, it defines the notion of unconnected variables formally. In the following, let $Var_{mod}(SAS)$ denote the variables of the modules contained in the system $SAS$, i.e. if $SAS$ contains the set of modules $M$, we have $Var_{mod}(SAS) := Var(M)$.

**Definition 6.11 (Connections in Composition).** Let $SAS$ be an $SAS$ system and $SAS_1$ and $SAS_2$ two $SAS$ systems that are composable with respect to $SAS$. Further let $SAS_3$ and $SAS_4$ be the projections of $SAS$ such that $SAS_1 \sim SAS_3$ and $SAS_2 \sim SAS_4$.

- **The old functional connections are defined by**

  \[
  Conn_{old}(SAS_1, SAS_2, SAS) = (\text{conn}_{d1} \cap (Var_{mod}(SAS_1) \times Var_{mod}(SAS_1))) \cup (\text{conn}_{d2} \cap (Var_{mod}(SAS_2) \times Var_{mod}(SAS_2)))
  \]

- **The old adaptive connections are defined by**

  \[
  Conn_{old}(SAS_1, SAS_2, SAS) = (\text{conn}_{a1} \cap (Var_{mod}(SAS_1) \times Var_{mod}(SAS_1))) \cup (\text{conn}_{a2} \cap (Var_{mod}(SAS_2) \times Var_{mod}(SAS_2)))
  \]

- **The functional composition connections $Conn_{d||}(SAS_1, SAS_2, SAS)$ are defined as:** for all $v_1 \in output_{d1}$ such that there is $v_3 \in output_{d3}$ with $v_1 = v_3$, and for all $v_2 \in input_{d4}$ such that there is $v_4 \in input_{d4}$ with $v_2 = v_4$, and there is $x_3 \in Var_{mod}(SAS_3)$ and $x_4 \in Var_{mod}(SAS_4)$ with $\text{conn}_{d3}(x_3, v_3)$ and $\text{conn}_{d4}(v_4, x_4)$ in $SAS$, then $(v_1, v_2) \in Conn_{d||}$, and for all $v_1 \in input_{d1}$ such that there is $v_3 \in input_{d3}$ with $v_1 = v_3$, and for all $v_2 \in output_{d2}$ such that there is $v_4 \in output_{d4}$ with $v_2 = v_4$, and there is $x_3 \in Var_{mod}(SAS_3)$ and $x_4 \in Var_{mod}(SAS_4)$ with $\text{conn}_{d3}(v_3, x_3)$ and $\text{conn}_{d4}(x_4, v_4)$ in $SAS$, then $(v_2, v_1) \in Conn_{d||}$.

- **The adaptive composition connections $Conn_{a||}(SAS_1, SAS_2, SAS)$ are defined as:** for all $v_1 \in output_{a1}$ such that there is $v_3 \in output_{a3}$ with $v_1 = v_3$, and for all $v_2 \in input_{a4}$ such that there is $v_4 \in input_{a4}$ with $v_2 = v_4$, and there is $x_3 \in Var_{mod}(SAS_3)$ and $x_4 \in Var_{mod}(SAS_4)$ with $\text{conn}_{a3}(x_3, v_3)$ and $\text{conn}_{a4}(v_4, x_4)$ in $SAS$, then $(v_1, v_2) \in Conn_{a||}$, and for all $v_1 \in input_{a1}$ such that there is $v_3 \in input_{a3}$ with $v_1 = v_3$, and for all $v_2 \in output_{a2}$ such that there is $v_4 \in output_{a4}$ with $v_2 = v_4$, and there is $x_3 \in Var_{mod}(SAS_3)$ and $x_4 \in Var_{mod}(SAS_4)$ with $\text{conn}_{a3}(v_3, x_3)$ and $\text{conn}_{a4}(x_4, v_4)$ in $SAS$, then $(v_2, v_1) \in Conn_{a||}$.

Let $Conn_{a}(SAS_1, SAS_2, SAS) \subseteq Var_{mod}(SAS_1) \cup Var_{mod}(SAS_2)$ denote the unconnected adaptive module variables where $v \in Conn_{a}(SAS_1, SAS_2, SAS)$ iff
Chapter 6. Compositional Reasoning

Fig. 6.6. Composition of SAS$_1$ and SAS$_2$

there is no $w \in \text{Var}_\text{mod}(\text{SAS}_1) \cup \text{Var}_\text{mod}(\text{SAS}_2)$ such that $(v, w)$ or $(w, v) \in \text{Conn}_\text{add}(\text{SAS}_1, \text{SAS}_2, \text{SAS}) \cup \text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS})$.

Let $\text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS}) \subseteq \text{Var}_\text{mod}(\text{SAS}_1) \cup \text{Var}_\text{mod}(\text{SAS}_2)$ denote the unconnected functional module variables where $v \in \text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS})$ iff there is no $w \in \text{Var}_\text{mod}(\text{SAS}_1) \cup \text{Var}_\text{mod}(\text{SAS}_2)$ such that $(v, w)$ or $(w, v) \in \text{Conn}_\text{add}(\text{SAS}_1, \text{SAS}_2, \text{SAS}) \cup \text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS})$.

Using the definition of the connections induced by a composition, we can now formally define the composition of two systems SAS$_1$ and SAS$_2$ with respect to a global SAS. As an example, Figure 6.6 shows the composition of SAS$_3$ and SAS$_2$.

**Definition 6.12 (Composition).** Let SAS be an SAS system and SAS$_1$ and SAS$_2$ two systems that are composable with respect to SAS. The composition of SAS$_1$ and SAS$_2$ is defined as

$$\text{SAS}_1 \parallel_{\text{SAS}} \text{SAS}_2 = (M_\parallel, \text{in}_\parallel, \text{out}_\parallel, \text{conn}_\parallel)$$

where

- $M_\parallel = M_1 \cup M_2$ is union of modules of the systems SAS$_1$ and SAS$_2$
- $\text{in}_\parallel = \{v_{\text{sys}} \mid v \in \text{adapt}_\text{in}_1 \cup \text{adapt}_\text{in}_2 \land v \in \text{Conn}_\text{a}(\text{SAS}_1, \text{SAS}_2, \text{SAS})\}$
- $\text{in}_\parallel = \{v_{\text{sys}} \mid v \in \text{in}_1 \cup \text{in}_2 \land v \in \text{Conn}_\text{d}(\text{SAS}_1, \text{SAS}_2, \text{SAS})\}$
- $\text{out}_\parallel = \{v_{\text{sys}} \mid v \in \text{adapt}_\text{out}_1 \cup \text{adapt}_\text{out}_2 \land v \in \text{Conn}_\text{d}(\text{SAS}_1, \text{SAS}_2, \text{SAS})\}$
- $\text{out}_\parallel = \{v_{\text{sys}} \mid v \in \text{out}_1 \cup \text{out}_2 \land v \in \text{Conn}_\text{d}(\text{SAS}_1, \text{SAS}_2, \text{SAS})\}$
- $\text{conn}_\text{a} = \text{Conn}_\text{add}(\text{SAS}_1, \text{SAS}_2, \text{SAS}) \cup \text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS})$
  $\cup \{v_{\text{sys}} \mid v_{\text{sys}} \in \text{in}_\parallel \} \cup \{(v, v_{\text{sys}}) \mid v_{\text{sys}} \in \text{out}_\parallel \}$
- $\text{conn}_\text{d} = \text{Conn}_\text{add}(\text{SAS}_1, \text{SAS}_2, \text{SAS}) \cup \text{Conn}_\parallel(\text{SAS}_1, \text{SAS}_2, \text{SAS})$
  $\cup \{v_{\text{sys}} \mid v_{\text{sys}} \in \text{in}_\parallel \} \cup \{(v, v_{\text{sys}}) \mid v_{\text{sys}} \in \text{out}_\parallel \}$
There is a close relationship between projection and composition if two systems to be composed are direct disjoint projections from the original SAS. Composition of two projected subsystems is the projection onto the union of the modules of both subsystems. If both subsystems form a partition of the original system, their composition yields the original system.

**Lemma 6.13 (Connection between Projection and Composition).** Let $SAS$ be a system and $SAS_1 = \biguplus_{i \in I} M_i$ for $I \subseteq \{1, \ldots, n\}$ and $SAS_2 = \biguplus_{j \in J} M_j$ for $J \subseteq \{1, \ldots, n\}$ two subsystems. If $I \cap J = \emptyset$, it holds that $SAS_1 \parallel_{SAS} SAS_2 = \biguplus_{M_i \in SAS_1 \cup SAS_2} M_i$.

If $I \cup J = \{1, \ldots, n\}$, it holds that $SAS = SAS_1 \parallel SAS_2$.

**Proof.** Immediately by Definition 6.8 of the projection operator and Definition 6.12 of the composition operator. The proof relies on the fact that the projection necessary for the composability is equal to the respective systems $SAS_1$ and $SAS_2$. Then the definition of the newly introduced connections by composition collapses to the connections already present in $SAS$. By renaming the interface variables of $SAS$, it can easily be achieved that they are named $\text{v\_sys}$ where $\text{v}$ is the connected module variable in order to obtain $SAS = SAS_1 \parallel SAS_2$. □

As a next step we can consider how properties of systems are preserved under composition and decomposition using simulation arguments. However, as composition and decomposition change the variables at the borders of the respective systems, we restrict ourselves to properties where atomic propositions contain only module variables. Due to the definition of the SAS semantics, variables at the border of a system are always directly connected to module variables and contain the same values (cf. Lemma 3.17). So, this restriction is no restriction with respect to expressivity, but for ease of presentation.

Composition of two subsystems does not change the behaviour of the single subsystems. The states of the transition system induced by system composition is the union of the states of the two composed systems except for the newly generated or omitted variables at the system border. The possible transitions of the composed systems do not change either. Rather some transitions may no longer be enabled as the set of possible input values to the subsystems becomes smaller by composition. Hence, composition of a system $SAS$ with a composable system $SAS'$ can only restrict the possible
behaviours of the system $SAS$ with respect to variables of the contained modules $\text{Var}_{\text{mod}}(SAS)$. This means that there is a consistent simulation with respect to a property $\varphi$ that is expressible over the module variables of $SAS$, $v \in \text{Var}_{\text{mod}}(SAS)$, to the composition of $SAS$ and $SAS'$, $SAS \parallel SAS'$.

**Theorem 6.14 (Composition restricts Behaviour).** Let $SAS$ and $SAS'$ be two SAS systems that are composable with respect to a global system $SAS_g$ and let $\varphi$ be a property over the module variables of $SAS$, i.e. $\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(SAS)$. Then it holds that

$$T_{SAS \parallel SAS_g \parallel SAS'} \preceq_{[\varphi]} T_{SAS}$$

**Proof.** We have to define a simulation relation between the states $\sigma' \in \Sigma'$ of $T_{SAS \parallel SAS_g}$ and the states $\sigma \in \Sigma$ of $T_{SAS}$. Let the relation $\mathcal{R} \subseteq \Sigma' \times \Sigma$ be defined such that two states $\sigma$ and $\sigma'$ are in relation if they coincide on the values of the module variables of $SAS$, i.e. $(\sigma', \sigma) \in \mathcal{R}$ iff $\sigma'(v) = \sigma(v)$ for all $v \in \text{Var}_{\text{mod}}(SAS)$. As by composition the interpretation of the variables and values remains unchanged, we use the identity function as concretisation $C$.

1. **Initial Simulation:** Let $\sigma'_0 \in \Sigma'_0$ denote an initial state in $T_{SAS \parallel SAS'}$. Then by definition of the composition, $\sigma'_0|_{\text{Var}_{\text{mod}}(SAS)}$ is an initial state of the $SAS$. Hence, there exists a state $\sigma_0 \in \Sigma$ such that $(\sigma'_0, \sigma_0) \in \mathcal{R}$.

2. **Step Simulation:** Let $\sigma'_i \leadsto \sigma'_{i+1}$ be a transition in $T_{SAS \parallel SAS'}$ and $(\sigma'_i, \sigma_i) \in \mathcal{R}$. Then choose the input such that $\sigma_i \leadsto \sigma_{i+1}$ is a transition in $SAS$. By construction of the composition, this yields that $(\sigma'_{i+1}, \sigma_{i+1}) \in \mathcal{R}$.

3. **Consistency:** For all atomic propositions expressible over module variables of $SAS$, $a \in \text{Atoms}(\varphi)$ we have to show that for all $(\sigma', \sigma) \in \mathcal{R}$ it holds that $\sigma \models a$ iff $\sigma' \models a$. As $\sigma'(v) = \sigma(v)$ for all $v \in \text{Var}_{\text{mod}}(SAS)$, this equivalence holds. □

If two systems are in consistent simulation, the composition of both systems with a third system results again in a consistent simulation. The prerequisite for this composition is that both systems in simulation are composable with respect to the third system. The set of consistent atomic propositions is the set of atomic propositions expressible over the module variables of the third system together with the originally consistent atomic propositions over module variables of the two similar systems.
6.1. Foundations for Compositional Reasoning

**Theorem 6.15 (Composition preserves Simulation).** Let \( \text{SAS}, \text{SAS}' \) and \( \text{SAS}'' \) be three subsystems such that \( \text{SAS} \) and \( \text{SAS}' \) as well as \( \text{SAS} \) and \( \text{SAS}'' \) are composable with respect to a system \( \text{SAS}_g \). Let \( \varphi \) be a property over the module variables of \( \text{SAS}'' \), i.e. \( \text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(\text{SAS}'') \), and \( \psi \) be a property over the atomic propositions occurring in \( \varphi \) and all atomic propositions expressible in \( \mathcal{L}_{\text{SAS}} \) over module variables in \( \text{SAS} \). Then it holds that

\[
\mathcal{T}_{\text{SAS}'} \preceq_{[\varphi]} \mathcal{T}_{\text{SAS}''} \quad \text{implies} \quad \mathcal{T}_{\text{SAS}'''} \preceq_{[\psi]} \mathcal{T}_{\text{SAS}_g \text{SAS}''}
\]

**Proof.** We extend the simulation relation between \( \text{SAS}' \) and \( \text{SAS}'' \) to their composition with \( \text{SAS} \) and prove that this extended relation is a consistent simulation relation. Let \( \mathcal{R} \subseteq \Sigma' \times \Sigma'' \) be the consistent simulation relation between \( \text{SAS}' \) and \( \text{SAS}'' \). We define the relation \( \mathcal{R}' \subseteq \Sigma \parallel \Sigma' \times \Sigma \parallel \Sigma'' \) as follows:

\[
(\sigma, \sigma') \in \mathcal{R}' \quad \text{iff} \quad \forall v \in \text{Var}_{\text{mod}}(\text{SAS}). \sigma(v) = \sigma'(v) \wedge (\sigma|_{\text{Var}_{\text{mod}}(\text{SAS}')} : \sigma'|_{\text{Var}_{\text{mod}}(\text{SAS}'')}) \in \mathcal{R}
\]

As concretisation function we use the identity function on module variables of \( \text{SAS} \) and the concretisation function given by the assumption on variables of \( \text{Var}_{\text{mod}}(\text{SAS}''). \)

1. Initial Simulation: Let \( \sigma_0 \in \Sigma \parallel \Sigma' \). Then by definition of composition and the assumption that \( \mathcal{R} \) is a simulation, we know that for \( \sigma_0|_{\text{Var}_{\text{mod}}(\text{SAS}')} \) there is an initial state \( \sigma'_0 \) in \( \text{SAS}' \) and \( \sigma''_0 \) of the \( \text{SAS}'' \) such that \( (\sigma'_0, \sigma''_0) \in \mathcal{R} \). Thus, there exists a state \( \sigma''_0 \in \Sigma \parallel \Sigma'' \) such that \( (\sigma_0, \sigma''_0) \in \mathcal{R}' \).

2. Step Simulation: Let \( (\sigma_i, \sigma''_i) \in \mathcal{R}' \) and \( \sigma_i \leadsto \sigma_{i+1} \). By definition of composition and by the assumption that \( \mathcal{R} \) is a simulation, we know that for \( \sigma_i|_{\text{Var}_{\text{mod}}(\text{SAS}')} \) there exists \( \sigma'_i \) in \( \text{SAS}' \) with \( (\sigma'_i, \sigma''_i) \in \mathcal{R} \) and \( \sigma'_i \leadsto \sigma'_{i+1} \) such that there is \( \sigma''_{i+1} \) with \( \sigma'_i \leadsto \sigma''_{i+1} \) in \( \text{SAS}'' \). Hence, there is a transition in \( \text{SAS} \parallel \text{SAS}'' \) from \( \sigma''_i \) to \( \sigma''_{i+1} \) such that \( (\sigma_{i+1}, \sigma''_{i+1}) \in \mathcal{R}' \).

3. Consistency: Let \( a \in \text{Atoms}(\psi) \) and \( (\sigma, \sigma') \in \mathcal{R}' \). If \( \text{Var}(a) \subseteq \text{Var}_{\text{mod}}(\text{SAS}) \), we know that \( a = C(a) \). Hence, \( \sigma' \models a \iff \sigma \models a \) as \( \sigma(v) = \sigma'(v) \) for all \( v \in \text{Var}_{\text{mod}}(\text{SAS}) \). If \( a \in \text{Atoms}(\varphi) \) then we know that \( \sigma' \models a \iff \sigma \models C(a) \) as \( (\sigma|_{\text{Var}_{\text{mod}}(\text{SAS}')}, \sigma'|_{\text{Var}_{\text{mod}}(\text{SAS}'')}) \in \mathcal{R} \) and \( \mathcal{R} \) is a consistent simulation with respect to \( \varphi \) by assumption.

\[
\square
\]

6.1.3 Maximal Models

In order to develop compositional reasoning strategies for model checking, [GL94] introduce the notion of maximal models (or tableaus) for temporal
logic formulae. A maximal model for a property \( \varphi \) allows all behaviours that are consistent with the property \( \varphi \). This means that a maximal model is a model satisfying \( \varphi \) and is the largest model with respect to the simulation ordering. Simulation constitutes a preorder on models [CGP99] and captures the notion of ”having more behaviors”. A system simulating another system has more possible execution paths than the simulated system. A model of a property \( \varphi \) is a maximal model with respect to the behaviours consistent with \( \varphi \) if it, first, satisfies \( \varphi \) and, second, precedes all other models of this property with respect to the simulation ordering. The existence of maximal models (Kripke structures) for LTL and ACTL formulae is established by a tableau-based technique in [CGP99]. In [KV00], an automata-theoretic approach is presented for construction maximal models for ACTL* formulae.

Based on the notion of maximal models, satisfaction of a formula can also be expressed by simulation. If a model is simulated by the maximal model for a property \( \varphi \), the model also satisfies \( \varphi \), and vice versa. Maximal models can be used in assume-guarantee reasoning as model for the assumption (cf. Section 6.2.2) and for expressing properties of the controlled system in the embedded systems domain (cf. Section 6.2.5). The system or the considered system part is composed with the maximal model of the respective property. Then it is verified that the guarantee or the controller property holds in the composed system which allows inferring that the assumption implies the guarantee or that the controlled system property implies the controller property. The use of maximal models allows solving both tasks by standard model checking techniques.

In order to use maximal models for compositional verification of SAS systems, we define a maximal SAS model using consistent simulation as ordering. In this context, we fix the concretisation function required for a consistent simulation relation to the identity function. Consistent simulation constitutes a preorder on SAS models for universal properties in \( \mathcal{AL}_{SAS} \). It can easily be shown that a consistent simulation with respect to a property \( \varphi \) and the identity function as concretisation is reflexive. From Theorem 6.1, we further know that in this case consistent simulation is also transitive. A system \( M_\varphi \) is called a maximal model for a property \( \varphi \in \mathcal{AL}_{SAS} \) if it satisfies \( \varphi \) and, furthermore, all other SAS systems satisfying \( \varphi \) are in consistent simulation with \( SAS \).
6.1. Foundations for Compositional Reasoning

**Definition 6.16 (Maximal SAS Model).** \( M_\varphi \) is a maximal SAS model for \( \varphi \in A_L^{SAS} \) if \( T_{M_\varphi} \models \varphi \), and for every other SAS system \( SAS' \) with \( T_{SAS'} \models \varphi \), we have that \( T_{SAS'} \preceq_{[\varphi]} T_{M_\varphi} \), using the identity function as concretisation function \( C \).

The existence of maximal SAS models for \( A_L^{SAS} \) formulae can be justified along the lines of [KV00]. The Kripke structure constructed in [KV00] as a maximal model for ACTL* formulae can be encoded into a SAS model satisfying the conditions for a maximal SAS model in Definition 6.16. In Section 6.2.2, we show how to construct a maximal SAS model for the linear fragment of \( L_{SAS} \) by encoding a tableau for an LTL property into a SAS model. The LTL tableau is generated according to the algorithm provided in [CGP99].

A maximal model is not necessarily compatible with the simulated models. In order to enable composition with the maximal model, we define the notion of maximal compatible models. A maximal compatible model \( \tilde{M}_\varphi \) for a property \( \varphi \) is the smallest model satisfying \( \varphi \) compatible to all other models also satisfying \( \varphi \) and simulated by \( \tilde{M}_\varphi \). The maximal compatible model \( \tilde{M}_\varphi \) is also simulated by the maximal model \( M_\varphi \), i.e. \( T_{\tilde{M}_\varphi} \preceq_{[\varphi]} T_{M_\varphi} \).

**Definition 6.17 (Maximal Compatible Model).** \( \tilde{M}_\varphi \) is the maximal compatible model for \( \varphi \in A_L^{SAS} \) if \( T_{\tilde{M}_\varphi} \models \varphi \), and for every other SAS system \( SAS' \) with \( T_{SAS'} \models \varphi \), we have that \( T_{SAS'} \preceq_{[\varphi]} T_{\tilde{M}} \) using the identity function as concretisation function \( C \) and \( SAS' \sim \tilde{M}_\varphi \).

Furthermore, if we want to compose a maximal model with another system \( S \), we need the notion of a maximal composable model. The maximal composable model \( MC_\varphi \) is the largest model of the property \( \varphi \) composable with the system \( S \). The maximal composable model is simulated by the maximal model of \( \varphi \), i.e. \( T_{MC_\varphi} \preceq_{[\varphi]} T_{M_\varphi} \).

**Definition 6.18 (Maximal Composable Model).** \( MC_\varphi \) is the maximal composable model for \( \varphi \in A_L^{SAS} \) with respect to a subsystem \( SAS \) of a system \( S \) if \( T_{MC_\varphi} \models \varphi \) and SAS and \( MC_\varphi \) are composable, and for every other SAS system \( SAS' \) with \( T_{SAS'} \models \varphi \) and \( SAS' \) and SAS are composable, it holds that \( T_{SAS'} \preceq_{[\varphi]} T_{MC_\varphi} \) using the identity function as concretisation \( C \).

Between maximal compatible and maximal composable models, there exists a simulation relationship. Let \( SAS \) be a system consisting of two subsystems, i.e. \( SAS = SAS_1 \parallel SAS_2 \) where \( SAS_1 \) satisfies a formula \( \varphi \), \( T_{SAS_1} \models \varphi \). It can be shown that the maximal compatible model \( \tilde{M}_\varphi \) with
respect to $SAS_1$ and the maximal composable model $MC_\varphi$ with respect to $SAS_2$ are similar. The maximal composable model $MC_\varphi$ can be more general than the maximal compatible model $\tilde{M}_\varphi$. The reason is that a compatible model must have the same interface as $SAS_1$ whereas a composable model does not have to provide connections with respect to all system border variables of $SAS_2$. Hence, composition of $MC_\varphi$ with $SAS_2$ can leave variables of $SAS_2$ unconnected such that those are connected to the new system border in the composition. This is because the projection necessary for composability can be smaller than the system part $SAS_1$.

**Lemma 6.19 (Correspondence between Maximal Compatible and Maximal Composable Models).** Let $SAS = SAS_1 \parallel SAS_2$ and $\varphi \in \mathcal{AL}_{SAS}$. Let $MC_\varphi$ denote the maximal model for $\varphi$ composable with $SAS_2$ and $\tilde{M}_\varphi$ the maximal model for $\varphi$ compatible with $SAS_1$. Then, it holds that $T_{\tilde{M}_\varphi} \preceq [\varphi] T_{MC_\varphi}$ using the identity function as concretisation function $C$.

**Proof.** As $\tilde{M}_\varphi \sim SAS_1$, we have that $\tilde{M}_\varphi$ is composable with $SAS_2$. Additionally, $T_{\tilde{M}_\varphi} \models \varphi$. Hence by definition of maximal composable model, it holds that $T_{\tilde{M}_\varphi} \preceq [\varphi] T_{MC_\varphi}$ using the identity function as concretisation function $C$. □

Maximal models can be used as an alternative way to show that a system satisfies a property [GL94]. It is equivalent that an SAS model satisfies a property $\varphi$ and that there is a consistent simulation with respect to $\varphi$ between the model and the maximal model $M_\varphi$.

**Lemma 6.20 (Maximal Model Lemma).** Let $M_\varphi$ denote the maximal model for $\varphi \in \mathcal{AL}_{SAS}$ and $SAS$ a system. Then it holds that

$$T_{SAS} \preceq [\varphi] T_{M_\varphi} \iff T_{SAS} \models \varphi$$

using the identity function as concretisation function $C$.

**Proof.** The proof is based on the definition of the maximal model and property preservation by consistent simulation.

- **Left-to-right direction:** We assume that $T_{SAS} \preceq [\varphi] T_{M_\varphi}$. As $M_\varphi$ is the maximal model for $\varphi$, $T_{M_\varphi} \models \varphi$, and by property preservation by consistent simulation, it holds that $T_{SAS} \models \varphi$.
- **Right-to-left direction:** We assume that $T_{SAS} \models \varphi$. Then by definition of the maximal model $M_\varphi$, we get that $T_{SAS} \preceq [\varphi] T_{M_\varphi}$. □
The same result can be established for the maximal compatible model with restriction to maximal compatible models. The proof is analogue to the proof of the maximal model lemma above.

**Lemma 6.21 (Maximal Compatible Model Lemma).** Let $\bar{M}_\varphi$ denote the maximal compatible model for $\varphi \in \mathcal{AL}_{SAS}$ with respect to a system $SAS$. Then it holds for all compatible models $SAS'$ with respect to $SAS$ that

$$T_{SAS'} \preceq_{[\varphi]} T_{\bar{M}_\varphi} \iff T_{SAS'} \models \varphi$$

using the identity function as concretisation function $C$.

*Proof.* The proof is based on the definition of the maximal compatible model and property preservation by consistent simulation. It is analogue to the proof of Lemma 6.20. □

A similar result also holds for maximal composable models with the restriction to composable models. The proof is analogue to the proof of Lemma 6.20.

**Lemma 6.22 (Maximal Composable Model Lemma).** Let $MC_\varphi$ denote the maximal composable model for $\varphi \in \mathcal{AL}_{SAS}$ with respect to a system $SAS$. Then it holds for all systems $SAS'$ that are composable with $SAS$ that

$$T_{SAS'} \preceq_{[\varphi]} T_{MC_\varphi} \iff T_{SAS'} \models \varphi$$

using the identity function as concretisation function $C$.

*Proof.* The proof is based on the definition of the maximal composable model and property preservation by consistent simulation. It is analogue to the proof of Lemma 6.20. □

Maximal models can furthermore reduce the problem of implication checking. Let $\varphi$ be a universal $\mathcal{L}_{SAS}$ property and $M_\varphi$ the maximal model for $\varphi$. Then checking the implication $\varphi \rightarrow \psi$ for a $\mathcal{L}_{SAS}$ property $\psi$ is equivalent to showing that the maximal model of $\varphi$ is also a model for $\psi$, formally $T_{M_\varphi} \models \psi$.

### 6.2 Compositional Reasoning Strategies

Compositional reasoning is a divide-and-conquer approach splitting verification problems for a global model into smaller problems over parts of the
model. In that, the modular structure of the model is exploited for verification complexity reduction. In this section, we show how simple decompositions of models, assume-guarantee reasoning and interface abstraction can be applied as compositional reasoning strategies to reduce verification complexity for SAS models. Additionally, we provide a compositional reasoning strategy considering the special features of adaptive systems. It allows proving functional properties without considering adaptation behaviour. Furthermore, we present a technique how to deal with the characteristics of the controlled system when verifying properties over the controller in the embedded systems domain. This technique is analogue to assume-guarantee reasoning. Examples for the application of the compositional strategies presented in this section can be found in the experimental evaluation in Section 7.2.

6.2.1 Decomposition

Decomposition of verification tasks into a set of smaller subtasks reduces verification complexity significantly. Decomposition of SAS models splits the global system to be analysed into a set of subsystems. Additionally, the property to be verified over the global system is transformed into local properties over the subsystems. The decomposition process has to ensure that it suffices to establish the local properties over the subsystems in order to infer the global system property. In which way a model can be decomposed depends on the structure of the system and the considered property. If a property only refers to variables of a subsystem in the global system, it can suffice to check the validity of the property over this subsystem. If the global property can be expressed as a conjunction or a disjunction of properties over subsystems in the global system, it is sound to check each conjunct or disjunct over the affected subsystem and propagate the results to the validity of the global system. Additionally, a property referring only to one module can be established over the module in isolation.

In order to be able to infer a global property from local results, the considered property has to be a universal property imposing constraints over all paths of a system. By decomposition the considered subsystem is embedded into an environment that can exhibit any possible behaviour. If a universal property of a subsystem in this chaotic environment is true, it also holds in the global system. The reason is that the subsystem as a part of the original system can only have less behaviours than in the chaotic environment. For existential properties, there can be a path in the chaotic
environment such that the property is true. However, in the original system, this path may be impossible. Thus, validity of existential properties cannot be established by decomposition. Furthermore, a property may fail to hold in the decomposition while it is in fact true in the original system because the chaotic environment may exhibit behaviours that are not possible in the original system and invalidate the property. Hence, if we can verify a property over a subsystem in isolation, we can transfer the result to the global system. However, if verification on the decomposed system fails, we cannot say anything about the validity in the global system.

The following theorem serves as a basis for our decomposition rules over SAS models. It is a consequence of Theorem 6.14 that a system composed with a second system is always in a consistent simulation with itself with respect to all expressible properties over the subsystem. This yields that a universal property over a subsystem \( \text{SAS} \) remains true in any system containing \( \text{SAS} \). Thus, if \( \mathcal{T}_{\text{SAS}} \models \varphi \) where \( \varphi \) is build from atomic propositions over \( \text{SAS} \), also \( \mathcal{T}_{\text{SAS} \parallel \text{SAS}'} \models \varphi \) holds for composable systems \( \text{SAS}' \).

For ease of presentation, we restrict the theorem to properties without system border variables and use Lemma 3.17 for transforming a property containing system border variables to an equivalent property without border variables.

**Theorem 6.23 (Hierarchical Validity).** For two SAS systems \( \text{SAS} \) and \( \text{SAS}' \) that are composable and \( \varphi \) an \( \text{AL}_{\text{SAS}} \) property over \( \text{Var}_{\text{mod}}(\text{SAS}) \), it holds that

\[
\mathcal{T}_{\text{SAS}} \models \varphi \quad \text{implies} \quad \mathcal{T}_{\text{SAS} \parallel \text{SAS}'} \models \varphi
\]

**Proof.** Immediately from the consistent simulation between \( \text{SAS} \parallel \text{SAS}' \) and \( \text{SAS} \) with respect to properties expressible over \( \text{Var}_{\text{mod}}(\text{SAS}) \) in Theorem 6.14 and the property preservation result by consistent simulation in Theorem 5.8. \( \square \)

**Decomposition into Subsystems**

Using the previous theorem, we can define our first decomposition rule that allows projecting a global system onto the subsystem a universal property to be verified refers to. If we are able to verify the property over the subsystem, the result can be transferred to the global system. Note, that failure to verify the property over the decomposed subsystem does not imply that the property is invalid over the global system. The proof of the first decomposition rule uses Theorem 6.23 instantiated with a global system and a subsystem constructed by projection.
Theorem 6.24 (Decomposition for Subsystem). Let $SAS$ be an SAS system with the set of modules $M = \{m_1, \ldots, m_n\}$. Let $\varphi_i$ be an AL$_{SAS}$ property with $\text{Var}(\varphi) \subseteq \bigcup_{i \in I} \text{Var}(m_i)$ for each $I \subseteq \{1, \ldots, n\}$. Then,

$$T_{\bigcup_{i \in I} m_i} \models \varphi \text{ implies } T_{SAS} \models \varphi$$

Proof. We use Theorem 6.23, where $SAS = SAS' \parallel SAS''$ with $SAS' = \bigcup_{i \in I} m_i$ and $SAS'' = \bigcup_{j \in \{1, \ldots, n\} \setminus I} m_j$. As $T_{SAS'} \models \varphi$ by assumption, we obtain the result that $T_{SAS} \models \varphi$. □

The second and third decomposition rule exploit the structure of the property to be verified in addition to the system structure. If a global property to be verified can be expressed as a conjunction or as a disjunction of universal properties referring only to subsystems of the global system, it suffices to show the conjuncts or disjuncts over the subsystems. The second decomposition rule in the following theorem states that for a conjunctive global property, all conjuncts must be true over the respective subsystems such that the global property is valid. The proof goes by instantiating the hierarchical validity result in Theorem 6.23 for each conjunct and applying Boolean logic to complete the proof.

Corollary 6.25 (Decomposition over Conjunction). Let $SAS$ be an SAS system with the set of modules $M = \{m_1, \ldots, m_n\}$. Let $\varphi_i$ be an AL$_{SAS}$ property with $\text{Var}(\varphi_i) \subseteq \text{Var}(\bigcup_{i \in I} m_i)$ for $I \subseteq \{1, \ldots, n\}$. Then,

$$\forall i \in \{1, \ldots, n\}. T_{\bigcup_{i \in I} m_i} \models \varphi_i \text{ implies } T_{SAS} \models \bigwedge_{i=1}^{n} \varphi_i$$

Proof. Immediately from Theorem 6.24 and the semantics of Boolean conjunction. □

The third decomposition rule considers a global property expressible as a disjunction of universal properties over subsystems of the global system. Here, it suffices to show the validity of one disjunct over the respective subsystem in order to establish validity of the global property. The proof again goes by instantiating the hierarchical validity result and using Boolean logic.
Corollary 6.26 (Decomposition over Disjunction). Let $SAS$ be an SAS system with the set of modules $M = \{m_1, \ldots, m_n\}$. Let $\varphi_i$ be an $\mathcal{AL}_{\text{SAS}}$ property with $\text{Var}(\varphi_i) \subseteq \text{Var}(\bigcup_{i \in I} m_i)$ for $I \subseteq \{1, \ldots, n\}$. Then,

$$\exists i \in \{1, \ldots, n\}. \mathcal{T}_{\bigcup_{i \in I} m_i} \models \varphi_i \text{ implies } \mathcal{T}_{SAS} \models \bigvee_{i=1,\ldots,n} \varphi_i$$

Proof. Immediately from Theorem 6.24 and the semantics of Boolean disjunction.}

The decomposition rules can also be extended to formulae where conjunctions and disjunctions of universal properties are intertwined. As an example, consider a disjunction of conjunctions of universal formulae over a global system $SAS$, e.g. $\varphi = (\varphi_1 \land \varphi_2) \lor (\varphi_3 \land \varphi_4)$. If the two disjuncts are only concerned with subsystems of $SAS$, we can first apply the disjunctive decomposition rule. Then we have to show that the first subsystem satisfies the first disjunct $\mathcal{T}_{SAS_1} \models \varphi_1 \land \varphi_2$ and that the second subsystem satisfies the second $\mathcal{T}_{SAS_2} \models \varphi_3 \land \varphi_4$. For each of the subsystems, we can apply the conjunctive decomposition rule in case $\varphi_i$ for $i \in \{1, \ldots, 4\}$ only affects a subset of the modules of the respective subsystem. In this way, the decomposition rules help reducing the verification problem sizes significantly before applying other reduction techniques to the remaining proof goals.

**Decomposition into Modules**

The previous decomposition rules considered the projection of an SAS system to subsystems. A more rigourous decomposition technique is to focus on modules contained in a system in isolation without constructing projections of the system. The difference between projecting an SAS system onto a module and considering the module in isolation is the system border introduced by the projection. The following lemma captures the correspondence that a property over the projection of a system onto a module is true if and only if the property over the module in isolation is true. In the property, variables at the system border of the projected module are replaced by the connected module variables. It is a special case of Lemma 3.17 which considers the equivalence between properties containing system border variables and properties containing only module variables. The lemma allows us to transfer the decomposition techniques previously established for subsystems to modules in isolation.
Lemma 6.27 (Module Decomposition, Projection and Validity). Let $SAS$ be a system containing the modules $M = \{m_1, \ldots, m_n\}$ and $\varphi$ an $\mathcal{AL}_{SAS}$ property with $\text{Var}(\varphi) \subseteq \text{Var}(\bigcup m_i)$ for $i \in \{1, \ldots, n\}$. Further, let $\varphi'$ be the property derived from $\varphi$ by replacing all occurrences of a variable $v \in \text{Var}(\bigcup m_i) \setminus \text{Var}(m_i)$ by the variable $v'$ if either $(v, v') \in \text{conn}_{a \uplus m_i} \cup \text{conn}_{d \uplus m_i}$ or $(v', v) \in \text{conn}_{a \uplus m_i} \cup \text{conn}_{d \uplus m_i}$. Then it holds that $T_{\bigcup m_i} \models \varphi$ iff $T_{m_i} \models \varphi'$.

Proof. The proof is by induction on the structure of $\varphi$.

- Induction Base: $\varphi = a$ where $a$ is an atomic proposition. Then, we have two cases. First, $\text{Var}(a) \subseteq \text{Var}(m_i)$. It holds that $T_{\bigcup m_i} \models a$ iff $T_{m_i} \models a$ as the semantics of the internal module is the same in both systems by definition of local and global SAS semantics.
  Second, $\text{Var}(a) \subseteq \text{Var}(\bigcup m_i) \setminus \text{Var}(m_i)$. Let $T_{\bigcup m_i} \models a$. Then we replace each $v \in \text{Var}(a) \setminus \text{Var}(m_i)$ in $a$ by $v' \in \text{Var}(m_i)$ for $(v, v') \in \text{conn}_{a \uplus m_i} \cup \text{conn}_{d \uplus m_i}$ or $(v', v) \in \text{conn}_{a \uplus m_i} \cup \text{conn}_{d \uplus m_i}$ obtaining $a'$. By definition of the global SAS semantics, we have that in each state $s$ of $\bigcup m_i$, $s(v) = s(v')$. Thus, $T_{m_i} \models a'$ as the semantics of $m_i$ inside $\bigcup m_i$ and in isolation are the same. The reverse holds by the same argument.
- Induction Step: The equivalence for Boolean and temporal connectives follows immediately from the induction hypothesis. □

Using the above lemma, we can formulate the first decomposition rule from Theorem 6.24 for decomposition of a system in terms of the contained modules. As the properties do not reference system border variables by assumption, a transformation of the property is not necessary.

Theorem 6.28 (Property Preservation of $\mathcal{AL}_{SAS}$ under Module Decomposition). Let $SAS$ be an SAS system with the set of modules $M = \{m_1, \ldots, m_n\}$. Let $\varphi_i$ be an $\mathcal{AL}_{SAS}$ property with $\text{Var}(\varphi_i) \subseteq \text{Var}(m_i)$. Then,

$$T_{m_i} \models \varphi_i \text{ implies } T_{SAS} \models \varphi_i$$

Proof. Immediately from Theorem 6.24 and the previous Lemma 6.27 □

Theorems 6.25 and 6.26 can also be formulated over single modules using Theorem 6.28. This allows verifying properties that are expressed as conjunctions or disjunctions of universal properties over single modules by considering the modules in isolation.
Corollary 6.29 (Module Decomposition over Conjunction and Disjunction). Let \( \text{SAS} \) be an \( \text{SAS} \) system with the set of modules \( M = \{m_1, \ldots, m_n\} \). Let \( \varphi_i \) be an \( \mathcal{AL}_{\text{SAS}} \) property with \( \text{Var}(\varphi_i) \subseteq \text{Var}(m_i) \) for each \( i \in \{1, \ldots, n\} \). Then it holds that, for a conjunction

\[
\forall i \in \{1, \ldots, n\}. T_{m_i} \models \varphi_i \text{ implies } T_{\text{SAS}} \models \bigwedge_{i=1}^{i=n} \varphi_i
\]

and for a disjunction

\[
\exists i \in \{1, \ldots, n\}. T_{m_i} \models \varphi_i \text{ implies } T_{\text{SAS}} \models \bigvee_{i=1}^{i=n} \varphi_i
\]

Proof. Analogue to the proof of Theorem 6.28 from Theorem 6.24 and Lemma 6.27 \( \square \)

6.2.2 Assume-Guarantee Reasoning

Verification by decomposition embeds a subsystem in a chaotic environment generating all possible inputs for the subsystem. Thus, it may happen that the property is not valid over the subsystem within the chaotic environment. But, if we restrict the behaviour of the environment to satisfy a given assumption, we may be able to establish the validity of the property. A subsystem is embedded into an environment satisfying a certain assumption. The environment assumption allows verifying the considered property over the subsystem. If the remaining system actually satisfies the environment assumption, the assumption can be discharged and the property can be established over the global system. This assume-guarantee paradigm was first advocated by Pnueli [Pnu85] for temporal logic using a notation of the form \( \langle \varphi \rangle M \langle \psi \rangle \). The reading is that if the environment of a system \( M \) satisfies the assumption \( \varphi \), then \( M \) satisfies the guarantee \( \psi \). The advantage of assume-guarantee reasoning is that the state space of the global system does not have to be considered, but only the state spaces of both system parts in separation. This counters the state-explosion problem for model checking while allowing verification of more properties than by simple decomposition. The disadvantage is however that an assumption strong enough to establish the guarantee has to be found.

Assume-Guarantee Reasoning for \( \text{SAS} \)

Figure 6.7 depicts assume-guarantee reasoning in a system \( \text{SAS} \). It proceeds in the following steps that we prove sound in this section:
• Decompose the global system into two disjoint parts $SAS = SAS_1 \parallel SAS_2$

• Give an assumption $\varphi$ for the first part $SAS_1$ and show that it holds for the first subsystem, i.e. $T_{SAS_1} \models \varphi$.

• Verify that the second part $SAS_2$ satisfies the guarantee $\psi$ under the assumption $\varphi$, i.e. $\langle \varphi \rangle_{SAS_2} \langle \psi \rangle$

• From the assume-guarantee reasoning rules conclude that $T_{SAS} \models \psi$

Assume-guarantee pairs for SAS models $\langle \varphi \rangle_{SAS_2} \langle \psi \rangle$ denoting that the subsystem $SAS_2$ satisfies the guarantee $\psi$ under the assumption $\varphi$ are defined in terms of SAS models as follows.

**Definition 6.30 (Assume-Guarantee Pairs).** In an SAS system $SAS$, for a subsystem $M$ and all composable subsystems $M'$ with respect to $M$ and two $\mathcal{AL}_{SAS}$ formulae $\varphi$ and $\psi$ with $\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(M')$ and $\text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(M)$, the assume-guarantee reasoning pair $\langle \varphi \rangle_{SAS_2} \langle \psi \rangle$ holds iff it holds that $T_{M\parallel M'} \models \psi$ for a subsystem $M'$ that satisfies $\varphi$, $T_{M'} \models \varphi$.

In our framework with maximal, maximal composable and maximal compatible models introduced in Section 6.1.3, we can define the meaning of an assume-guarantee pair as follows: If the subsystem $M$ is composed with the maximal composable model $MC_{\varphi}$ for the assumption $\varphi$ and it can be verified that the composition satisfies the guarantee property $\psi$, then the assume-guarantee pair $\langle \varphi \rangle_{M} \langle \psi \rangle$ is valid. This approach is adapted from [GL94] where assume-guarantee reasoning is reduced to standard model-

![Fig. 6.7. Assume-Guarantee Reasoning](image-url)
6.2. Compositional Reasoning Strategies

checking by maximal model construction for Kripke structures and Moore automata.

**Theorem 6.31 (Assume-Guarantee Reasoning with Maximal Models).** In an SAS system $SAS$, for a subsystem $M$ and two formulae $\varphi$ and $\psi$ with $\text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(M)$ from $\mathcal{AL}_{SAS}$ and $MC_\varphi$ the maximal model of $\varphi$ composable with $M$ with respect to $SAS$ it holds that

$$T_{MC_\varphi|\bar{M}} = \psi \iff \langle \varphi \rangle M \langle \psi \rangle$$

**Proof.**

- **Left-to-right-direction:** Assume that $T_{MC_\varphi|\bar{M}} = \psi$. As $MC_\varphi$ is the maximal model for $\varphi$ for all models $M'$ that are composable with $M$ and $T_{M'} \models \varphi$, we have that $T_{M'} \preceq_{[\varphi]} T_{MC_\varphi}$ using the identity function as concretisation function $C$. By Theorem 6.15, we have that $T_{M'|\bar{M}} \preceq_{[\varphi]} T_{MC_\varphi|\bar{M}}$ using the identity function as concretisation function $C$. By transitivity of $\preceq_{[\varphi]}$, we can conclude that $T_{M'|\bar{M}} \models \psi$. Hence, $\langle \varphi \rangle M \langle \psi \rangle$ holds.

- **Right-to-left-direction:** Assume $\langle \varphi \rangle M \langle \psi \rangle$. By definition, for all $M'$ with $T_{M'} \models \varphi$ that are composable with $M$ it holds that $T_{M'|\bar{M}} \models \psi$. As $MC_\varphi$ is the maximal model composable with $M$, in particular $T_{MC_\varphi} \models \varphi$, and hence $T_{MC_\varphi|\bar{M}} \models \psi$. □

In the following theorem, we formulate the main assume-guarantee reasoning result for SAS models. If a system can be partitioned into two subsystems such that one satisfies an assumption $\varphi$ and the other satisfies the guarantee $\psi$ under the assumption $\varphi$, the complete system satisfies the guarantee $\psi$. Note, that assumption and guarantee have to be universal properties only referring to module variables in both subsystems. The rule can be extended to system border variables using Lemma 3.17. The proof of the assume-guarantee reasoning rule uses the properties of maximal composable models, transitivity of simulation relations and the fact that composition of systems preserves simulation relations.

**Theorem 6.32 (Assume-Guarantee Reasoning Rule).** Let $SAS$ be a system $SAS = SAS_1 \parallel SAS_2$ and $\varphi$ and $\psi$ two $\mathcal{AL}_{SAS}$ properties with $\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(SAS_1)$ and $\text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(SAS_2)$. Then the following assume-guarantee rule is sound:

$$T_{SAS_1} \models \varphi$$

$$\langle \varphi \rangle SAS_2 \langle \psi \rangle$$

$$T_{SAS} \models \psi$$
Proof. (1) By Maximal Compatible Model Lemma 6.22, we can restate the first hypothesis as $T_{SAS_1} \preceq_{\{\varphi\}} T_{\tilde{M}_\varphi}$ using the identity function as concretisation function $C$ since $SAS_1$ is in particular compatible with itself.

(2) For any formula $\psi'$ expressible in $L_{SAS}$ over $\text{Var}^{\text{mod}}(SAS_2)$, it holds by Theorem 6.15 that $T_{SAS_1\|SAS_2} \preceq_{[\varphi \land \psi']} T_{\tilde{M}_\varphi}\|SAS_2}$ using the identity function as concretisation function $C$ as $SAS_1$ and $SAS_2$ are composable by construction and $\tilde{M}_\varphi$ and $SAS_2$ are composable because $\tilde{M}_\varphi$ and $SAS_1$ are compatible. Disjointness of variables can be obtained by renaming, if necessary.

(3) By Theorem 6.31, we can rewrite the second hypothesis as $T_{MC_\varphi}\|SAS_2} \models \psi$ where $MC_\varphi$ is the maximal composable model with respect to $SAS_2$. By Maximal Composable Model Lemma 6.22, we further obtain $T_{MC_\varphi}\|SAS_2} \preceq_{[\psi]} T_{\tilde{M}_\varphi}$ using the identity function as concretisation function $C$.

(4) Since $SAS = SAS_1 \| SAS_2$, by Lemma 6.19, we know that $\tilde{M}_\varphi$, the maximal compatible model for $\varphi$ with respect to $SAS_1$ and the maximal composable model $MC_\varphi$ are similar, i.e. $T_{\tilde{M}_\varphi} \preceq_{[\varphi]} T_{MC_\varphi}$ using the identity function as concretisation function $C$. Then, by Theorem 6.15 we have that $T_{\tilde{M}_\varphi}\|SAS_2} \preceq_{[\varphi \land \psi']} T_{MC_\varphi}\|SAS_2} using the identity function as concretisation function $C$.

(5) Then, by (3), (4) and transitivity of $\preceq_{[\varphi \land \psi']} \preceq_{[\psi]} T_{\tilde{M}_\varphi}$ using the identity function as concretisation function $C$ as $AP(\psi) \subseteq AP(\varphi \land \psi')$ by assumption that $\text{Var}(\psi) \subseteq \text{Var}^{\text{mod}}(SAS_2)$.

(6) By (2) and (5) and transitivity of $\preceq_{[\varphi \land \psi']} \preceq_{[\psi]} T_{\tilde{M}_\varphi}$ using the identity function as concretisation function $C$.

(7) By Maximal Model Lemma 6.20, we obtain $T_{SAS_1\|SAS_2} \models \psi$ which implies $T_{SAS} \models \psi$. □

Further assume-guarantee reasoning rules can be justified in the same way. As an example, we establish an assume-guarantee reasoning rule for a conjunctive property where each conjunct only affects one part of the system. A similar result could be proven for disjunctive properties. For the conjunctive assume-guarantee reasoning rule, we assume that one part of the system $SAS_1$ satisfies an assumption $\varphi_1$ while the other part $SAS_2$ satisfies another assumption $\varphi_2$. Now, we have two assume-guarantee pairs, first that $SAS_1$ satisfies a guarantee $\psi_1$ under the assumption $\varphi_2$ and, second, that $SAS_2$ satisfies another guarantee $\psi_2$ under the assumption $\varphi_1$. This allows us to conclude that the combined system satisfies the conjunction of the two guarantees.
Theorem 6.33 (Conjunctive AGR-Rule). Let $SAS = SAS_1 \parallel SAS_2$ and $\varphi_1, \varphi_2, \psi_1, \psi_2$ be $\mathcal{AL}_{SAS}$ properties with $\text{Var}(\varphi_1), \text{Var}(\psi_1) \subseteq \text{Var}_{\text{mod}}(SAS_1)$ and $\text{Var}(\varphi_2), \text{Var}(\psi_2) \subseteq \text{Var}_{\text{mod}}(SAS_2)$. Then it holds that

$$T_{SAS_1} \models \varphi_1, \quad T_{SAS_2} \models \varphi_2, \quad \langle \varphi_2 \rangle_{SAS_1} \langle \psi_1 \rangle, \quad \langle \varphi_1 \rangle_{SAS_2} \langle \psi_2 \rangle$$

$$T_{SAS} \models \psi_1 \land \psi_2$$

Proof. For completing the proof, we apply Theorem 6.32 twice.

1. From $T_{SAS_1} \models \varphi_1$ and $\langle \varphi_1 \rangle_{SAS_2} \langle \psi_2 \rangle$, we obtain $T_{SAS_1 \parallel SAS_2} \models \psi_2$.
2. From $T_{SAS_2} \models \varphi_2$ and $\langle \varphi_2 \rangle_{SAS_1} \langle \psi_1 \rangle$, we obtain $T_{SAS_1 \parallel SAS_2} \models \psi_1$.
3. Using Boolean logic, we get the result $T_{SAS_1 \parallel SAS_2} \models \psi_1 \land \psi_2$.

The authors of [CAC06] identify two main problems of the assume-guarantee reasoning strategy for compositional verification. First, it has to be determined how to partition a system into two subsystems such that one satisfies an assumption while the other satisfies the guarantee. Second, a suitable assumption for verification of the guarantee has to be discovered. Both aspects significantly affect the efficiency of the assume-guarantee reasoning strategy. In this direction, approaches to automatically generate assumptions necessary for assume-guarantee reasoning [Ang87, JGP03] can help to overcome these problems.

Assume-Guarantee Reasoning and Model Checking

In order to use the assume-guarantee reasoning strategy with standard model checking, checking the validity of assume-guarantee pairs $\langle \varphi \rangle_{SAS} \langle \psi \rangle$ has to be transformed into a model checking problem. In the literature, there are three different approaches to implement assume-guarantee reasoning within model checking: linear, linear-branching and branching modular model checking [KV97]. In linear modular model-checking, assumption and guarantee are specified in linear time temporal logic, in linear-branching modular model checking the assumption is specified in linear time temporal logic whereas the guarantee is from the universal fragment of branching time temporal logic, in branching modular model checking assumption and guarantee are from the universal fragment of branching time temporal logic. Linear modular model checking of the assume-guarantee pair $\langle \varphi \rangle_{SAS} \langle \psi \rangle$ can be reduced to checking that a system $SAS$ satisfies
Chapter 6. Compositional Reasoning

190

the implication \( \varphi \rightarrow \psi \). This restricts the paths of \( \text{SAS} \) that should satisfy \( \psi \) to those that also satisfy \( \varphi \). More formally, for \( \varphi \) and \( \psi \in L_{\text{SAS}} \), we have

\[
\langle \varphi \rangle_{\text{SAS}} \langle \psi \rangle \quad \text{iff} \quad T_{\text{SAS}} \models \varphi \rightarrow \psi
\]

In the branching and linear-branching case this construction is not possible. In fact, a maximal model representing the assumption has to be constructed and composed with the model to be verified. If the guarantee can be verified on this composition, the assume-guarantee pair is established. A maximal model for an assumption can be constructed by tableau for linear time temporal logic (LTL) and universal formulae of branching time temporal logic (ACTL). For universal formulae of universal branching time temporal logic (ACTL*) only an automata-theoretic approach exist [KV97].

Thus, in order to implement the assume-guarantee reasoning strategy by model checking for SAS models, we have to decide whether to construct maximal SAS models for linear assumptions from \( L_{\text{SAS}} \) or for universal branching time assumptions from \( AB_{\text{SAS}} \). Theoretically, LTL and ACTL, and consequently also \( L_{\text{SAS}} \) and \( AB_{\text{SAS}} \), are incomparable. A detailed account of their common fragment can be found in [Mai00]. From a practical viewpoint, linear time temporal logic seems to be more natural for a system designer as it describes properties over all computation paths of a system. Such properties are often stronger than ACTL properties. While a designer might opt for a weak guarantee, she cannot be willing to choose a weak assumption. Rather, a strong assumption may be necessary in order to verify a weak guarantee. Often those assumptions are not expressible in ACTL. Furthermore, properties that can be expressed in ACTL but not in LTL occur very rarely in practice. Therefore, ACTL is not expressive enough for modular model checking according to Vardi [Var01]. For these reasons, we implement a maximal model construction for \( L_{\text{SAS}} \) properties in order to use model checking for assume-guarantee reasoning in the linear or linear branching case for SAS models. As \( L_{\text{SAS}} \subseteq A_{\text{SAS}} \), all results previously established for formulae \( \varphi \in A_{\text{SAS}} \) are also valid for \( L_{\text{SAS}} \) formulae. For an \( L_{\text{SAS}} \) formula \( \varphi \), the equivalent \( A_{\text{SAS}} \) formula is \( A \varphi \).

Maximal Model Construction by Tableau

A maximal SAS model for a \( L_{\text{SAS}} \) formula \( \varphi \) can be constructed by a tableau method presented in [CGP99]. A tableau of a formulae \( \varphi \) is a structure \( T_{\varphi} = (S, S_0, \leadsto, F) \) where \( S \) is the set of states, \( S_0 \) the set of initial states, \( \leadsto \subseteq S \times S \) the transition relation and \( F \subseteq S \) a fairness condition. A
6.2. Compositional Reasoning Strategies

path of the tableau $T_\varphi$ is a sequence of states $s_0 s_1 \ldots$ from $S$ where $s_0 \in S_0$ and for all $i \geq 0$ $s_i \leadsto s_{i+1}$, A path of this structure is called fair if it visits states in $F$ infinitely often. The tableau structure constructed in [CGP99] has the maximal model property, i.e. it satisfies $\varphi$, $T_\varphi \models \varphi$, and for all other structures $M'$, if $M'$ satisfies $\varphi$, $M' \models \varphi$, then $M'$ is similar to the tableau, $M' \preceq T_\varphi$.

In the following, we assume that the formula $\varphi$ only uses the next time operator $X$ and the until operator $U$. This can be achieved by transforming the property according to well-known temporal equivalences, such as

- $G \varphi \equiv \neg F \neg \varphi$
- $F \varphi \equiv \text{true} U \varphi$

Furthermore, we assume that atomic propositions in $\varphi$ are only equalities between variables and values. We define the set of fixed values $\text{Fixed}(v) \subseteq \text{Val}$ for a variable $v \in \text{Var}(\varphi)$ as the values $v$ explicitly assumes in atomic propositions of $\varphi$ by $\text{Fixed}(v) = \{ \text{val} \in \text{Val} | v = \text{val} \in \text{Atoms}(\varphi) \}$. Let $\text{Dom}(v) \subseteq \text{Val}$ denote the domain of $v$ which are all values $v$ assumes in an SAS model. The domain $\text{Dom}(v)$ for each $v \in \text{Var}(\varphi)$ has to be finite which can for instance be achieved by data domain abstractions (cf. Section 5.3). For constructing a maximal compatible model, we have to make sure that the maximal model contains all variables required for compatibility. This can be achieved by adding conjuncts to the property $\varphi$ for the required variables. The conjuncts are of the form $v = \ast$ denoting that the variable $v$ is set to an arbitrary value. $v = \ast$ is understood as $\bigvee_{\text{val} \in \text{Dom}(v)} v = \text{val}$ which supposes that $\text{Dom}(v)$ is finite. Clearly, it holds that $\varphi \equiv \varphi \land (v = \ast)$ for any $v \notin \text{Var}(\varphi)$.

In order to define the different parts of the tableau structure constructed in [CGP99], we need some additional notation. We define the set of elementary formulae for the property $\varphi$ by $el(\varphi)$ with

- If $\varphi \equiv \text{true}$ or $\varphi = \text{false}$, $el(\varphi) = \emptyset$.
- If $\varphi \equiv a$ where $a$ atomic, $el(a) = \{ a \}$.
- If $\varphi \equiv \neg \varphi_1$, $el(\varphi) = el(\varphi_1)$
- If $\varphi \equiv \varphi_1 \land \varphi_2$ or $\varphi = \varphi_1 \lor \varphi_2$, $el(\varphi) = el(\varphi_1) \cup el(\varphi_2)$.
- If $\varphi \equiv X \varphi_1$, $el(X \varphi_1) = \{ X \varphi_1 \} \cup el(\varphi_1)$
- If $\varphi \equiv (\varphi_1 U \varphi_2)$, $el(\varphi) = \{ X (\varphi_1 U \varphi_2) \} \cup el(\varphi_1) \cup el(\varphi_2)$

The set of states $S$ of the tableau structure $T_\varphi$ is defined as the powerset of the set of elementary formulae of $\varphi$ without unrealisable states. States are not realisable in an SAS module if
two atomic propositions are contained that require the same variable to have two different values, e.g. \((x = 5 \land x = 3)\).

- an atomic proposition \(v = val\) is not contained in a state, but \(\text{Fixed}(v) = \text{Dom}(v)\). This means that all values \(v\) can be set to are explicitly mentioned in an atomic proposition leaving no value \(v\) can be set to in this state.

Such an unrealisable state implies false, as it contains elementary formulæ that are not consistent with each other or unsatisfiable. In the LTL tableau construction of [CGP99], this case cannot occur as the set of atomic propositions \(\mathcal{AP}\) that a formula \(\varphi\) can be built from are independent of each other. Any two atomic propositions \(p\) and \(q\) \(\in\) \(\mathcal{AP}\) can be satisfied at the same time, i.e. \(p \land q \neq false\). This, however, is different for \(\mathcal{L}_{\text{SAS}}\) properties where atomic propositions do not have to be independent. Hence, we have to remove the inconsistent states from the tableau. The set of tableau states \(S\) is defined by

\[
S = \mathcal{P}(\text{el}(\varphi)) \setminus \{P \in \mathcal{P}(\text{el}(\varphi)) | P \models false\}
\]

The set of subformulæ of a property \(\varphi\) is defined by \(\text{sub}(\varphi)\) with

- If \(\varphi \equiv \text{true}\) or \(\varphi = \text{false}\) or \(\varphi = a\) where a atomic then \(\text{sub}(\varphi) = \{\varphi\}\).
- If \(\varphi \equiv \neg \varphi_1\), \(\text{sub}(\varphi) = \{\varphi\} \cup \text{sub}(\varphi_1)\)
- If \(\varphi \equiv \varphi_1 \land \varphi_2\) or \(\varphi = \varphi_1 \lor \varphi_2\), \(\text{sub}(\varphi) \equiv \{\varphi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2)\)
- If \(\varphi \equiv X \varphi_1\), \(\text{sub}(\varphi) = \{\varphi\} \cup \text{sub}(\varphi_1)\)
- If \(\varphi \equiv \varphi_1 \cup \varphi_2\), \(\text{sub}(\varphi) = \{\varphi\} \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2)\)

Furthermore, we have to define a function \(\text{sat}\) assigning a set of states to each elementary formula and each subformula of \(\varphi\). Intuitively, these are the states in which the respective subformula is true. The function \(\text{sat}\) maps \(\text{true}\), \(\text{false}\), the elementary formulæ and the subformulæ of \(\varphi\) to a set of states, i.e. \(\text{sat} : \{\text{true}, \text{false}\} \cup \text{el}(\varphi) \cup \text{sub}(\varphi) \to \mathcal{P}(\text{el}(\varphi))\) where

- \(\text{sat}(\text{true}) = S, \text{sat}(\text{false}) = \emptyset\)
- \(\text{sat}(\varphi_1) = \{s \mid \varphi_1 \in s\}\) where \(\varphi_1 \in \text{el}(\varphi)\). Note, that this also includes formulæ of the form \(X \varphi_2\).
- \(\text{sat}(\neg \varphi_1) = S \setminus \text{sat}(\varphi_1)\)
- \(\text{sat}(\varphi_1 \land \varphi_2) = \text{sat}(\varphi_1) \cap \text{sat}(\varphi_2)\)
- \(\text{sat}(\varphi_1 \lor \varphi_2) = \text{sat}(\varphi_1) \cup \text{sat}(\varphi_2)\)
- \(\text{sat}(\varphi_1 \cup \varphi_2) = \text{sat}(\varphi_2) \cup (\text{sat}(\varphi_1) \cap \text{sat}(X (\varphi_1 \cup \varphi_2)))\)
The initial states of the tableau are those states where $\varphi$ is initially true. These are the states that are contained in $\text{sat}(\varphi)$, thus

$$S_0 = \text{sat}(\varphi)$$

The transition relation $\leadsto$ is defined such that every elementary formula contained in a state is true in this state. If a formula $X\varphi_1$ is contained in a state, all states that are reachable via one transition should satisfy $\varphi_1$. In the other direction, if a formula $\varphi_1$ is not contained in a state, all reachable states should not contain $\varphi_1$. This leads to the definition of $\leadsto$ as follows:

$$s \leadsto s' \quad \text{iff} \quad \bigwedge_{(X\varphi_1) \in \text{el}(\varphi)} s \in \text{sat}(X\varphi_1) \iff s' \in \text{sat}(\varphi_1)$$

Unfortunately, this definition of the transition relation $\leadsto$ does not guarantee that for properties of the form $(\varphi_1 U \varphi_2)$ the eventuality $h$ ever becomes true. In the tableau, as it is right now, $\varphi_1 U \varphi_2$ is also true on a path where $\varphi_1$ holds infinitely often. We have to impose an additional condition on the transition relation in order to make sure that $\varphi_2$ is satisfied eventually. A path $\pi$ starting in $s$ only satisfies $\varphi_1 U \varphi_2$ iff we have $s \in \text{sat}(\varphi_1 U \varphi_2)$ and either $s \in \text{sat}(\varphi_2)$ or there is a later state $t$ on the path $\pi$ such that $t \in \text{sat}(\varphi_2)$. This requirement can be captured in a fairness constraint on the constructed structure. Thus, each state contained in the set $\{\text{sat}(\neg(\varphi_1 U \varphi_2)) \lor \varphi_2) \mid \varphi_1 U \varphi_2 \in \text{sub}(\varphi)\}$ must be visited infinitely often. This guarantees that either a state satisfying $h$ is visited infinitely often which immediately satisfies the eventuality property or that a state satisfying $\neg(\varphi_1 U \varphi_2)$, i.e. outside of $(\varphi_1 U \varphi_2)$, must be visited infinitely often. Such a state can only be reached by construction of the transition relation through a state in $\text{sat}(\varphi_2)$ which also guarantees the eventuality property.

After construction, the tableau can be minimised by removing states that are not reachable from an initial state. As [GK07] observes, the tableau for the negation of $\varphi$ has exactly the same states, but a complementary set of initial states. The states reachable from the initial states in the tableau can be marked in linear time by a breadth-first or depth-first graph traversal. Afterwards, all unmarked states can be removed.

**Definition 6.34 (Reduced Tableau).** Let $T_\varphi = (S, S_0, \leadsto, F)$ be the tableau for the LTL formula $\varphi$. Then we define $T_{\varphi\text{red}} = (S_{\text{red}}, S_{0\text{red}}, \leadsto_{\text{red}}, F_{\text{red}})$ by

- $S_{\text{red}} = S \setminus \{s \mid s \text{ not reachable from } s_0 \in S_0\}$
- $S_{0\text{red}} = S_0 \cap S_{\text{red}}$
\[ \sim_{red} = \sim \cap (S_{red} \times S_{red}) \]
\[ F_{red} = F \cap S_{red} \]

where \( s \) is not reachable iff there does not exists a path \( \pi = s_0s_1 \ldots \) with \( s_0 \in S_0 \) in \( T_\varphi \) such that for any \( i \geq 0 \), it holds that \( s_i = s \).

As an example for the presented tableau construction, consider the property \( \varphi = G(x = 5) \). This can be rewritten as \( \neg (true \ U (\neg x = 5)) \). The elementary formulae of \( \varphi \) are
\[ el(\varphi) = \{ X(true \ U (\neg x = 5)), x = 5 \} \]

The set of states of the tableau \( T_\varphi \) can be encoded as a 2-bit-vector, where 1 for the first bit encodes whether the first elementary formula is contained in the state and 0 that it is not contained and the same for the second bit. The set of states \( S \) is given by \( S = \{(00, (01), (10), (00))\} \). The \texttt{sat} sets of the subformulae and elementary formulae of \( \varphi \) are as follows:

- \( \text{sat}(true) = S \)
- \( \text{sat}(x = 5) = \{01, 11\} \)
- \( \text{sat}(\neg x = 5) = \{00, 10\} \)
- \( \text{sat}(X(true \ U (\neg x = 5))) = \{10, 11\} \)
- \( \text{sat}(true \ U (\neg x = 5)) = \text{sat}(x = 5) \cup \text{sat}(true) \cap \text{sat}(X(true \ U (\neg x = 5))) = \{00, 10\} \cup \{10, 11\} = \{00, 10, 11\} \)
- \( \text{sat}(\neg (true \ U (\neg x = 5))) = S \setminus \text{sat}(true \ U (\neg x = 5)) = \{01\} \)

The initial state is \( S_0 = \{01\} \). The transition relation \( \sim \) is defined as depicted in Figure 6.8. The fairness set is the set of states contained in \( \text{sat}(\neg (true \ U (\neg x = 5))) \lor (\neg x = 5) = \{01\} \cup \{00, 10\} = \{01, 00, 10\} \). The fair states are shaded in Figure 6.8. The only state that is reachable from the initial state is the initial state itself. The other states can be removed without changing the possible computation paths of the tableau.

**Encoding Tableau Structures into SAS Models**

Given the reduced tableau \( T_{\varphi_{red}} \) for a property \( \varphi \), we can encode it as an SAS model such that the induced transition system has exactly the same paths as the tableau. The SAS model obtained by the encoding is a maximal model for the property \( \varphi \) and can be used for implementing the assume-guarantee reasoning strategy over SAS models by model checking. In the remainder of this section, we will refer to the reduced tableau \( T_{\varphi_{red}} \) simply by \( T_\varphi \) as the reduction does not change the set of possible paths of the tableau.
The system $SAS_\varphi$ encoding the tableau $T_\varphi = (S, S_0, \leadsto, F)$ has only one module called $m_\varphi$ with one functional configuration that is always enabled, i.e. the guard is simply true. We encode the states of the tableau $S$ by local states of the module $m_\varphi$. We use a local variable state which takes as values the bitvector encoding of the elementary propositions contained in the respective tableau state $s \in S$. Therefore, we fix an ordering of the elementary formulae $el(\varphi)$ of $\varphi$. A bitvector $b$ encodes a state $s \in S$ of the tableau by a 1 at position $i$ if the $i$-th elementary formula is contained in $s$ and by a 0 otherwise. The module $m_\varphi$ has no adaptive local variables because it suffices to capture the local state in one functional variable.

**Definition 6.35 (Bitvector Encoding of Elementary Formulae).** Let $\varphi$ be a $\mathcal{LL}_{SAS}$ formula and $el(\varphi) = [el_1, \ldots, el_n]$ a list of the elementary formulae of $\varphi$. Let $T_\varphi = (S, S_0, \leadsto, F)$ be the reduced tableau for $\varphi$. We define $\text{bits}(s) = b_1 \ldots b_n$ as the bitvector corresponding to $s \in S$ where $b_i = 1$ iff $el_i \in s$. The inverse operation $\text{bits}^{-1}(b)$ constructs the state $s \in S$ from the bitvector $b$ by $el_i \in \text{bits}^{-1}(b)$ iff $b_i = 1$.

The input and output variables of $m_\varphi$, functional as well as adaptive, are constructed from the atomic propositions of $\varphi$. For any atomic proposition $(v = val) \in Atoms(\varphi)$, we generate a variable $v$. The distinction if $v$ is an input, output, functional or adaptive variable must be made dependent on the scenario in which the maximal SAS model is to be used such that variables to be connected are of the same kind. For instance, if the assumption of an assume-guarantee pair contains a proposition $x = 5$ where $x$ is
a functional output variable of the system part for which the assumption is expressed, \( x \) becomes a functional input variable. The sets of functional and adaptive input and output variables must be disjoint.

**Definition 6.36 (Input and Output Variables in Tableau Encoding).**

Let \( \varphi \) denote an \( \mathcal{L}_{\text{SAS}} \) formula where \( \text{Var}(\varphi) \) denotes its variables. Then \( \text{inputs}_d(\varphi) \subseteq \text{Var}(\varphi) \) and \( \text{inputs}_a(\varphi) \subseteq \text{Var}(\varphi) \) denote the functional and adaptive input variables of \( \varphi \), and \( \text{outputs}_d(\varphi) \subseteq \text{Var}(\varphi) \) and \( \text{outputs}_a(\varphi) \subseteq \text{Var}(\varphi) \) denote the functional and adaptive output variables of \( \varphi \), if the input and output variables form a partition of the variables of \( \varphi \), i.e. \( \text{Var}(\varphi) = \text{inputs}_d(\varphi) \sqcup \text{inputs}_a(\varphi) \sqcup \text{outputs}_d(\varphi) \sqcup \text{outputs}_a(\varphi) \).

The initial states of the tableau \( S_0 \) are the states where \( \varphi \) is satisfied. As there can be more than one initial state in the tableau, we nondeterministically choose the initial state of the SAS module \( m_\varphi \). As initialisation of the local variable state, we take the bitvector representing the selected initial tableau state, i.e. \( \text{bits}(s_0) \) for \( s_0 \in S_0 \). The input and output variables are initialised according to the atomic propositions contained in the selected initial tableau state. If a proposition \( v = \text{val} \) for an input or output variable \( v \) is contained in the state, the variable \( v \) is initialised with \( \text{val} \), i.e. \( \text{init}(v) = \text{val} \). A variable \( w \in \text{Var}(\varphi) \) for which no atomic proposition is included in the state has to be initialised nondeterministically with a value that is different from any value for this variable contained in any atomic proposition. Thus, \( w \) is initialised with \( \text{init}(w) = \text{choose}(\text{Dom}(w) \setminus \text{Fixed}(w)) \) where \( \text{choose} \) is a non-deterministic choice from a set \( S \), i.e. \( s = \text{choose}(S) \) iff \( s \in S \). This initialisation is necessary in order to make sure that each state satisfies exactly the propositions it contains. It holds that for any variable \( w \in \text{Var} \) we have \( \text{Dom}(w) \setminus \text{Fixed}(w) \neq \emptyset \) since else this state would be unrealisable.

The transition functions of the SAS module \( m_\varphi \) encoding the tableau \( T_\varphi \) are generated according to the tableau transitions. Figure 6.9 gives a schematic description of this translation. If there is a transition \( s \rightsquigarrow s' \) in \( T_\varphi \), there is also a transition in the \( \text{next}_\varphi \) function from \( \text{bits}(s) \) to \( \text{bits}(s') \). This transition is conditional in that the inputs the module \( m_\varphi \) receives have to comply with the atomic propositions on the input variables included in the state \( s' \). For each atomic proposition \( \text{in}_i = \text{val} \) for an input variable \( \text{in}_i \) contained in \( s' \), we add the condition \( \text{in}_i = \text{val} \) to the transition. Furthermore, for each input variable \( \text{in}_j \in \text{in} \cup \text{adapt}_\text{in} \) for which no atomic proposition is included in \( s' \), we add constraints that \( \text{in}_j \) is not set to
any of its fixed values, i.e. $\bigvee_{val \in Fixed} in_j \neq val$. Let $inputs(s')$ denote the conditions a state $s'$ imposes on the inputs:

$$inputs(s') = \bigwedge_{v \in in \cup adapt} \{ v = val \mid v = val \in s' \} \land \{ \bigvee_{val \in Fixed} v \neq val \mid v = val \notin s' \}$$

If there are more than one possible transition from $s$ with the same conditions on the inputs, we non-deterministically choose one of the possibilities.

For each state $s \in S$, we add a conditional assignment to the $next\_state$ function that is constructed as follows: Let $I = \{1, \ldots, n\}$ denote the indices of the transitions $s \leadsto s'_i$ originating from $s$ in $T_\varphi$. The set of indices $I$ can be split into a partition of disjoint subsets $I_j \subseteq I$ for $j = 1, \ldots, m$ ($m \leq n$) such that for any $j_1 \neq j_2$, $I_{j_1} \cap I_{j_2} = \emptyset$, and each $I_j$ contains the set of indices for those states that coincide on the input constraints, i.e. for any $j_1, j_2 \in I_j$ we have $inputs(s'_{j_1}) = inputs(s'_{j_2})$. For each set $I_j$, we build a conditional assignment $next\_state(I_j)$ that can be encoded as an SAS conditional assignment where for some $k \in I_j$,

$$next\_state(I_j) = \text{if } (state = bits(s) \land inputs(s'_k))$$
$$\quad \text{then } choose_{j \in I_j} (state = bits(s'_j))$$

Let $next\_state(s)$ denote the assignments in the $next\_state$ function corresponding to state $s$ which is defined by $next\_state(s) = [next\_state(I_j)]_{j \in \{1, \ldots, m\}}$.

The $next\_out$ function and the $adapt\_next\_out$ function are generated from the transitions of the reduced tableau as well. The construction is very similar to the construction of the $next\_state$ function except that the output variables are assigned instead of the local state variable. For a state $s' \in S$,
we define the set of functional output assignments $\text{outputs}_d(s')$ and the set of adaptive output assignments $\text{outputs}_a(s')$ the state $s'$ induces by

\[
\text{outputs}_d(s') = \{v := \text{val} \mid (v = \text{val}) \in s' \land v \in \text{out}\} \\
\quad \cup \{v := \text{choose}\{\text{val} \in \text{Dom}(v) \setminus \text{Fixed}(v)\} \mid \\
\quad \quad \quad v = \text{val} \notin s' \land v \in \text{out}\}
\]

\[
\text{outputs}_a(s') = \{v := \text{val} \mid (v = \text{val}) \in s' \land v \in \text{adapt}\_\text{out}\} \\
\quad \cup \{v := \text{choose}\{\text{val} \in \text{Dom}(v) \setminus \text{Fixed}(v)\} \mid \\
\quad \quad \quad v = \text{val} \notin s' \land v \in \text{adapt}\_\text{out}\}
\]

For each state $s \in S$ and each functional and adaptive output variable, we add a conditional assignment to the \textit{next}\_\textit{out} or the \textit{adapt}\_\textit{next}\_\textit{out} function, respectively. For all transitions $s \xrightarrow{i} s'$ where $i \in I = \{1, \ldots, n\}$, we again need the partition of $I$ into $I_j \subseteq I$ for $j = 1, \ldots, m$ such that $I_j$ contains the set of indices of the states that coincide on the input constraints. Then for each set $I_j$, we construct a conditional assignment that can be encoded as SAS conditional assignment in the \textit{next}\_\textit{out} function or the \textit{adapt}\_\textit{next}\_\textit{out} function, respectively, for some $k \in I_j$ by

\[
\text{next}\_\textit{out}(I_j) = \text{if } (\text{state} = \text{bits}(s) \land \text{inputs}(s'_k)) \text{ then choose}_{j \in I_j}(\text{outputs}_d(s'_j)) \\
\text{adapt}\_\textit{next}\_\textit{out}(I_j) = \text{if } (\text{state} = \text{bits}(s) \land \text{inputs}(s'_k)) \text{ then choose}_{j \in I_j}(\text{outputs}_a(s'_j))
\]

Let $\text{next}\_\textit{out}(s)$ denote the assignments in the \textit{next}\_\textit{out} function corresponding to state $s$ which is defined by

\[
\text{next}\_\textit{out}(s) = [\text{next}\_\textit{out}(I_j)]_{j \in \{1, \ldots, m\}}
\]

Further, let $\text{adapt}\_\textit{next}\_\textit{out}(s)$ denote the assignments of the \textit{adapt}\_\textit{next}\_\textit{out} function for the state $s$ defined by

\[
\text{adapt}\_\textit{next}\_\textit{out}(s) = [\text{adapt}\_\textit{next}\_\textit{out}(I_j)]_{j \in \{1, \ldots, m\}}
\]

Using the previously introduced notions, we can now define the SAS module $m_\varphi$ as encoding of the (reduced) tableau $T_\varphi$. The module is a non-deterministic SAS module with a fairness condition $F$. A fairness condition $F$ denotes a set of states of the SAS module that must be visited infinitely often on a fair path. Validity of a formula $\varphi$ can then be restricted to the fair paths. In the module $m_\varphi$, the fairness condition corresponds to the fairness condition of the tableau and requires that the local state variable $\text{state}$ assumes a bitvector encoding of the fair tableau states infinitely often. This ensures that the eventualities in until formulae are satisfied.
6.2. Compositional Reasoning Strategies

Definition 6.37 (SAS Module Encoding of Tableau). Let $T_{\text{red}} = (S, S_0, \leadsto, F)$ be the reduced tableau for $\varphi$. We define the corresponding non-deterministic SAS module $m_{\varphi}$ by

$$m_{\varphi} = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt})$$

where

- $\text{in} = \text{inputs}_d(\varphi)$ and $\text{out} = \text{outputs}_d(\varphi)$
- $\text{loc} = \{\text{state}\}$ where state ranges over the bitvector encodings of $S$, i.e. $\{\text{bits}(s) \mid s \in S\}$
- Select non-deterministically $s_0 \in S_0$, then $\text{init}(\text{state}) = \text{bits}(s_0)$ and for all $v \in \text{in} \cup \text{out}$, $\text{init}(v) = \text{val}_1$ for $(v = \text{val}_1) \in s_0$ or $\text{val}_1 \in \text{Dom}(v) \setminus \text{Fixed}(v)$ if $(v = \text{val}) \not\in s_0$
- $\text{confs} = \{\text{conf}\}$ with $\text{conf} = (\text{true}, \text{next}_\text{state}, \text{next}_\text{out})$ where
  - $\text{next}_\text{state} = \text{next}_\text{state}(s) | s \in S$
  - $\text{next}_\text{out} = \text{next}_\text{out}(s) | s \in S$
- $\text{adapt} = (\text{adapt}_\text{in}, \text{adapt}_\text{out}, \emptyset, \text{adapt}_\text{init}, \emptyset, \text{adapt}_\text{next}_\text{out})$ with
  - $\text{adapt}_\text{in} = \text{inputs}_a(\varphi)$ and $\text{adapt}_\text{out} = \text{outputs}_a(\varphi)$ and for all $v \in \text{adapt}_\text{out}$, $\text{adapt}_\text{init}(v) = \text{val}_1$ for $(v = \text{val}_1) \in s_0$ or $\text{val}_1 \in \text{Dom}(v) \setminus \text{Fixed}(v)$ if $(v = \text{val}) \not\in s_0$
  - $\text{adapt}_\text{next}_\text{out} = \text{adapt}_\text{next}_\text{out}(s) | s \in S$

The fairness condition $F_{\varphi}$ is defined by $F_{\varphi} = \bigvee_{s \in F} (\text{state} = \text{bits}(s))$ where $F = \{\text{sat}((\neg (g \cup h)) \lor h) \mid g \cup h \in \text{sub}(\varphi)\}$.

Note that in the definition of an SAS module $m_{\varphi}$ the strict separation between functional and adaptive behaviour is no longer maintained. While in standard SAS modules the adaptive transition function $\text{adapt}_\text{next}_\text{out}$ may only depend on adaptive variables, here it also depends on the functional state variable $\text{state}$ and on functional input variables. This is due to the automatic construction of the module $m_{\varphi}$ from the tableau. Whether functional and adaptive transitions can be separated depends on the structure of the property $\varphi$ for which the tableau is constructed.

Now, we have to prove that the SAS module $m_{\varphi}$ is indeed a maximal model for $\varphi$. First, we have to show that it satisfies $\varphi$, $T_{m_{\varphi}} \models \varphi$, and, second, that for any module $m'$ with $T_{m'} \models \varphi$, there is a simulation relation such that $T_{m'} \preceq_{[\varphi]} T_{m_{\varphi}}$. In this direction, we first establish a lemma stating that any state $s$ in the module $m_{\varphi}$ satisfies exactly those formulae $\psi$ for which the tableau state corresponding to the bitvector encoding $\text{state}(\text{loc})$ belongs to $\text{sat}(\psi)$. $\text{sat}(\psi)$ intuitively denotes the set of tableau states that satisfy $\psi$. 
The proof is by induction on the structure of the formula \( \psi \). As induction base, we need a lemma for the atomic propositions valid in a state \( s \) of \( m_\varphi \).

**Lemma 6.38 (Validity of Atomic Propositions).** For all atomic formulae \( a \in \text{Atoms}(\varphi) \) and for each state \( s \in S_{m_\varphi} \), it holds that

\[
\text{bits}^{-1}(s(\text{loc})) \in \text{sat}(a) \text{ iff } s \models a
\]

**Proof.** We assume that \( a = (v = \text{val}) \) as \( \text{Atoms}(\varphi) \) is restricted to equalities between variables and values as atomic propositions.

- **Left-to-Right-Direction:** Let \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(a) \). Then by definition of \( \text{sat} \), it holds that \( a \in \text{bits}^{-1}(s(\text{loc})) \). By construction of the tableau module \( m_\varphi \), we get that \( s(v) = \text{val} \) and thus \( s \models (v = \text{val}) \).

- **Right-to-Left-Direction:** For a proof by contradiction, assume that \( s \models a \), but \( \text{bits}^{-1}(s(\text{loc})) \notin \text{sat}(a) \). Then by definition of \( \text{sat} \), \( a \notin \text{bits}^{-1}(s(\text{loc})) \). But then by translation \( s(v) \neq \text{val} \), which contradicts \( s \models a \). \( \square \)

By induction, we lift this result to more complex subformulae of \( \varphi \). However, validity has to be restricted to fair paths. A path \( \pi \) of \( T_{m_\varphi} \) is fair iff it infinitely often visits states where the fairness condition is true, i.e. \( \pi = s_0s_1 \ldots \) is fair iff \( \{i \mid s_i \models F \varphi\} \) is infinite. A formulae \( \psi \) is fairly valid on a path \( \pi \), \( \pi \models_F \psi \), if \( \pi \models \varphi \) and \( \pi \) is fair. The restriction to fair paths is important such that a state \( s \) with \( s \in \text{sat}(\varphi_1 \lor \varphi_2) \) actually satisfies the formula \( \varphi_1 \lor \varphi_2 \). On a fair path, the eventuality \( \varphi_2 \) will become true while on unfair paths \( \varphi_1 \) can remain valid infinitely often without \( h \) holding once. The lemma yields that \( \varphi \) fairly holds on \( T_{m_\varphi} \), i.e. \( T_{m_\varphi} \models_F \varphi \), as the initial states \( s_0 \) are those states with \( s_0(\text{loc}) = \text{bits}(s') \) where \( s' \in \text{sat}(\varphi) \) by construction of the tableau.

**Lemma 6.39 (Validity of Subformulae).** For all subformulae \( \psi \in \text{sub}(\varphi) \) and for each state \( s \in S_{m_\varphi} \), it holds that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi) \) iff \( s \models_F \psi \).

**Proof.** By induction over the structure of \( \psi \).

- **Base Case:** If \( \psi = \text{true} \), then for every state \( s \) it holds that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\text{true}) \) and satisfies \( \text{true} \). If \( \psi = \text{false} \), there is no state \( s \) such that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\text{false}) \). The proposition holds trivially. Let \( \psi = a \) where \( a \in \text{Atoms}(\varphi) \). This case follows immediately from Lemma 6.38.
6.2. Compositional Reasoning Strategies 201

- Induction Step:
  - \( \psi = \neg \psi_1 \): Let \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\neg \psi_1) \). By definition of sat, we get that \( \text{bits}^{-1}(s(\text{loc})) \in S_{\tau_\psi} \setminus \text{sat}(\psi_1) \). By induction hypothesis, we have that for all \( s \in S_{m_\varphi} \), where \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \), it holds that \( s \models \psi_1 \), and for all \( \text{bits}^{-1}(s(\text{loc})) \not\in \text{sat}(\psi_1) \) that \( s \not\models \psi_1 \) which is equivalent to \( s \models \neg \psi_1 \).
  - \( \psi = \psi_1 \land \psi_2 \): Let \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1 \land \psi_2) \). By definition of sat, we have that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \cap \text{sat}(\psi_2) \) and consequently, that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \) and \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_2) \). Then, we can apply the induction hypothesis and obtain that \( s \models \psi_1 \) and \( s \models \psi_2 \). This yields \( s \models \psi \).
  - \( \psi = \psi_1 \lor \psi_2 \): Let \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1 \lor \psi_2) \). By definition of sat, we have that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \cup \text{sat}(\psi_2) \) and consequently, that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \) or \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_2) \). For which we can apply the induction hypothesis and obtain that \( s \models \psi_1 \) or \( s \models \psi_2 \). This yields \( s \models \psi \).
  - \( \psi = X\psi_1 \): Let \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(X\psi_1) \). By construction of \( \rightsquigarrow \) there is a transition \( s \rightsquigarrow s' \) with \( \text{bits}^{-1}(s'(\text{loc})) \in \text{sat}(\psi_1) \). For \( s' \) we can apply the induction hypothesis and obtain that \( s' \models \psi_1 \). This yields that \( s \models X\psi_1 = \psi \).
  - \( \psi = \psi_1 \cup \psi_2 \): Left-to-right-direction: By definition of sat either \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_2) \) or \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(X(\psi_1 \cup \psi_2)) \) and \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \). By induction hypothesis, we know that \( s \models \psi_2 \lor (X(\psi_1 \cup \psi_2) \land \psi_1) \). As we only consider fair paths, we know that on a fair path \( \pi = ss_1 \ldots \) there exists a state \( s_j \) such that \( s_j \models \psi_2 \). This implies that \( s \models \psi_1 \cup \psi_2 \).

Right-to-left-direction: Let \( s \models \psi_1 \cup \psi_2 \). Then it holds that \( s \models \psi_2 \lor (X(\psi_1 \cup \psi_2) \land \psi_1) \) for which we can apply the induction hypothesis and obtain that either \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_2) \) or \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(X(\psi_1 \cup \psi_2)) \) and \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1) \). By definition of sat, we get that \( \text{bits}^{-1}(s(\text{loc})) \in \text{sat}(\psi_1 \cup \psi_2) \).

Now, we have to establish that \( m_\varphi \) is consistently similar to all other SAS systems satisfying \( \varphi \). To this end, for an SAS system \( SAS' \models \varphi \) and the tableau module \( m_\varphi \), we define a relation \( R \). For all elementary formulae of \( \varphi \), it holds that a state \( s' \in \Sigma_{SAS'} \) is related to a state of the tableau module \( s \in S_{m_\varphi} \) by \( R(s', s) \) if and only if the bitvector encoding of the local state variable in \( s \) corresponds to a tableau state that contains all elementary formulae of \( \varphi \) that are satisfied in \( s' \). The next lemma captures
that the relation preserves validity of subformulae, which is necessary for the simulation proof.

**Lemma 6.40 (Subformulae Validity in Relation).** Let $\text{SAS}'$ be a system with $T_{\text{SAS}'} \models \varphi$ and $m_\varphi$ the tableau module for $\varphi$. Let $R \subseteq \Sigma_{\text{SAS}'} \times S_{m_\varphi}$ with $(s', s) \in R$ iff for all $p \in \text{el}(\varphi)$ it holds that $s' \models p$ iff $p \in \text{bits}^{-1}(\text{loc}(s))$. If $(s', s) \in R$, then for all $\psi \in \text{sub}(\varphi) \cup \text{el}(\varphi)$ it holds that $s' \models \psi$ iff $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\psi)$.

**Proof.** By induction on the structure of $\psi$.

- If $\varphi = \text{true}$ and $\varphi = \text{false}$, the proposition holds trivially.
- Let $\psi \in \text{el}(\varphi)$. Then by definition of $R$, we have that $s' \models \psi$ iff $\psi \in \text{bits}^{-1}(\text{loc}(s))$. By definition of $\text{sat}$, we have $\psi \in \text{bits}^{-1}(\text{loc}(s))$ iff $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\psi)$ as $\psi$ is elementary. This includes all atomic formulae and also formulae of the form $X\psi_1$.
- $\psi = \neg \psi_1$: By definition of negation, $s' \models \psi$ iff $s' \not\models \psi_1$. By induction hypothesis, this yields that $s' \not\models \psi$ iff $\text{bits}^{-1}(\text{loc}(s)) \not\subseteq \text{sat}(\psi_1)$. But by definition of $\text{sat}$, we get the equivalence that $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\neg \psi_1)$.
- $\psi = \psi_1 \land \psi_2$ and $\psi = \psi_1 \lor \psi_2$ can be easily checked using the induction hypothesis and the definition of $\text{sat}$.
- $\psi = \psi_1 \cup \psi_2$: By definition of $\cup$, $s' \models \psi$ holds iff either $s' \models \psi_2$ or $s' \models \psi_1$ and $s' \models X(\psi_1 \cup \psi_2)$. By induction hypothesis, we obtain that this holds iff either $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\psi_2)$ or $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\psi_1)$ and $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(X(\psi_1 \cup \psi_2))$. The last case has already been handled in the base case. By definition of $\text{sat}$, we get that this is equivalent to $\text{bits}^{-1}(\text{loc}(s)) \subseteq \text{sat}(\psi_1 \cup \psi_2)$. □

It remains to show that $R$ as in Lemma 6.40 is a simulation relation between the module $m_\varphi$ and all other SAS systems $\text{SAS}'$ satisfying $\varphi$. Note that this lemma and its proof do not require fairness. However, a consistent simulation also constitutes a fair simulation which is a simulation restricted to fair paths.

**Lemma 6.41 (Consistent Simulation).** For any system $\text{SAS}'$ with $T_{\text{SAS}'} \models \varphi$, there is a consistent simulation to $m_\varphi$ with respect to $\varphi$. It holds that

$$T_{\text{SAS}'} \preceq_{[\varphi]} T_{m_\varphi}$$

using the identity function as concretisation function $C$.

**Proof.** Let $R \subseteq \Sigma_{\text{SAS}'} \times S_{m_\varphi}$ with $(s', s) \in R$ iff for all $p \in \text{el}(\varphi)$ it holds that $s' \models p$ iff $p \in \text{bits}^{-1}(\text{loc}(s))$ as in Lemma 6.40.
6.2. Compositional Reasoning Strategies

- **Initial Simulation:** As $T_{SAS'} \models \varphi$, we know that every initial state $s'_0 \in \text{Init}'$ satisfies $\varphi$. By Lemma 6.40, we have that $s'_0 \models \varphi$ iff $\text{bits}^{-1}(\text{loc}(s)) \in \text{sat}(\varphi)$. But these are exactly the initial states of the tableau, and by our translation to SAS, we know that if $\text{bits}^{-1}(\text{loc}(s)) \in \text{sat}(\varphi)$ then $s \in \text{Init}$.

- **Step Simulation:** Let $s'_i \sim s'_{i+1}$ be a transition in $SAS'$ and $(s'_i, s_i) \in R$. Let $X \psi_1, \ldots, X \psi_n$ be the elementary formulae of $\varphi$ that $s'_i$ satisfies. Then $s'_{i+1}$ must satisfy $\psi_1, \ldots, \psi_n$. By definition of $R$, we know that for all $j$, it holds that $X \psi_j \in \text{bits}^{-1}(\text{loc}(s_i))$ and $\psi_j \in \text{bits}^{-1}(\text{loc}(s_{i+1}))$. By construction of $\rightarrow$, there is a transition in the tableau between $\text{bits}^{-1}(\text{loc}(s_i))$ and $\text{bits}^{-1}(\text{loc}(s_{i+1}))$, and by our translation to SAS, also a transition between $s_i$ and $s_{i+1}$, i.e., $s_i \rightarrow s_{i+1}$.

- **Consistency:** Let $(s', s) \in R$ and $s \models a$ for $a \in \text{Atoms}(\varphi)$. By Lemma 6.38, we have that $s \models a$ iff $\text{bits}^{-1}(\text{loc}(s)) \in \text{sat}(a)$. By definition of $\text{sat}$, it holds $\text{bits}^{-1}(\text{loc}(s)) \in \text{sat}(a)$ iff $a \in \text{bits}^{-1}(\text{loc}(s))$. By definition of $R$, we obtain that $a \in \text{bits}^{-1}(\text{loc}(s))$ iff $s' \models a$. \hfill $\square$

Based on the previous lemma that there exists a consistent simulation between $m_\varphi$ and any other SAS system $SAS'$ satisfying $\varphi$, we can use Theorem 5.3 in order to conclude that for any path $\pi'$ in $SAS'$ there exists a corresponding path $\pi$ in $m_\varphi$ such that corresponding states in the path satisfy the same subformulae of $\varphi$. The existence of corresponding paths implies the existence of corresponding fair paths.

**Lemma 6.42 (Corresponding Paths).** Let $SAS'$ be an SAS system with $T_{SAS'} \models \varphi$. Let $\pi'$ be a path of SAS such that $\pi' \models \varphi$. Then there exists a path $\pi$ in $m_\varphi$ such that $\pi \models \varphi$, and for all $s_i \in \pi$ and $s'_i \in \pi'$ and $\psi \in \text{sub}(\varphi)$, if $s_i \models \psi$, then $s'_i \models \psi$.

**Proof.** In Lemma 6.41, we have shown that $T_{SAS'} \preceq_{\{\varphi\}} T_{m_\varphi}$ with simulation relation $R(s', s)$ from Lemma 6.40. By Lemma 5.3, we get that for any path $\pi' = s'_0s'_1 \ldots$ of $SAS'$ there exists a corresponding path $\pi = s_0s_1 \ldots$ of $M_\varphi$ such that $R(s'_i, s_i)$ for all $i \geq 0$ holds. By Lemma 6.40, we know that if $(s'_i, s_i) \in R$, for all $\psi \in \text{sub}(\varphi)$, it holds that $s'_i \models \psi$ iff $\text{bits}^{-1}(\text{loc}(s_i)) \in \text{sat}(\psi)$. By Lemma 6.39, we can conclude that $s_i \models \psi$ which in particular holds for $s_0$ and $\varphi$. \hfill $\square$

Now we can show that the module $m_\varphi$ constructed from the tableau is a maximal model for $\varphi$ with respect to its fair paths. This result implies that we can use $m_\varphi$ as maximal model for the assumption in order to reduce assume-guarantee reasoning to a model checking problem.
Theorem 6.43 (Maximal SAS Module). The module $m_\varphi$ constructed from tableau as in Definition 6.37 is a maximal SAS model for $\varphi$ with respect to its fair paths.

Proof. By Lemma 6.39, we know that $T_{m_\varphi} \models F \varphi$. By Lemma 6.41, for all systems SAS with $T_{SAS} \models \varphi$, we know that $T_{SAS} \preceq [\varphi] T_{m_\varphi}$ using the identity function as concretisation function $C$ which yields the result. □

In order to compose the tableau module $m_\varphi$ with a subsystem for assume-guarantee reasoning, $m_\varphi$ has to be embedded into an SAS system. This is required by the definition of the composition operator. To this end, a system border with one system input and output variable for each module input or output variable of $m_\varphi$ and the respective connections are introduced. The names of the interface variables of the new system border are chosen corresponding to the projection of the subsystem $SAS_1$ for which the assumption is made. This ensures composability with the second subsystem $SAS_2$ for which the guarantee is to be shown. The following definition provides the notion of an enclosing system for a module $m$.

Definition 6.44 (Enclosing System). Let $m$ be an SAS module with $m = (in, out, loc, init, confs, adapt)$. By SAS$_m$ we define the enclosing SAS system where

- $M = \{m\}$
- $input_d = \{v_{sys} \mid v \in in\}$ and $output_d = \{v_{sys} \mid v \in out\}$
- $input_a = \{v_{sys} \mid v \in adapt_{in}\}$ and $output_a = \{v_{sys} \mid v \in adapt_{out}\}$
- $conn_d = \{(v_{sys}, v) \mid v \in in\} \cup \{(v, v_{sys}) \mid v \in out\}$
- $conn_a = \{(v_{sys}, v) \mid v \in adapt_{in}\} \cup \{(v, v_{sys}) \mid v \in adapt_{out}\}$

By Lemma 6.27, we know that a module $m$ satisfies a property $\varphi$ if and only if the projection of a system onto the module $m$ satisfies $\varphi$. The projection of a system onto one module and the embedding of a module into an enclosing environment are structurally equivalent. Thus, we know that for the module $m_\varphi$ in its enclosing environment it holds that $T_{SAS_{m_\varphi}} \models \varphi$ as $m_\varphi$ itself satisfies $\varphi$. Furthermore, we know that for all systems SAS$'$ satisfying $\varphi$, there is a consistent simulation to $m_\varphi$ and, thus, also a consistent simulation to $SAS_{m_\varphi}$. Therefore, SAS$_{m_\varphi}$ is a maximal system model for $\varphi$.

Theorem 6.45 (Maximal SAS System). Let $m_\varphi$ be the maximal module for $\varphi$. Then, the system SAS$_{m_\varphi}$ is a maximal system model for $\varphi$. 


6.2. Compositional Reasoning Strategies

Proof. Since $T_{m_\varphi} =_F \varphi$, by Lemma 6.27 we know that $T_{SAS_{m_\varphi}} =_F \varphi$. Further, we know that for all systems $SAS'$ with $T_{SAS'} = \varphi$, it holds $T_{SAS'} \preceq_{[\varphi]} m_\varphi$ with respect to a simulation relation $R$ and the identity function as concretisation function $C$. Then $R$ is also a simulation relation between $SAS'$ and $SAS_{m_\varphi}$ such that $T_{SAS'} \preceq_{[\varphi]} T_{SAS_{m_\varphi}}$ with respect to $\varphi$ and the identity function as concretisation function $C$, as $\text{Var}(\varphi) \subseteq \text{Var}(m_\varphi)$. □

Using the maximal model for $\varphi$ constructed from the tableau, we can now implement assume-guarantee reasoning for SAS models by model checking. For establishing the assume-guarantee pair $\langle \varphi \rangle_{SAS} \langle \psi \rangle$, we construct a maximal system model for the assumption $SAS_{m_\varphi}$ and compose it with the remaining part of the system. Then, we model check the composition $SAS \parallel SAS_{m_\varphi}$ in order to establish the validity of $\psi$.

If we cannot express the fairness constraint of the module $m_\varphi$ in the applied model checker, this has the following consequences for assume-guarantee reasoning: Let $m_\varphi^{-F}$ denote the module $m_\varphi$ without fairness constraints. If we can prove for the composition of the system $SAS$ with the unfair tableau module $m_\varphi^{-F}$ that it satisfies the guarantee $\psi$, i.e. $T_{SAS_{m_\varphi^{-F}} \parallel SAS} = \psi$, we can infer the validity of the assume guarantee pair $\langle \varphi \rangle_{SAS} \langle \psi \rangle$. The justification is as follows: As the proof of Lemma 6.41 does not require the notion of fairness, it holds that all systems $SAS'$ satisfying $\varphi$ are in consistent simulation with $m_\varphi^{-F}$, $T_{SAS'} \preceq_{[\varphi]} T_{m_\varphi^{-F}}$. If $SAS'$ and $SAS$ are composable, by Theorem 6.15, we obtain that $T_{SAS \parallel SAS_{m_\varphi^{-F}} \parallel SAS} \preceq_{[\varphi]} T_{SAS_{m_\varphi^{-F}} \parallel SAS}$. By transitivity of simulation and maximal model lemma, we get that $T_{SAS \parallel SAS} = \psi$ from which we conclude $\langle \varphi \rangle_{SAS} \langle \psi \rangle$. Thus, for establishing the validity of assume-guarantee pairs it is sufficient to consider the tableau structure without fairness constraints.

However, the other direction of the assume-guarantee reasoning rule with maximal models (Theorem 6.31) does not hold for the tableau without fairness. If we have $\langle \varphi \rangle_{SAS} \langle \psi \rangle$, we know that for all systems $SAS'$ that are composable with $SAS$ and satisfy $\varphi$, it holds that $T_{SAS' \parallel SAS} = \psi$. But as $m_\varphi$ only satisfies $\varphi$ on its fair paths, it does not hold in general that $T_{SAS_{m_\varphi^{-F}} \parallel SAS} = \psi$. Hence, it can be the case that we cannot show the assume-guarantee pair $\langle \varphi \rangle_{SAS} \langle \psi \rangle$ using the unfair tableau $m_\varphi^{-F}$ although the assume-guarantee pair actually is valid.

6.2.3 Interface Abstraction

Interface abstraction is a third compositional reasoning strategy proposed in [CLM89]. In the interface abstraction approach, the global system is
split into two disjoint subsystems. The first subsystem is the part of the global system the considered property refers to. The second subsystem is not directly affected by the property. The second subsystem is reduced with respect to the behaviour observable at its interface. Generally, this is significantly less complex than the original subsystem. Afterwards, the first subsystem and the reduced second subsystem are composed again. Composability of the first subsystem and the reduced second subsystem is ensured by the restriction that the second subsystem is reduced with respect to its interface behaviour which means that the interface itself remains unchanged. Formally, this is captured by requiring compatibility between the interfaces of the original and the reduced second subsystem. If the original second subsystem is consistently simulated by the reduced subsystem and if the considered property is from the universal fragment of $L_{SAS}$, it suffices to show the considered property over the composition of the first subsystem and the reduced second subsystem in order to conclude that also the original system satisfies the property. Figure 6.10 depicts this situation for the system $SAS$ of Figure 6.3. The property $\varphi$ refers only to the left system part $SAS_1$. The right system part $SAS_2$ is reduced with respect to its interface. If $\varphi$ can now be established over the composition of $SAS_1$ and $SAS'_2$, it also holds true in the original system $SAS$.

\begin{center}
\begin{tikzpicture}
  \node (SAS1) at (0,0) {$SAS_1$};
  \node (SAS2) at (3,0) {$SAS'_2$};
  \draw[->] (SAS1) -- (SAS2);
  \node (Var) at (-1,-3) {$\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(SAS_1)$};
  \node (TSAS1) at (0,-5) {$T_{SAS_1 || SAS'_2} \models \varphi$};
  \node (TSAS) at (3,-5) {$T_{SAS} \models \varphi$};
  \node (SAS2) at (3,-2) {$SAS_2 \sim SAS'_2$};
  \node (SAS2) at (3,-3) {$SAS_2 \preceq_{[\varphi]} SAS'_2$};
\end{tikzpicture}
\end{center}

Fig. 6.10. Interface Abstraction

Interface abstraction is a useful approach to reduce verification complexity if, first, simple decomposition induces a too coarse abstraction of the second subsystem’s behaviour by replacing it by the chaotic envi-
6.2. Compositional Reasoning Strategies

Ronen et al. Second, interface abstraction can be used instead of assume-guarantee reasoning if a suitable assumption strong enough to verify a guarantee cannot be provided. Additionally, interface abstraction is beneficial if a large assumption would be necessary for verification of a guarantee. The reason is that the complexity of assume-guarantee reasoning in the size of the assumption can be fairly high. As [KV97] argues, assume-guarantee reasoning is only tractable for small assumptions. In an assume-guarantee pair, assumptions and guarantees are often given as conjunctions $\langle \phi_1 \land \ldots \land \phi_n \rangle \land \langle \psi_1 \land \ldots \land \psi_m \rangle$. While it is possible to decompose the guarantee and verify $\langle \phi_1 \land \ldots \land \phi_n \rangle M \langle \psi_i \rangle$ in isolation, assumptions cannot be decomposed in a similar fashion. So application of assume-guarantee reasoning to overcome the state-explosion problem may result in an assumption-explosion problem as [KV97] puts it. Hence, interface abstraction is a way to avoid finding an assumption suitable for verifying a guarantee and, furthermore, to prevent a potential assumption-explosion by reducing system parts to their relevant interface behaviour with respect to a property.

The following theorem formulates the first interface abstraction rule for SAS models corresponding to the first assume-guarantee reasoning rule. It allows splitting a global system into two subsystems where the one not affected by the considered property is reduced with respect to its interface behaviour. This reduction is restricted such that the reduced subsystem has to be compatible with and consistently similar to the original subsystem. If a universal property holds over the composition of the reduced subsystem and the subsystem affected by the property, the property also holds over the original system. The proof uses the results of Theorem 6.15 that simulation is preserved by composition and of Theorem 5.8 that properties are preserved by consistent simulation.

**Theorem 6.46 (Interface Abstraction Rule).** Let $\text{SAS}$ be a system with $\text{SAS} = \text{SAS}_1 \parallel \text{SAS}_2$, and let $\text{SAS}_2'$ be a system with $T_{\text{SAS}_2} \preceq_{[\varphi]} T_{\text{SAS}_2'}$ and $\text{SAS}_2 \sim \text{SAS}_2'$. Let $\varphi$ be a $\mathcal{AL}_{\text{SAS}}$ property with $\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(\text{SAS}_1)$. Then it holds that

$$T_{\text{SAS}_1 \parallel \text{SAS}_2} \vdash \varphi \quad \text{implies} \quad T_{\text{SAS}} \vdash \varphi$$

**Proof.** By assumption, we have that $T_{\text{SAS}_2} \preceq_{[\varphi]} T_{\text{SAS}_2'}$. Since, $\text{SAS}_2'$ is compatible with $\text{SAS}_2$, by Theorem 6.15 we can compose both $\text{SAS}_2$ and $\text{SAS}_2'$ with $\text{SAS}_1$ and obtain the following simulation relationship:

$$T_{\text{SAS}_1 \parallel \text{SAS}_2} \preceq_{[\varphi]} T_{\text{SAS}_1 \parallel \text{SAS}_2'}$$
By property preservation by simulation in Theorem 5.8, we know that if $T_{SAS_1\parallel SAS_2} \models \varphi$ also $T_{SAS_1\parallel SAS_2} \models \varphi$. □

Note that in Theorem 6.46, the reduction of $SAS_2$ with respect to its interface behaviour towards the subsystem $SAS'_2$ is formalised by requiring a consistent simulation with respect to the considered property $\varphi$ and compatibility between $SAS_2$ and $SAS'_2$. But, as $\varphi$ only contains variables of the subsystem $SAS_1$, the simulation between $SAS_2$ and $SAS'_2$ is not further restricted. Also a completely non-deterministic subsystem $SAS'_2$ would simulate $SAS_2$ with respect to $\varphi$. If this chaotic subsystem $SAS_2$ is also compatible to $SAS'_2$, it satisfies the requirements of the theorem. In this scenario, the interface abstraction rule is still sound. But it is no better than decomposition in order to show a property over a reduced system. This is due to the fact that a non-deterministic subsystem $SAS'_2$ corresponds to a chaotic environment for $SAS_1$ considered in reasoning by decomposition. So, the closer $SAS'_2$ matches the interface behaviour of $SAS_2$, the more likely it is to establish the considered property over the composition of the first and the second reduced subsystem if the property is true in the original system. Additional restrictions on the simulation relation can capture that the interface behaviour of $SAS_2$ is closely reflected by $SAS'_2$, but those restrictions are not required for soundness of the interface abstraction rule.

Furthermore, the algorithm that reduces a subsystem to the relevant behaviour at its interface is not specified in the interface abstraction rules. The only requirement the reduction algorithm has to satisfy is that the reduced subsystem is in consistent simulation with the original subsystem with respect to the property $\varphi$ and that original and reduced subsystem are compatible. In our framework, we can, for instance, use the model transformation techniques presented in Chapter 5 in order to reduce a subsystem with respect to its interface behaviour. To this end, module slicing can remove all system parts not influencing the behaviour of the interface. However, the use of module slicing for interface abstraction limits the reduction potential. Performing module slicing on the overall system in general provides a better reduction as also the first subsystem that remains unchanged in interface abstraction can be reduced. If a more sophisticated abstraction technique for the second subsystem with respect to its interface behaviour is applied, reductions can be much larger. A possible approach in this direction would be to compute an overapproximation of the interface behaviour of the second subsystem by abstract interpretation based
techniques [CC77, CC79] and use this as reduced second subsystem for verification.

As corollary of the first interface abstraction rule, we can also formulate a conjunctive interface abstraction rule which corresponds to the second conjunctive assume-guarantee reasoning rule. It allows verifying a conjunctive property by repeatedly applying the interface abstraction rule for each conjunct. Furthermore, it is possible to use interface abstraction for a property build from a disjunction of formulae. Also, interleavings of conjunctions and disjunctions in a formula can also be reduced into simpler verification tasks over the single conjuncts or disjuncts by interface abstraction.

Corollary 6.47 (Conjunctive and Disjunctive Interface Abstraction Rules). Let \(\text{SAS} \) be a system with \(\text{SAS} = \text{SAS}_1 \parallel \text{SAS}_2\) and \(\varphi\) an \(\mathcal{AL}_{\text{SAS}}\) formula with \(\text{Var}(\varphi) \subseteq \text{Var}_{\text{mod}}(\text{SAS}_1)\). Further, let \(\text{SAS} = \text{SAS}_3 \parallel \text{SAS}_4\) and \(\psi\) be an \(\mathcal{AL}_{\text{SAS}}\) formula with \(\text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(\text{SAS}_3)\). If \(T_{\text{SAS}_2} \preceq_{[\varphi]} T_{\text{SAS}_2}^\prime\) and \(T_{\text{SAS}_4} \preceq_{[\psi]} T_{\text{SAS}_4}^\prime\) and \(\text{SAS}_2 \sim \text{SAS}_2^\prime\) and \(\text{SAS}_4 \sim \text{SAS}_4^\prime\), then it holds that

1. \(T_{\text{SAS}_1 \parallel \text{SAS}_2} \models \varphi\) and \(T_{\text{SAS}_3 \parallel \text{SAS}_4} \models \psi\) implies \(T_{\text{SAS}} \models \varphi \land \psi\)
2. \(T_{\text{SAS}_1 \parallel \text{SAS}_2} \models \varphi\) or \(T_{\text{SAS}_3 \parallel \text{SAS}_4} \models \psi\) implies \(T_{\text{SAS}} \models \varphi \lor \psi\)

Proof. By applying Theorem 6.46 twice, we obtain from \(T_{\text{SAS}_1 \parallel \text{SAS}_2} \models \varphi\) that \(T_{\text{SAS}} \models \varphi\) and from \(T_{\text{SAS}_3 \parallel \text{SAS}_4} \models \psi\) that \(T_{\text{SAS}} \models \psi\). By semantics of Boolean conjunction and disjunction, we get the results. \(\square\)

6.2.4 Compositional Verification of Functionality

Synchronous adaptive systems separate functionality from adaptation behaviour by implementing a set of predetermined functional configurations in each module. Configurations are activated depending on the module’s environment. This form of adaptation by predetermined reconfiguration offers potential for special compositional reasoning strategies exploiting the structural separation of functionality and adaptation behaviour. In this section, we present a compositional proof rule for properties referring to functional behaviour only. The proof rule allows verifying a functional property over the functional specification of a module in isolation if the module does not have a functional state. This approach is orthogonal to adaptive slicing (cf. Section 5.2.3) where for an adaptive property the functional behaviour can be removed.

A functional property of a synchronous adaptive system refers to functional variables only. Such a property can only be valid in a module if it
is true independent of which configuration is activated in the module in each cycle. Thus, the property must be at least valid in each configuration in isolation. However, if all configurations satisfy a property, the conclusion that also the module itself satisfies it is not possible. If we consider the behaviour of one configuration in isolation, the computation paths of the configuration consist of module states that are derived from the initial functional state using the next state function of this configuration alone. However, for the behaviour of a module all possible interleavings of configurations have to be considered. A module configuration assigns functional local and output variables according to its transition functions. As the active configuration of a module depends on the environment, it can change in each cycle. A configuration can become active in a state which was computed by another configuration. This state may not occur in a run of the configuration alone. Thus, in the module the configuration behaves differently as if it is analysed in isolation. This prevents to deduce the validity of a functional property in a module from the properties of single configurations.

The following example also shows that in presence of local functional variables functional behaviour cannot be verified in isolation. Consider a system for determining the occupancy of a room. In the occupancy detection module, it implements a camera-based occupancy detection, a motion-based occupancy detection and an entry-exit-based occupancy detection. The camera-based configuration uses a camera for detecting people in the room. The motion-based configuration uses a motion sensor for detecting persons in the room by their movements. The entry-exit-based detection recognises how many people are entering and exiting the room. It possesses a local functional variable capturing the number of persons currently present in the room. It is initialised with zero as in the initial state of the system no-one is inside the room. If one person enters the room, the counter is incremented by one. If a person leaves the room, it is decremented by one. A simple functional safety property of this module is that the counter is always equal or greater to zero. This property holds if the entry-exit-based occupancy detection configuration is considered in isolation as the value of the counter always reflects the number of persons inside the room. This number cannot become smaller than zero. If we now consider a run of the module also involving other configurations, we can no longer be sure that the counter corresponds to the number of people inside the room. If we start with the entry-exit-based detection, everything is fine until another configuration is used that does not keep track of the num-
ber of people inside the room. When switching back to the entry-exit-based detection, the value of the counter is no longer consistent with the room’s occupancy. Assuming that it was 1 when the configuration was left and that 2 persons entered the room while the configuration was not active, the counter can drop to -2 if all persons leave the room. This violates the property over the complete module while it was true for the entry-exit-based configuration in isolation.

As we can see from the example, verification of functional properties over modules with a functional local state cannot be done by analysing the configurations in isolation since configurations communicate with each other via the shared local state. Hence, soundness of a compositional proof rule for functional properties requires purely reactive modules without a functional local state. Properties interesting for reactive modules are invariants of the form \( \varphi = AG \varphi_1 \) where \( \varphi_1 \) does not contain any temporal operators because reasoning about a computation history is impossible without functional state.

In order to formulate a compositional proof rule for verifying functional invariants over reactive modules solely over the functional behaviour, we define the functionality implemented by a configuration as a configuration transition system. Therefore, we extract the functional variables and their initialisation function from an SAS module together with the next_state and the next_output functions of the considered configuration. We assume that the functionality is given by an SAS function description restricting the focus to deterministic functional behaviour. The set of states of a configuration transition system is the set of possible valuations of the functional variables. A state is initial if the variables are set to their initial values. A state transition is performed if the output variables are set according to the next_output function and the local variables are set according to the next_state function for a given functional input.

**Definition 6.48 (Transition System for Configuration).** For an SAS module \( m = (in, out, loc, init, confs, adapt) \) with configurations \( confs = \{ (guard_j, next_state_j, next_out_j) \} \), we define the configuration transition system for configuration \( conf_j \) by \( T_{conf_j} = (S, S_0, \rightarrow) \) where

- \( S \) is the set of states defined by \( S : in \cup out \cup loc \rightarrow Val \).
- \( S_0 \subseteq S \) is the set of initial states such that \( s_0 \in S_0 \) iff \( s_0(v) = init(v) \) for \( v \in in \cup loc \cup out \).
A transition $s \rightarrow s'$ between two states $s$ and $s' \in S$ can be performed if there exist $s'|_m$ such that $s'|_{out} = \text{next}_\cdot_{out}(s|_m \cup s'|_m)$ and $s'|_{loc} = \text{next}_\cdot_{state}(s|_m \cup s'|_m)$.

The following theorem formulates the compositional proof rule for functional invariants over modules without functional state. It allows deducing the validity of the invariant over the module from the validity of the invariant in the single configurations. The soundness proof uses the fact that without a functional local state the computed output only depends on the input that is the same for each configuration.

**Theorem 6.49 (Compositional Proof Rule for Functional Properties).** Let $m$ be an SAS module $m = (in, out, loc, init, confs, adapt)$ without functional state, i.e. $loc_m = \emptyset$, and $\text{AG} \varphi$ a functional invariant property from $L_{SAS}$ where $\varphi$ contains no temporal operators and $\text{Var}(\varphi) \subseteq \text{in}_m \cup \text{out}_m$. Then, the compositional proof rule for functional invariants

$$
\frac{T_{conf_j} \models \text{AG} \varphi \quad \text{for all } j = 1, \ldots, n}{T_m \models \text{AG} \varphi}
$$

is sound.

**Proof.** By hypothesis, we have that $T_{conf_j} \models \text{AG} \varphi$. By definition of the validity of $\text{AG} \varphi$, we have that for all reachable states $s_j \in S_j$ of $T_{conf_j}$, it holds that $s_j \models \varphi$. As $loc_m = \emptyset$, the computed output for each configuration $j$ only depends on the functional input. Since the input is the same for all configurations, whether in isolation or in the module, from the assumption that $s_j \models \varphi$ for any configuration $j$, we can conclude that $T_m \models \text{AG} \varphi$. \[\square\]

If the functionality of the configurations in a module $m$ is given by a pre/post-condition specification over the functional variables instead of an SAS functional description, the proof rule can also be applied for modules without functional state with respect to invariant properties. For each configuration, it has to be checked that the pre/post-condition specification implies the functional invariant. For a specification in higher-order logic, this implication can be checked by theorem proving.

Although the proof rule is fairly restricted, it is useful for verification of functional properties of the sensor modules in the vehicle stability control system example. Sensor modules simply react to input from the environment and convert it into output for controller modules. They do not have a
functional state such that checking functional invariants suffices. The different configurations are mainly used to increase fault-tolerance by reacting to changing conditions in the environment. Important properties of sensor modules are safety invariants, e.g. that the output of the module is always bigger than zero or that no division by zero occurs in the computation. As an example, consider the module $V_{\text{YawCalculation}}$ computing the yaw rate of the car. It has six different configurations as shown in Figure 2.4. If we want to check that no division by zero occurs in any functional configuration, we can apply the compositional proof rule for functional properties as the module does not have a functional state. It suffices to consider each functional configuration in isolation. The behaviour of the configurations $\text{Measured}$ and $\text{Off}$ is uncritical as no computations are performed. In the configuration $\text{SteeringBased}$, the yaw rate is computed by multiplication of the steering angle with a factor that depends on the reference velocity of the car. A division by zero is impossible in this configuration. If the yaw rate is derived from the front wheels in configuration $\text{FrontWheelBased}$, it is computed by taking the difference of the wheel speeds of the front wheels divided by the tread width times the cosine of the steering angle. As the cosine only becomes zero if the steering angle is 90 degrees which is impossible for the car, no division by zero can occur. In configuration $\text{RearWheelBased}$, the yaw rate is computed by the difference of the rear wheels divided by the tread width such that no division by zero can occur. But in configuration $\text{AyBased}$, the yaw rate is determined by dividing the lateral acceleration $ay$ by the speed of the car in longitudinal direction $v_{\text{carRef}}$. If the car does not move, the speed is zero which leads to a division by zero. Thus, we must add a constraint that this operation is only admissible if the reference velocity of the car is non-zero. This error was detected using the compositional proof rule for functional properties in an early version of the vehicle stability example and corrected to ensure absence of runtime errors.

### 6.2.5 Properties of the Controlled System

In embedded systems, properties of the controller depend on features of the controlled system. In our example, the properties of the vehicle stability control system are influenced by the characteristics of the car on which the system is implemented. In order to verify properties of adaptive embedded systems, the behaviour of the controlled system has to be handled appropriately. Assume-guarantee reasoning as presented in Section 6.2.2
only works on intra-system level. However, the same ideas can be adapted to the validity of properties of a controller in a controlled system. In this direction, we formulate the controlled system as an environment of the controller that is modelled by a synchronous adaptive system. The characteristics of the controlled system are formulated as assumptions on the environment. Establishing properties of the controller, i.e. the SAS model, then reduces to conditional validity of system properties under environment assumptions. Using the maximal model construction by tableau, conditional validity checking can be reduced to a standard model checking problem.

SAS models are inherently open reactive systems. They receive input from the environment, i.e. the controlled system, and produce output to the environment. An SAS system is called closed if it does not possess any input or output variables. An SAS environment can be defined as an SAS module that closes the considered system. To this end, the output variables of an SAS environment are connected to the inputs of the SAS system. The inputs of the SAS environment are connected to the outputs of the SAS system.

**Definition 6.50 (SAS Environment).** For an SAS system \( \text{SAS} = (M, \text{inputs}_a, \text{inputs}_d, \text{output}_a, \text{outputs}_d, \text{conn}_a, \text{conn}_d) \), an SAS module \( \text{Env}_{\text{SAS}} = (\text{in}, \text{out}, \text{loc}, \text{init}, \text{confs}, \text{adapt}) \) is an environment, if \( \text{in} = \{v_e \mid v \in \text{output}_d\} \), \( \text{out} = \{v_e \mid v \in \text{input}_d\} \), \( \text{adapt}_{\text{in}} = \{v_e \mid v \in \text{output}_a\} \) and \( \text{adapt}_{\text{out}} = \{v_e \mid v \in \text{input}_a\} \).

The composition of an SAS system with an corresponding SAS environment is defined by matching variable names. Composing an open SAS system with an corresponding SAS environment results in a closed SAS. This system contains the same set of modules as the original SAS with the additional environment module. The connections in the resulting SAS are the

![Fig. 6.11. Environment Composition](image-url)
same connections as in the original SAS. Only module variables previously
connected to system input and output variables in the original SAS are
now connected to the corresponding variables of the environment module
$Env_{SAS}$. Thus, the system border of the original system disappears. Figure
6.11 shows a system composed with an environment.

**Definition 6.51 (Environment Composition).** Let $SAS$ be an SAS sys-
tem and $Env_{SAS} = (in, out, loc, init, confs, adapt)$ an corresponding SAS envi-
ronment. We define the environment composition of the SAS system and the
environment $Env_{SAS}$ by $SAS \parallel_{env} Env_{SAS} =$

$$(M \cup \{Env_{SAS}\}, inputs_{ae}, inputs_{de}, output_{ae}, output_{de}, conn_{ae}, conn_{de})$$

where

- $inputs_{ae} = \emptyset$ and $inputs_{de} = \emptyset$
- $output_{ae} = \emptyset$ and $output_{de} = \emptyset$
- $conn_{a} = conn_{a} \cup \{(v_e, v') | v_e \in adapt_{out} \land \exists v \in inputs_{a}. (v, v') \in conn_{a}\}$
  $\cup \{(v', v_e) | v_e \in adapt_{in} \land \exists v \in output_{a}. (v', v) \in conn_{a}\}$
- $conn_{d} = conn_{d} \cup \{(v_e, v') | v_e \in out \land \exists v \in inputs_{d}. (v, v') \in conn_{d}\}$
  $\cup \{(v', v_e) | v_e \in out \land \exists v \in output_{d}. (v', v) \in conn_{d}\}$

This definition of environment composition is similar to system composi-
tion such that environment composition can be defined in terms of system
composition. We take the system $SAS$ composed with an environment mod-
ule $Env_{SAS}$ as defined above as the given system. The projection onto the
modules of the original system $SAS$ and the projection onto the environ-
ment module $Env_{SAS}$ by $\parallel$ leads to two composable SAS systems which are
exactly the same as the system and the environment before environment
composition. Composing the projected subsystems using the composition
operator $\parallel$ with respect to the given system leads exactly to the system
obtained by environment composition $\parallel_{env}$.

**Lemma 6.52 (Equality of System and Environment Composition).** Let
$SAS$ be a system and $Env_{SAS}$ an corresponding environment. Then

$$SAS \parallel_{env} Env_{SAS} = SAS \parallel_{SAS} Env_{SAS} \parallel_{env} Env_{SAS}$$

**Proof.** Projecting $SAS \parallel_{env} Env_{SAS}$ onto its parts leads to $\parallel SAS = SAS$ and
$\parallel Env_{SAS} = Env_{SAS}$. Both subsystems are composable by $\parallel_{SAS} Env_{SAS}$. By
Lemma 6.13, we can infer that $SAS \parallel_{env} Env_{SAS} = SAS \parallel_{SAS} Env_{SAS} \parallel_{env} Env_{SAS}$. □
SAS environments can be used to express formally that a system (controller) property is conditionally valid under an environment assumption which represents the characteristics of the controlled system. An SAS system conditionally satisfies a property \( \psi \) with respect to an environment assumption \( \phi \) if for all SAS environments satisfying \( \phi \) the system composed with the environment satisfies \( \psi \). As in assume-guarantee reasoning, we denote this as triple \( \{ \phi \} \text{SAS}\{ \psi \} \). It is important to note that the environment assumption \( \phi \) may only refer to the behaviour of the system \( \text{SAS} \) at its interface, i.e. the input and output variables, whereas the property \( \psi \) of the system \( \text{SAS} \) may refer to the system’s internal behaviour. Furthermore, for a correct formal account we have to transform the assumption \( \phi \) imposed on the input and output of \( \text{SAS} \) to a condition \( \phi' \) expressed over the environment variables in order to check that an environment satisfies the condition. In assume-guarantee reasoning, this is not necessary as the assumption \( \phi \) over the first system part \( \text{SAS}_1 \) is formulated with respect to the module variables in \( \text{SAS}_1 \).

**Definition 6.53 (Conditional Validity).** Let \( \text{SAS} \) be a system and \( \phi \) an \( \mathcal{AL}_{\text{SAS}} \) property with \( \text{Var}(\phi) \subseteq \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d \) and \( \psi \) a \( \mathcal{AL}_{\text{SAS}} \) property with \( \text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(\text{SAS}) \). Further, let \( \phi' = \phi[v/v_e] \) denote a \( \mathcal{AL}_{\text{SAS}} \) property that is derived from \( \phi \) by replacing every occurrence of \( v \in \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d \) by \( v_e \) for \( v_e \in \text{Env}_{\text{SAS}} \). The property \( \psi \) is conditionally valid for the system \( \text{SAS} \) under the environment assumption \( \phi \), denoted by \( \{ \phi \} \text{SAS}\{ \psi \} \), iff for all environments \( \text{Env}_{\text{SAS}} \) with \( T_{\text{Env}_{\text{SAS}}} | = \phi' \) the composition satisfies \( \psi \), i.e. \( T_{\text{Env}_{\text{SAS}}} | \| \text{SAS} | = \psi \).

Conditional validity can be reformulated in terms of maximal environment models. This is analogue to expressing validity of assume-guarantee reasoning pairs by maximal models. A maximal environment model \( \text{Env}_\phi \) is an SAS environment that is a maximal model for a property \( \phi \), this means that the environment satisfies \( \phi \), \( T_{\text{Env}_\phi} | = \phi \), and for all other environments \( \text{Env}'_{\text{SAS}} \) for SAS satisfying \( \phi \), there is a consistent simulation \( T_{\text{Env}'_{\text{SAS}}} \preceq_{|\phi} T_{\text{Env}_\phi} \) using the identity function as concretisation. A maximal environment model is always composable with the corresponding system.

**Definition 6.54 (Maximal Environment Model).** \( \text{Env}_{\text{SAS}_\phi} \) is a maximal environment model for a system \( \text{SAS} \) with respect to a property \( \phi \in \mathcal{AL}_{\text{SAS}} \) iff, first, \( \text{Env}_{\text{SAS}_\phi} \) satisfies \( \phi \), i.e. \( T_{\text{Env}_{\text{SAS}_\phi}} | = \phi \), and second, for all other environments \( \text{Env}'_{\text{SAS}} \) with \( T_{\text{Env}'_{\text{SAS}}} | = \phi \), it holds that \( T_{\text{Env}'_{\text{SAS}}} \preceq_{|\phi} T_{\text{Env}_{\text{SAS}_\phi}} \) using the identity function as concretisation \( C \).
6.2. Compositional Reasoning Strategies

Establishing the validity of a controller property under an assumption on the controlled system can be reduced to a standard model checking problem using maximal environment models. For soundness of the proof rule, environment assumption and system property have to be from the universal fragment of $\mathcal{L}_{SAS}$. In practice, environment assumptions are often intuitively expressed in linear time temporal logic that is included in the universal fragment because assumptions over all behaviours of the environment have to be expressed.

**Theorem 6.55 (Conditional Validity by Maximal Environment Models).** Let $SAS$ be a system, $\psi \in \mathcal{AL}_{SAS}$ a system property over $SAS$ with $\text{Var}(\psi) \subseteq \text{Var}_{\text{mod}}(SAS)$ and $\varphi \in \mathcal{AL}_{SAS}$ an assumption over the environment referring to the interface variables of $SAS$, $\text{Var}(\varphi) \subseteq \text{input}_a \cup \text{input}_d \cup \text{output}_a \cup \text{output}_d$. Then for an maximal environment model for $\varphi$, $\text{Env}_{SAS\varphi}$, it holds that

$$\{\varphi\}_{SAS}\{\psi\} \iff T_{\text{Env}_{SAS\varphi}\parallel SAS} = \psi$$

**Proof.**

• Left-to-right-direction: Assume $\{\varphi\}_{SAS}\{\psi\}$. By definition, for all environments $\text{Env}_{SAS}$ with $T_{\text{Env}_{SAS}\parallel SAS} \models \varphi'$ the composition satisfies $\psi$, i.e. $T_{\text{Env}_{SAS}\parallel SAS} \models \psi$. So, in particular $T_{\text{Env}_{SAS\varphi}\parallel SAS} \models \psi$.

• Right-to-left-direction: Assume $T_{\text{Env}_{SAS\varphi}\parallel SAS} \models \psi$. As $\text{Env}_{SAS\varphi}$ is the maximal environment, for all other environments $\text{Env}_{SAS}'$ satisfying $\varphi'$, we have $T_{\text{Env}_{SAS}'\parallel SAS} \leq_{[\varphi]} T_{\text{Env}_{SAS\varphi}}$. By Lemma 6.52, we can apply the result of Theorem 6.15 that composition preserves simulation to obtain $T_{\text{Env}_{SAS}\parallel SAS} \leq_{[\varphi]} T_{\text{Env}_{SAS\varphi}\parallel SAS}$. By assumption and transitivity of simulations, we get the result that $T_{\text{Env}_{SAS}\parallel SAS} \models \psi$ and subsequently $\{\varphi\}_{SAS}\{\psi\}$. \qed

For construction of a maximal environment model, the maximal model construction by tableau for an assumption $\varphi \in LL_{SAS}$ presented in Section 6.2.1 can be applied. Therefore, the environment assumption $\varphi$ has to be transformed such that it refers to the input and output variables of the environment as in Definition 6.53. Furthermore, conjuncts of the form $v_e = *$ have to be added for all variables at the system border of the system $SAS$ not referenced explicitly in $\varphi$ in order to obtain an environment composable with $SAS$. Then a tableau for the transformed property $\varphi$ is constructed and encoded into an $SAS$ module $m_\varphi$ as described in Definition 6.37 which is exactly the desired maximal environment model. By Theorem 6.43, we know that $m_\varphi$ satisfies $\varphi$ on its fair paths and, furthermore, that for all other environments $T_{\text{Env}} \models \varphi$ there is a consistent simulation.
with respect to $\varphi$, i.e. $T_{\text{Env}} \preceq_{[\varphi]} T_{m_\varphi}$. For establishing conditional validity, the constructed maximal environment model $m_\varphi$ is composed with the system $SAS$ by environment composition to $m_\varphi \parallel_{\text{env}} SAS$. Then the composed system is translated to the input of a model checker which checks the validity of the system property $\psi$. If $\psi$ holds, conditional validity of $\{\varphi\}SAS\{\psi\}$ is established. If we cannot express fairness properties over the paths of $m_\varphi \parallel_{\text{env}} SAS$, the same consequences for conditional validity apply as for assume-guarantee reasoning. If we can verify $T_{m_\varphi \parallel_{\text{env}} SAS} \models \psi$, we can conclude that $\{\varphi\}SAS\{\psi\}$ holds. If we fail to prove $T_{m_\varphi \parallel_{\text{env}} SAS} \models \psi$, we do not know whether $\{\varphi\}SAS\{\psi\}$ actually holds in $SAS$ because the unfair paths in the composed system may invalidate the property.

Verification of conditional validity of controller properties under assumptions on the controlled system can be combined with model transformations in order to reduce verification complexity. Suppose, we want to show that $\{\varphi\}SAS\{\psi\}$ holds, but the system $SAS$ contains infinite data domains not suitable for model checking. Then, the system $SAS$ can be transformed using data domain abstraction such that the abstract system $\hat{SAS}$ only uses finite data domains. Additionally, both the assumption $\varphi$ is abstracted to $\hat{\varphi}$ and the system property $\psi$ to $\hat{\psi}$. From the validity of $\{\hat{\varphi}\}\hat{SAS}\{\hat{\psi}\}$, we can infer that also $\{\varphi\}SAS\{\psi\}$ holds as data domain abstraction is property-preserving for universal properties.

### 6.3 Combining Verification Complexity Reductions

Compositional reasoning strategies and model transformations can be combined in order to increase the potential verification complexity reduction. In fact, the presented techniques constitute building blocks to be assembled depending on the structure of the system and the property to be verified. The transformation building blocks in our framework are model reductions by slicing (cf. Section 5.2) and data domain abstractions (cf. Section 5.3). The compositional reasoning strategies for SAS models are:

- Decomposition (cf. Section 6.2.1)
- Assume-Guarantee Reasoning (cf. Section 6.2.2)
- Interface Abstraction (cf. Section 6.2.3)
- Compositional Verification of Functionality (cf. Section 6.2.4)

A combination of reduction building blocks has to take the property to be verified into account as some reductions preserve full $L_{SAS}$ while others are...
restricted to the universal fragment. Furthermore, some reduction techniques can be used automatically while others require user interaction. For instance, assume-guarantee reasoning requires finding an assumption strong enough to verify the guarantee. For data domain abstractions, the user has to determine how data domains are to be abstracted in order to be able to verify a property. In contrast, model reduction by slicing can be used fully automatically requiring only the system and the property to be verified. For correctness of combinations of reduction building blocks, we rely on the results establishing property preservation over combined transformations (cf. Section 6.1.1).

The building blocks for verification complexity reduction can be applied in any order as long as the resulting system is a valid SAS model. However, the order in which different reduction building blocks are applied has consequences for the analysis effort required for the reduction and also for the actual reduction itself. Nevertheless, there are general guidelines which building blocks can be applied for which class of verification problems and in which order they are used best. Verification complexity reduction should always start with building blocks offering large reductions of the considered model. This avoids doing reductions or transformations on system parts that are removed later by another technique. For instance, if we first abstract the complete system and by decomposition analyse only one module in detail, abstraction of the other modules was in vain. To this end, verification complexity reduction should start with compositional reasoning strategies if applicable since these in general return smaller verification problems. Then model transformations can be applied on the remaining verification tasks. This also ameliorates the effort for computing transformations as the system to be transformed is in general smaller.

Figure 6.12 shows graphically how the proposed verification complexity reduction building blocks for a given system and a given property to be verified can be assembled. First, it is determined whether compositional reasoning strategies are applicable. This means the property $\phi$ to be verified has to be from the universal fragment of $L_{SAS}$. Further, it should only affect a subset of the modules in the system $\bigcup_i m_i$ with $\text{Var}(m_i) \subseteq \text{Var}(\phi)$. Otherwise, model transformations have to be performed on the complete system.

When compositional reasoning is possible, decomposition techniques should be used first because these are the simplest compositional methods offering the largest reduction potential. On the decomposed system, further reductions and transformations are possible. If after decomposi-
tion only verification tasks considering purely functional properties over single modules without a functional state are left, we can apply the compositional proof rule for functional properties (FuncCompR). Afterwards, it is also possible to use model transformations for reducing the configurations further, e.g. abstraction of functional data domains. If the property is not purely functional, the module has got a functional state or more than one module is involved in the verification task, model transformations can be directly applied after decomposition. If the universal property then can be established, we are done.

However, decomposition poses the risk that a property cannot be verified over the decomposed system although it is valid in the original system. In this case, other compositional reasoning strategies can be applied. If we can find an assumption reflecting the behaviour of the environment of the considered subsystem in the original system, we can use assume-guarantee reasoning. The following model transformations by slicing and abstraction have to consider the assumption as well as the property to be shown as guarantee. For slicing, the slicing criterion has to include the assumption and the guarantee as the validity of both has to be preserved. In case of abstractions, the subsystem has to be abstracted together with the

![Diagram](image)

**Fig. 6.12.** Composition of Reduction Building Blocks
assumption and the guarantee. For assume-guarantee reasoning by model checking, a maximal model for the abstracted assumption is generated. If we can prove the validity of the assume-guarantee pair \( \langle \bar{\phi} \rangle \text{SAS}_2 \langle \bar{\psi} \rangle \) in the abstract, we can also infer the concrete assume-guarantee pair \( \langle \phi \rangle \text{SAS}_2 \langle \psi \rangle \) as abstractions are property preserving for universal properties. If we can now establish the assumption \( \varphi \) either over the original system part \( \text{SAS}_1 \) or over an abstraction that can differ from the previously applied, we can deduce that \( \text{T}_{\text{SAS}_1|\text{SAS}_2} = \psi \) holds.

If no assumption can be found, interface abstraction can be used. Here, the part of the system directly affected by the property remains unchanged while the remaining system is reduced with respect to its interface behaviour simplifying the overall verification task. After compositional reasoning strategies have been considered, model transformations can further reduce verification complexity on the remaining verification tasks. Transformations should start with slicing before abstractions are carried out. This avoids unnecessary abstractions as otherwise an abstracted system part may be removed in the subsequent slicing step.

While the proposed combination of model transformations mainly aims at verification complexity reduction towards model checking, compositional reasoning strategies are also beneficial for verification by theorem proving. Compositional reasoning strategies can be formalised inside a theorem prover and used to split complex proof goals into smaller subgoals, usually simpler and easier to verify interactively. Model transformations are not as important for theorem proving. Abstraction of integer data domains is, for instance, not necessary as most theorem provers can deal with integers directly such that the loss of precision induced by abstraction can be avoided. Slicing can reduce the model to be verified interactively by removing variables and functional assignments. This increases clarity for the user of the theorem prover, but is not vital for the verification success as a theorem prover works on a symbolic system representation instead of on the state space of a model.

### 6.4 Related Work

Verification complexity reduction by compositional reasoning is used widely in the literature. In this section, we first review related work on compositional proof rules for temporal logics that are related to decomposition of SAS models and compositional verification of functionality. Second, we consider the application of assume-guarantee reasoning in different domains.
Analogue to the construction of a maximal SAS model for a property, we look at approaches generating an environment consistent with an assumption. Third, we discuss related work on interface abstraction, before we review combinations of different reduction techniques for more efficient verification.

**Compositional Reasoning for Temporal Logics**

In [FMS97], a deductive proof system is developed for compositional verification of LTL properties over modular transition systems. It relies modular proof rules for LTL introduced in [MP92]. The proof rules refer to the composition operators used for building modular transition systems and reflect their semantics. Their correctness is justified via refinement arguments which closely corresponds to our notion of property preservation by simulation. In particular, the definition of the case distinction operator in modular transition systems is closely related to the definition of configurations in SAS. Case distinction in a module means that a module implements a different behaviour dependent on the validity of a first-order logic guard formula. Therefore, the modular validity rule resembles our proof rule for isolated verification of functional properties. The proposed approach is implemented in the STeP [SUM99] system. It uses deductive methods to split the overall verification task into smaller subtasks that are verified using model checking. Kesten and Pnueli in [KP05] propose a compositional reasoning approach for CTL* combining model checking with deductive methods. Using deduction rules in a fully axiomatized proof system, general CTL* specifications are decomposed into basic assertional formulae which can be more easily verified. A compositional proof system for the modal μ-Calculus can be found in [ASW94]. McMillan [McM99] introduces temporal case splitting as a compositional verification technique. Here, a property is established for each value of a finite data domain. From this, the general validity of a property over all data values of the domain can be inferred.

**Assume-Guarantee Reasoning**

The assume-guarantee paradigm for temporal logic is introduced by Pnueli in [Pnu85]. The primitive formulae in his logic are triples of the form \( ⟨ϕ⟩P⟨ψ⟩ \) denoting that the process \( P \) behaves according to \( ψ \) if the environment is consistent with \( ϕ \). [Pnu85] uses linear modular model checking as both assumption and guarantee are expressed in LTL. In [GL94], branching modular model checking is pursued while [Jos87] focusses on linear branching modular model checking. In [GL94], the authors propose
a maximal model technique to reduce branching modular model checking to standard model checking. A maximal model for an ACTL formula is constructed by a tableau method. It is shown that the validity of an assertion \( \langle \varphi \rangle M \langle \psi \rangle \) where \( \varphi \) and \( \psi \) are ACTL formulae is equivalent to the fact that the module \( M \) composed with the maximal model for the assumption \( \varphi \) satisfies \( \psi \). In [KV97], it is shown how to reduce branching and linear-branching modular model checking to standard model checking procedures using automata-theoretic results.

[PMT02] applies assume-guarantee reasoning to hardware systems expressed in Verilog HDL. Assume-guarantee reasoning for JAVA programs is considered in [PDH99]. The approach in [GK07], uses assume-guarantee style reasoning for verification of aspect-oriented programs. In [SGH04], assume-guarantee reasoning with maximal models as proposed in [GL94] is adapted to the compositional verification of Java card applets. In this work, a global property of the overall system is verified assuming a specification for each applet that can be loaded. When loading an applet, only the assumption over this applet has to be established in order to ensure that the complete system satisfies its global properties. This is very similar to our approach dealing with characteristics of the controlled system.

In the literature, there are several approaches for the automatic construction of a system model consistent with an assumption. [PMT02] presents an approach to generate an environment in Verilog HDL satisfying an ACTL specification using a tableau-based procedure. The approach in [PDH99] deals with an automatic construction of closing environments for program fragments in JAVA in order to apply software model-checking. The enclosing environments consist of drivers for active objects and stubs for passive objects in the program. Environment assumptions can be specified using regular expressions over method calls coming from the environment. Furthermore, construction of environments consistent with LTL assumptions using a maximal model construction by tableau as in [CGP99] is supported. However, eventuality properties are not dealt with as fairness is ignored. The follow-up work [TD03] presents a method to automatically generate safe approximations for a Java program environment by overapproximation of the side effects of invoked methods. In the context of aspect-oriented programming, [GK07] proposes an approach to verification of aspect-oriented programs using maximal models. The base program is only specified by an LTL assumption whereas the aspects to be woven in are given explicitly. By an LTL tableau technique, a model for the base program is constructed and augmented with the aspects. Properties verified
over this augmented model hold true over all base programs satisfying the LTL assumption if combined with the explicitly given aspects.

Interface Abstraction

Interface processes as a means to reduce verification complexity for model checking are introduced in [CLM89]. A general interface rule is established that provides the basis for the interface abstraction approach. This interface rule is very similar to the interface abstraction rules for SAS. In [BCC97], interface processes are constructed by cone of influence reduction of system parts with respect to their interface behaviour. [AdAHM99] proposes an approach for the automatic generation of abstract interface modules by removal of variables in a process interface by existential quantification for overapproximation of the module’s behaviour. The overapproximation is refined using reachability information collected in an initial model checking run of the module in the most general environment and by controllability information obtained from generating an environment assumption for each process.

In [GSL96], interface specifications are used in order to perform compositional minimisation of finite-state systems. Iteratively, a composition of an already reduced partial system with an additional module is reduced subject to an interface specification given by the system designer. The control abstraction proposed in [KP98] is closely related to interface abstraction. In a system containing n similar processes, n-1 processes are abstracted with respect to their observable interface behaviour by a so called network invariant. The network invariant is then used to establish the overall system specification over its composition with one process. In [FMS97], interface abstraction replaces a module by the maximal model for the interface behaviour subject to a refinement relation. Interface abstraction is, furthermore, shown to be beneficial in the compositional verification of Java card applets [GH05] because only the interface behaviour of an applet may be publicly known at verification time.

Combining Reduction Techniques

Combining compositional reasoning techniques with system transformations can increase the potential reduction of verification complexity. In [SBL99], a combination of data abstraction and compositional reasoning is applied in the verification of a sliding window protocol in order to alleviate the state-explosion problem. In [BCC97], assume-guarantee reasoning with maximal models [GL94] in combination with interface abstraction is
used for efficient verification of the Futurebus+ standard cache coherence protocol. Compositional data abstraction is proposed in [LGS+95], where each module is abstracted in isolation. The composition of the abstract modules yields a property-preserving abstraction of the original system. [KP98] advocates to use modularisation and abstraction jointly in order to make formal verification of practical use. In that work, data abstraction on intra-module level is combined with control abstraction of the system structure. The STeP deductive model checking framework [SUM99] implements a combination of modular reasoning and abstraction as well as assume-guarantee reasoning for LTL. MOCHA [AHM+98], a model checking tool for reactive modules and alternating temporal logic specifications, incorporates assume-guarantee style reasoning and interface abstraction.
Chapter 7

Experimental Evaluation

"There are sadistic scientists who hurry to hunt down errors instead of establishing the truth."
(Marie Curie)

The integration of model-based development and formal verification for adaptive embedded system proposed in this thesis is prototypically realised in the AMOR (Automatic MOdel VerifieR) integration framework using Java. AMOR implements the translation of MARS adaptive system models and their properties into SAS models and $L_{SAS}$ properties which form a verification task. Static analysis techniques can be applied to the system model in order to establish its consistency. The verification task can be directly completed in a theorem prover by translating the system and the considered property to the input of a theorem prover. With the model transformations presented in Chapter 5 and the compositional reasoning strategies presented in Chapter 6, the complexity of the verification task is reduced within AMOR such that automatic verification techniques, i.e. model checking, can be efficiently applied. In this direction, AMOR supports the translation of SAS models and properties into the input languages of different model checkers. In this chapter, we review the main concepts used for the realisation of the AMOR integration framework and evaluate it at the development of the adaptive vehicle stability control system [ASS08] presented in Chapter 2 for verification of safety and correctness properties of functional and adaptation behaviour.
7.1 AMOR Integration Framework

The architecture of the AMOR integration framework depicted in Figure 7.1 is centred around a verification task. A verification task consists of an SAS system model and an $L_{SAS}$ property to be verified over the model. The components of AMOR can be classified in two dimensions, first, translations from and to the SAS model and the property in the verification task and, second, transformations for verification complexity reduction on the verification task itself. The component-based system architecture facilitates using different frontends providing input both for systems and properties and different verification backends, e.g. theorem provers and model checkers. Furthermore, model transformations and compositional reasoning strategies can be adapted such that they match the requirements of a concrete verification task most appropriately. Additionally, the architecture leaves space for extensions of verification complexity reduction techniques. Furthermore, it smoothly integrates translation validation of model transformations (cf. Section 5.1.2) as SAS models before and after a transformation can be translated into a representation inside a theorem prover.

The AMOR integration framework has been developed in the context of the EVAS project (Verification of Adaptive Embedded Systems). The EVAS project is concerned with the development of new methods for modelling, analysis and model-based synthesis of embedded systems that support dynamic adaptation. Figure 7.2 shows the proposed development process for
adaptive systems [ASS08]. The MARS modelling concepts have been integrated into the Generic Modelling Environment GME [LMB+01] by providing a GME meta-model. GME automatically produces a model representation in XML format, which can be used as input for validation and verification as well as for code generation. In this direction, first simulation of a model's adaptation behavior and visualization of reconfiguration sequences using adaptation sequence charts [TAFJ07] is supported. Second, it is possible to perform a probabilistic analysis of the adaptation behavior [AFT07] computing the probability that a configuration of a module is activated from the failure rates of sensors and actuators. Third, AMOR uses the XML output produced by GME as input language for system models. Properties over models are directly entered into AMOR. In the EVAS project, the XML representation of system models is also used for translating them into a representation in MATLAB-Simulink\(^1\), the de-facto standard in industrial development of embedded systems. In Simulink, functionality of the configurations can be actually implemented such that code generation for embedded system controllers using Simulink's Real Time Workshop becomes possible. If the implemented functionality in a configuration complies to the functional specifications given in the MARS model, the model properties verified with AMOR also hold for the generated code.

As model checking backend in the EVAS project, we use Averest, a framework for the specification, implementation, and verification of reactive systems [SS05]. In Averest, a system is defined in the synchronous programming language Quartz [Sch08], which is well-suited for describing

\(^1\) http://www.mathworks.com
adaptive systems obtained from SAS models, as both are based on a synchronous semantics. Specifications can be given in temporal logics, such as CTL and LTL to which $\mathcal{L}_{\text{SAS}}$ properties can be translated. In this thesis, we additionally use NuSMV [CCGR00] as a second model checking backend in order to compare the proposed verification complexity reduction techniques for different model checkers. NuSMV and Averest are both symbolic BDD-based model checkers such that the experimental results refer to the same underlying model checking technique. For direct verification of properties and for translation validation of model transformations, we use the interactive theorem proving environment Isabelle/HOL [NPW02].

7.1.1 Internal Representation of Verification Tasks

The core of the AMOR integration framework is constituted by a verification task. It consists of an SAS system and a property to be verified. Furthermore, it contains all information about applied model transformations and compositional reductions necessary for the generation of verification output. SAS systems and properties are represented in AMOR by immutable term data types generated by the Katja [Mic06] framework. Katja (Kaiserslautern Attribution System for Java) comprises a small and lightweight specification language for order-sorted recursive term data types for Java. It supports the definition of tuple, list and variants types in order to express the components of abstract syntax trees. With these constructs, the abstract syntax of SAS system models and $\mathcal{L}_{\text{SAS}}$ properties can be defined. Figure 7.3 shows the Katja specification for the abstract syntax of an SAS module which closely corresponds to the definition of SAS modules in Definition 3.12. A Module is a tuple consisting of a name, a list of input variables invars, a list of output variables outvars, a list of local variables locvars, their initialisation init, a list of configurations confs and the adaptation aspect adapt. The automatically generated data types can be instantiated in Java to represent concrete SAS models and properties. Additionally, generated operators on these data types, such as visitors and iterators, offer a convenient way for analysis and manipulation.

7.1.2 Translations from and to Verification Tasks

The component-based architecture of the AMOR integration framework allows using different frontends to provide input for SAS models and properties in a verification task. Likewise, different verification tools, e.g. theorem
7.1. AMOR Integration Framework

Module \(\text{Name name, VarList invars, VarList outvars, VarList locvars, VarMaps init, Configurations confs, Adaptation adapt}\)

Configurations \(*\ Configuration

Configuration \(\text{Name confname, Priority priority, Formula guard, FuncDescr next_state, FuncDescr next_out}\)

Adaptation \(\text{VarList adapt_ins, VarList adapt_outs, VarList adapt_locs, VarMaps adapt_init, FuncDescr adapt_next_state, FuncDescr adapt_next_out}\)

Fig. 7.3. SAS Module Specification in Katja

provers and model checkers, can be used as backends to complete a verification task. This allows exploiting the strengths of certain tools with respect to specific verification goals. While in the EVAS project frontends and backends for model input and verification output have been fixed, the integration of further modelling environments and verification tools can easily be achieved by providing a translation from or to an AMOR verification task.

**Translating Models in GME-XML to SAS Models**

In the EVAS project, MARS models are developed in the Generic Modelling Environment (GME) [LMB+01]. GME produces an output of the specified models in a generic XML format. This XML output is used as input for AMOR by translating the XML model description to SAS models along the lines of Section 3.1.3. Figure 7.4 shows a part of the XML description produced by GME for the model of the module \(\text{V.YawCalculation}\). For instance, the XML tag `<model [...] kind="Module" [...] > denotes a MARS module. The inner XML nodes describe the parts of a module, e.g. the tag `<model [...] kind="InPort" [...] > defines an input port and the tag `<model [...] kind="Configuration" [...] > defines a configuration.

**Translating Verification Tasks to Quartz Programs**

In the Averest framework [SS05], a system is described in the synchronous programming language Quartz [Sch08]. Quartz offers a module structure where program modules can be synchronously composed in a designated module called `main`. Synchronisation of modules is taken care of by the concept of micro and macro steps. From a programmer’s view, micro steps
do not take time while macro steps defined by the keyword `pause` consume one unit of time. At the end of each macro step, the modules synchronously composed in the main module explicitly synchronise.

A Quartz program corresponding to an SAS system in a verification task realises each SAS module as a Quartz module. In Figure 7.5, the Quartz program for the Module `V_YawCalculation` is shown. Input and output variables of an SAS module are translated to Quartz module input and output variables, respectively. Local variables are mapped to Quartz local module variables. Adaptive and functional variables are no longer distinguished. Values used in SAS models are translated to Quartz data types. For integers and Boolean values, the corresponding Quartz data types are used. String constants in SAS models, e.g. configuration names, are encoded as constants of bitvectors with finite length `nat[n]`. Values that cannot be
represented in this way have to be abstracted in order to generate a Quartz program for the module.

At the beginning of each module program in Quartz, the variables are initialised in a distinct macro step. In the example of the module V_YawCalculation, the variable capturing the used configuration in each cycle useconf is initialised with Off. The actual execution of a module is described by a loop comprising one macro step. This loop contains a switch statement in which each case corresponds to one configuration of the SAS module. The condition of each switch case is the configuration guard of the respective configuration. The switch cases are sorted with increasing configuration priority. In each switch case, the useconf variable is assigned to the name of the used configuration in order to reason about reconfiguration sequences. Furthermore, the SAS function descriptions of next state and next output functions of each configuration are translated to Quartz.
assignments in the respective switch case. Using the `next` operator, assignments come into effect in the next macro step. In order to obtain more efficient program representations, the adaptive next state and adaptive next output functions are multiplied out into the switch cases of the single configurations. For instance in the yaw rate module, the quality attached to the module output `v.yaw.15.quality` in configuration `Measured` is assigned to the constant value `measured`.

The Quartz modules corresponding to SAS modules are synchronously composed in the `main` module representing an SAS system. The main module’s interface comprises the system input and output variables. The body of the `main` module contains one macro step in which all modules are synchronously executed. Input and output parameters of the modules are instantiated according to the system connections. For non-deterministic SAS modules, we use the non-deterministic choice operator `choose` of Quartz in order to deal with the set of possible successor states. Furthermore, when a system or a module has been abstracted, Quartz constants representing the abstract values and Quartz macros capturing the interpretation of predicates and operators over the abstract values are generated.

Properties to be verified over a Quartz program are given in a distinct specification module `spec` in temporal logics, such as CTL or LTL, or in the $\mu$-calculus. Hence, besides the Quartz program representing the system in a verification task, a specification module capturing the property to be verified is generated. This specification module contains the representation of the $\mathcal{L}_{SAS}$ property in Quartz syntax. At the end of Figure 7.5, we see a specification module for the first generic adaptive property of the module `V.YawCalculation` requiring that it invariantly holds in the system ($\text{AG}$) that there exists a path ($\exists$) such that the module can eventually ($\text{F}$) leave the shutdown configuration `Off`. Fairness constraints $\varphi$ of maximal models are encoded in Quartz by a fairness assumption $\text{GF}\varphi$ added to the property to be verified. The formula $\text{GF}\varphi$ denotes that $\varphi$ holds infinitely often on a path.

**Translating Verification Tasks to NuSMV Programs**

NuSMV [CCGR00] is a new implementation and extension of the model checker SMV [McM93], the first symbolic model checker based on Boolean decision diagrams (BDD) for an efficient model representation. NuSMV allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in computation tree logic (CTL) and linear time temporal logic (LTL). A NuSMV finite
7.1. AMOR Integration Framework

State machine consists of a set of states as valuations of state variables, initial states and state transitions. State machines can be defined modularly by combining finite state machines for modules in a main module by synchronous or asynchronous composition. As NuSMV is intended for describing finite state machines, it only comprises finite data types, such as Booleans, scalars and arrays.

```plaintext
MODULE v_yaw(yaw_rate_measured_15_quality, [...])
VAR v_yaw_15_quality : {unavailable, [...], measured};
VAR uconf :{ Measured, [...], Off};

DEFINE CONF5 := yaw_rate_measured_15_quality = available;
DEFINE CONF4 := [...];
DEFINE CONF3 := [...];
DEFINE CONF2 := [...];
DEFINE CONF1 := [...];
DEFINE CONF0 := !(CONF1 | CONF2 | CONF3 | CONF4 | CONF5)

ASSIGN
  uconf := case
    CONF5 : Measured;
    CONF4 : DerivedFromSteeringAngle;
    CONF3 : DerivedFromV_FWheel;
    CONF2 : DerivedFromV_RWheel;
    CONF1 : DerivedFromAY;
    CONF0 : Off;
  esac;

init(v_yaw_15_quality) := unavailable;
next(v_yaw_15_quality) := case
  CONF5 : measured;
  CONF4 : steering_angle;
  CONF3 : v_wheelF;
  CONF2 : v_wheelB;
  CONF1 : ay;
  CONF0 : unavailable;
  esac;

SPEC AG (( module_v_yaw.uconf = Off ) -> EF (module_v_yaw.uconf != Off ) )
```

Fig. 7.6. NuSMV Program for Module V_YawCalculation

A NuSMV specification for an SAS verification task represents each SAS module in the system as a NuSMV module. Figure 7.6 depicts the NuSMV program for the module V_YawCalculation. In a NuSMV program, functional and adaptive variables are no longer distinguished. The input variables of a module are represented by the formal parameters of a mod-
ule, e.g. yaw_rate_measured_15_quality. Local and output variables are translated to state variables of a module using the keyword VAR. Additionally, a state variable uconf is contained in every module for capturing the used configuration. Variables are declared as enumeration types containing all string constants a variable assumes. As infinite data types, such as integers or real numbers, cannot be represented, data domain abstraction has to be applied before generation of NuSMV output if a system contains integer variables. The configuration guards of a module are represented by DEFINE statements in order to keep the program concise. DEFINE statements serve as macro definitions that are syntactically replaced when the model is instantiated from the program. Initial values for state variables are assigned with an init assignment. Using a next assignment, the value of a variable in the following state is set. Each state variable is assigned in a case expression where the first expression with a valid constraint is selected. For SAS modules, the different cases of a case expression reflect the configuration guards of the module in order of their priorities such that the assignment for the active configuration is chosen. In the example, if configuration Measured represented by the defined constant CONF5 is activated, the module output v_yaw_15_quality is assigned to measured.

Non-deterministic SAS modules are represented by INIT and TRANS constructs. After INIT, a propositional condition on state variables defines the set of initial states as states that satisfy the condition. TRANS defines a transition relation over states by a propositional condition on the current and the next values of the state variables. Fairness constraints of SAS modules can be expressed explicitly using the keyword FAIRNESS. A fairness constraint F can simply be stated as FAIRNESS F.

An SAS system is represented by the main module. The system input and output variables are mapped to state variables, where input variables remain uninitialised such that they can assume any value. Modules are instantiated in the main module using synchronous composition. System connections are reflected by the choice of the actual parameters of the modules. A module input connected to a system input uses the system variable as actual parameter. A module input connected to a module output takes the module output variable as actual parameter. The system output variables are assigned to the connected module output variables at the end of the main module.

Properties are expressed in a NuSMV program as specifications using CTL or LTL formulae such that properties from the linear time or
the branching time fragment of $L_{SAS}$ can be analysed. At the end of Figure 7.6, we see the first generic adaptive property for the module $V_{YawCalculation}$ as a NuSMV specification with the keyword SPEC. It states that the module does not stay in the shutdown configuration Off forever where the used configuration in the module is referenced by the expression $\text{module}_v\text{.yaw}\text{.unconf}$. As NuSMV is restricted to checking CTL or LTL specifications, for assume-guarantee reasoning in the linear-branching case maximal model generation for the assumption is indispensible because the formula $\varphi \rightarrow \psi$ is neither expressible in LTL nor in CTL.

Comparing the capabilities of the Averest model checker and the NuSMV model checker, it can be observed that Averest provides more high-level programming constructs such as loops or integer variables for describing synchronous system models. NuSMV, in contrast, only allows specifying finite state machines by means of state variables over finite domains and their initial states and transitions. Furthermore, Averest allows analysis of general $\mu$-calculus formulae, which are more expressive than CTL and LTL formulae, which can be checked with NuSMV. In Section 7.2.2, we further compared the performance of Averest and NuSMV regarding the verification of adaptation properties of the vehicle stability control system. The evaluation showed that model reduction is more important for NuSMV models in order to facilitate efficient verification since Averest internally employs sophisticated reductions exploiting properties of synchronous systems.

**Translating Verification Tasks to Isabelle/HOL Theories**

The translation of SAS system models and properties to Isabelle/HOL is done according to the principles described in Section 4.2. This translation is used both for direct verification of SAS model properties using theorem proving as well as for translation validation of model transformations. For translation validation, the SAS model and the property before and after the transformation are translated into an Isabelle/HOL representation. A proof script certifying that there exists a consistent simulation between the two systems is generated in order to prove the correctness of the transformation.

**7.1.3 Transformations on Verification Tasks**

The model transformations presented in Chapter 5 and the compositional reasoning strategies presented in Chapter 6 are implemented as compo-
nents in the AMOR integration framework. This allows adapting the tech-
niques and strategies for complexity reduction to a concrete verification 
task. Furthermore, the framework can easily be extended with further 
techniques to ease formal verification without changing the remaining sys-
tem parts.

**Compositional Reasoning Strategies**

In order to use compositional reasoning strategies in a verification task, a 
possible decomposition of the system into the modules referred to by the 
property is computed. Then, the user can decide whether to use decompo-
sition, assume-guarantee reasoning or interface abstraction to simplify the 
verification task. For decomposition (cf. Section 6.2.1), a reduced system 
containing only the affected modules is stored in the verification task. If 
only one module is referred to by the property, this module is stored as a 
valid reduction of the system. If further the considered property is a purely 
functional invariant over a single module and the module does not have 
a functional local state, the compositional proof rule for functional proper-
ties (cf. Section 6.2.4) is applicable. Then, a module for each configuration 
is generated that only contains the respective functional behaviour. For 
assume-guarantee reasoning (cf. Section 6.2.2), the user has to provide an 
assumption on the behaviour of the removed system parts. This assump-
tion is entered in the same way as the property to be verified and kept 
in the verification task besides the computed decomposition. For interface 
abstraction (cf. Section 6.2.3), the set of variables at the interface of the 
system part not in the decomposition is computed. Then, this system part 
is abstracted with respect to the behaviour at the interface and stored in 
the verification task. In the present implementation, we use module slicing 
with respect to the set of variables at the interface. However, the reduction 
can be replaced by more elaborate abstraction techniques as long as the 
behaviour observable at the interface remains unchanged.

An assume-guarantee reasoning task or checking the validity of a con-
troller property under an assumption on the controlled system can be con-
verted to a standard model-checking task. To this end, the maximal model 
construction by tableau for an assumption described in Section 6.2.2 can 
be used in AMOR. The maximal model for an assumption is constructed di-
rectly before verification output is generated such that the construction can 
profit from model transformations, in particular abstractions, carried out 
after the application of compositional reasoning strategies. Alternatively,
the assumption can be used as additional precondition for the property to be verified if the model checker can handle the resulting property.

**Slicing**

AMOR implements the slicing techniques presented in Section 5.2 with respect to the property in the verification task. For adaptive slicing, it is checked that the property is purely adaptive over the system. System slicing can be carried out on a system containing several modules. Module slicing is only applied to fully specified SAS systems and modules. If underspecified functionality is contained in a module, module slicing is not possible. The same holds for non-deterministic SAS modules. Those can only be handled by system slicing.

**Data Domain Abstractions**

AMOR implements a set of pre-computed data domain abstractions on integers, e.g. sign abstraction, in an abstraction library as proposed in Section 5.3.4. This supports the user to chose the appropriate abstraction for a verification task. An abstraction $\text{Abs} = (\text{Val}, \alpha, \text{Rel}, \text{BinOp})$ in the abstraction library contains the abstract domain $\text{Val}$, the abstraction mapping $\alpha$ and abstract interpretations for predicates $\text{Rel}$ and operations $\text{BinOp}$. The abstraction library can easily be extended with abstractions required for particular verification tasks.

For applying a data domain abstraction to a verification task, the abstraction to be carried out is selected from the abstraction library and stored in the verification task. This is necessary in order to generate the correct interpretations of the abstract predicates and operations in the verification output. Additionally, the abstractions of the considered system or module and of the property are stored in the verification task. A syntactic system abstraction is computed automatically along the lines of Section 5.3.2. Note, that syntactic data domain abstraction is restricted to modules where functionality is given by SAS function descriptions. The considered property in the verification task is abstracted according to the principles described in Section 5.3.3. Each concrete atomic proposition is replaced by a disjunction of equalities between a variable and an abstract value whose concretisations imply the concrete atomic proposition. In the current implementation, the abstraction of a property has to be provided manually. But property abstraction can be automatised using a decision procedure checking implications between concrete and abstract atomic propositions (cf. [GS97]).
7.1.4 AMOR User Interface

In order to ease the use of the AMOR integration framework, a graphical configuration wizard is provided guiding through the verification complexity reduction process until verification input is generated. First, the user enters the XML file containing the representation of the MARS model and the $L_{SAS}$ property to be verified (cf. Figure 7.7). Additionally, the user can select static analyses to be performed on the input model, such as consistency checks (cf. Section 4.1.1) or environment model analysis (cf. Section 4.1.2). Clicking the Next button, the MARS model is read from the XML file and translated to the corresponding SAS representation along the lines of Section 3.1.3. The property is translated to a term representation in Katja data types. System and property instantiate the current verification task on which the selected static analyses are performed.

According to the procedure for composition of reduction building blocks described in Figure 6.12, in a second step compositional reasoning strategies can be applied to the verification task in order to reduce its verification complexity (cf. Figure 7.8). To this end, it is checked that the property in the verification task is a universal property over a part of the system. If applicable, the user can select whether to use decomposition, assume-
guarantee reasoning, interface abstraction or the functional decomposition rule. For assume-guarantee reasoning, another property serving as assumption has to be entered. Additionally, it has to be determined if the assumption should be translated to a maximal model when verification output is generated. Depending on the selection of the compositional reasoning strategy, the verification task is transformed in order to capture system or module decompositions, an assumption or an interface abstraction. At each step, the AMOR configuration wizard allows the user to display the current verification task in order to track the performed actions. Furthermore, output to the Isabelle/HOL theorem prover can be generated for translation validation of the transformations. The AMOR configuration wizard also allows the user to skip any of the reduction steps and to compute verification output for the original or a partially reduced system directly.

Third, the user can select the slicing techniques of the desired granularity to apply to the current verification task (cf. Figure 7.9). Furthermore, a data domain abstraction to be carried out after slicing can be picked from the abstraction library. In the current implementation, the system or module in the verification task is abstracted automatically, while the abstract property as well as an abstract assumption have to be entered manually.
Finally, the user chooses the verification input she wants to generate (cf. Figure 7.10). In the current implementation, this can be input for the model checkers Averest and NuSMV or the theorem prover Isabelle/HOL.

7.2 Verification of Adaptive Vehicle Stability Control

The integration of model-based development and formal verification for adaptive embedded systems has been evaluated together with the development of the adaptive vehicle stability control system [ASS08] presented in Chapter 2. Using the AMOR integration framework, we have verified properties of functional, adaptation and combined behaviour of the system model. The properties covered application-specific as well as generic aspects. In this section, we first demonstrate how compositional reasoning strategies and model transformations facilitate verification of functional safety and correctness properties of the vehicle stability control system. Second, we provide empirical data that slicing and decomposition dramatically improve the efficiency of verification of adaptive properties, before we conclude with general results of the evaluation of our integration framework in the case example.
7.2. Verification of Adaptive Vehicle Stability Control

Fig. 7.10. User Interface Step 4: Generate Verification Input

7.2.1 Verification of Functional Behaviour

Compositional reasoning strategies and model transformations provided by the AMOR framework significantly reduce the complexity for verifying functional safety and correctness properties of the rear brake and gas servo module *RearBrakeGasServo* and of the core controller modules *SteeringAngleDelimiter*, *TractionControl* and *YawRateCorrector* in the vehicle stability control system. Data domain abstractions make verification tasks amenable to model checking. Different verification tools, i.e. theorem provers and model checkers, contribute to the completion of the same verification task in the case example. Functionality of module configurations in the vehicle stability control system model is given by pre/post-condition specifications. In most cases, the specifications could be translated to SAS function descriptions enabling the use of model checking. If it was not possible to encode the specification in an SAS function description, functional properties could still be handled with interactive theorem proving. Some functional properties depend on the physics of the car. These characteristics of the controlled system, i.e. the car, are out of the scope of the considered controller model and are dealt with by additional environment assumptions (cf. Section 6.2.5).
Rear Brake Gas Servo

The remote controlled car on which the vehicle stability control system is implemented only possesses one servo for brake and gas at its rear wheels. Also it only has one way to input the desired brake or gas value by the driver. Here, a positive value is interpreted as a gas value and a negative as a brake value. In the original layout of the car, the brake or gas value is directly transferred to the servo at the rear wheels such that a conflict between brake and gas values at the rear wheels cannot occur.

In the vehicle stability control system implemented on top of the remote controlled car’s hardware, however, the direct coupling is broken up in order to provide traction control and yaw rate correction functionalities. Figure 7.11 shows the extract of the vehicle stability control system dealing with the propagation of the brake and gas input to the respective servo for the rear wheels. The traction control module delimits the gas value in order to prevent a loss of traction at driveaway or if the wheel slip exceeds a threshold. The yaw rate corrector module influences the brakes in order to restrict the yaw rate of the car if the driver wants to go straight out.

In order to provide these two functionalities, two sensor modules Brakes and Gas are introduced that split the input brakeGas_input into a brake and a gas value. If the brake and gas input is greater than zero, the gas module sets its gas output to the brakeGas_input value, else the gas value is set to zero. If the brake and gas value is smaller or equal to zero, the brake module sets the brake value to the negation of the input brake and gas value brakeGas_input in order to obtain a positive brake value. Otherwise, the brake value is set to zero. The gas value is transferred to the traction control system which computes a delimited_gas value. The brake value is conveyed to the yaw rate corrector which adjusts the brake value to a corrected_rear_brake value. Both, corrected brake and delimited gas values, are inputs to the logical actuator module RearBrakeGasServo. This

![Diagram](image_url)

**Fig. 7.11.** Traction Control and Yaw Rate Correction in Vehicle Stability Control System
module sets the output `rear_brake_cntrl` for the physical brake gas servo of the rear wheels. If the delimited gas value is greater than zero, the controlled brake and gas value `rear_brake_cntrl` is assigned to the delimited gas value. If the corrected brake value is greater or equal to zero, the controlled brake and gas value `rear_brake_cntrl` is set to the negative value of the corrected brake value. This maintains the characteristics of the remote control car’s hardware that a positive brake and gas value corresponds to a gas value while a negative value denotes a brake value. Figure 7.12 shows the functionality of the configuration `On` of the `RearBrakeGasServo` module in pseudo code notation of the respective SAS function description. The configuration `On` is the only configuration of this module besides the shutdown configuration `Off`.

An important correctness property of the `RearBrakeGasServo` module is that if the module is not in the shutdown configuration `Off`, then if the gas value is positive, the gas value is assigned to the output, and if the brake value is positive, the negative of the brake value is assigned to the output. This can be described by the following $\mathcal{L}_{\text{SAS}}$ property (CorrectRearBrakeGasServo).

$$(\text{CorrectRearBrakeGasServo}) \equiv AG ((useconf_{\text{RearBrakeGasServo}} \neq \text{Off}) \rightarrow (\text{delimited_gas} > 0 \rightarrow \text{rear_brake_cntrl} = \text{delimited_gas}) \land (\text{corrected_rear_brake} > 0 \rightarrow \text{rear_brake_cntrl} = -\text{corrected_rear_brake}))$$

(CorrectRearBrakeGasServo) is an universal property that only refers to the `RearBrakeGasServo` module. In order to establish it for the vehicle stability control system, the module decomposition rule in Theorem 6.28 is applicable. If we can show that the property holds over the `RearBrakeGasServo` module alone, it can be lifted to the overall vehicle stability control system. As brakes and gas values as well as their delimited, corrected and controlled counterparts are modelled as integer values, the module cannot be model checked directly as it is infinite-state. With a sign abstraction as described in Section 5.3.2 of the integer val-
ues, the module can be reduced to finite-state while it is still possible to express the correctness property precisely. As abstract domain, we chose the set $\hat{\text{Val}} = \{\text{pos}, \text{neg}, \text{zero}\}$. For the abstraction, we map positive integers to $\text{pos}$, negative to $\text{neg}$ and $0$ to zero. Figure 7.13 shows the functionality of the configuration $\text{On}$ abstracted by sign abstraction. The operator $-$ is interpreted in the abstract domain as $-\text{pos} = \text{neg}$, $-\text{neg} = \text{pos}$ and $-\text{zero} = \text{zero}$. As in this example only input and output values are abstracted, we do not have to consider the power set domain such that the module behaviour remains deterministic. The correctness property is abstracted to $(\text{CorrectRearBrakeGasServoAbs})$.

$(\text{CorrectRearBrakeGasServoAbs}) \equiv (\text{useconf}_{\text{RearBrakeGasServo}} \neq \text{Off}) \rightarrow \begin{align*}
\text{BrakeGas} & \quad (\text{delimited_gas} = \text{pos} \rightarrow \text{rear_brake_cntrl} = \text{delimited_gas}) \\
\wedge (\text{corrected_rear_brake} = \text{pos} \\
& \rightarrow \text{rear_brake_cntrl} = -\text{corrected_rear_brake}))
\end{align*}$

By property preservation under data domain abstraction in Section 5.3, we know that if we can establish $(\text{CorrectRearBrakeGasServoAbs})$ over the abstract system the property $(\text{CorrectRearBrakeGasServo})$ also holds in the concrete system. However, translating the abstract version of the $\text{RearBrakeGasServo}$ module and the property $(\text{CorrectRearBrakeGasServoAbs})$ to the input of a model checker and running the model checker on the system gives the result that the property is not true. The reason is that by decomposition the module is embedded into an chaotic environment which can exhibit any possible behaviour. This also includes computation paths where the gas and the brake input to the $\text{RearBrakeGasServo}$ module is positive at the same time. On these paths, the property $(\text{CorrectRearBrakeGasServoAbs})$ does not hold. Hence, a decomposition is a too coarse abstraction of the module behaviour in this example.

In order to establish the correctness property, we have to add an assumption on the remaining system that the inputs to the rear brake and gas servo module are mutually exclusive. This means that if the $\text{delim-}

```plaintext
if (delimited_gas == pos) then
    rear_brake_cntrl := delimited_gas

if (corrected_brake == pos or corrected_brake == zero) then
    rear_brake_cntrl := -corrected_rear_brake
```

Fig. 7.13. Abstracted Functionality of $\text{RearBrakeGasServo}$ Module in Configuration $\text{On}$
7.2. Verification of Adaptive Vehicle Stability Control

The gas value is greater than zero, the corrected rear brake value is equal to zero, and that if the corrected rear brake value is greater than zero, the delimited gas value is equal to zero. This assumption can be expressed in $L_{SAS}$ by the property (BrakeGasExclusion).

$$\neg \neg \text{BrakeGasExclusion} \land \neg \neg \text{CorrectRearBrakeGasServo}$$

Using the assume-guarantee reasoning rule in Theorem 6.32, we know that if we can prove the guarantee (CorrectRearBrakeGasServo) under the assumption (BrakeGasExclusion) and if we can establish (BrakeGasExclusion) over the remaining system, the guarantee is also true for the complete system without the assumption. In order to use model checking for this assume-guarantee proof, we have to use the finite-state abstraction of the module. To this end, we also have to abstract the assumption to (BrakeGasExclusionAbs).

For showing the abstract assume-guarantee pair over the abstract RearBrakeGasServo module, we have two possibilities. As the assumption (BrakeGasExclusionAbs) is a linear property, assume-guarantee checking can first be reduced to checking the implication

$$\neg \neg \text{BrakeGasExclusionAbs} \land \neg \neg \text{CorrectRearBrakeGasServoAbs}$$

Second, we can construct a maximal compatible model for the assumption and show the guarantee over the system constructed by composing the maximal compatible model for the assumption (BrakeGasExclusionAbs) and the abstract RearBrakeGasServo module. Figure 7.14 shows the reduced tableau for the abstract assumption (BrakeGasExclusionAbs) constructed according to the algorithm presented in Section 6.2.2. Note that all states are fair and initial. The set of fair computation paths of the structure corresponds to all paths satisfying the abstract assumption. The tableau structure can be encoded in an SAS module, combined with the abstract RearBrakeGasServo module and model checked with respect to the guarantee property.

Now, we obtain the result that the correctness property (CorrectRearBrakeGasServoAbs) is true under the assumption.
(BrakeGasExclusionAbs). Using the results on property preservation of universal properties under data domain abstraction in Section 5.3, we know that the assume-guarantee pair (AGR)

\((\text{BrakeGasExclusion})\) \(m_{\text{RearBrakeGasServo}} ((\text{CorrectRearBrakeGasServo}))\)

is true. Instead of assume-guarantee reasoning, interface abstraction with respect to the interface behaviour of the RearBrakeGasServo module can also be used (cf. Section 6.2.3).

It remains to show that the remaining system, in particular the modules Brakes, Gas, YawRateCorrection and TractionControl, satisfy the assumption (BrakeGasExclusion). The property (BrakeGasExclusion) is implied by a conjunction of three universal properties depicted in Figure 7.15. First, we require that the input value brakeGasInput is correctly split by the brake and gas modules (Property1) if both are not in their shutdown configuration Off. Second, the sign of the input gas value has to be preserved by the traction control module (Property2), i.e. if the module is not in the shutdown configuration and the input gas value is greater than zero, the output delimited_gas value is also greater than zero. If the input gas is equal to zero, also the output must be equal to zero. The same has to hold for the YawRate Corrector module with respect to the brake and the corrected_rear_brake values (Property3).

As all three properties are universal, we can apply the conjunctive decomposition rule in Theorem 6.25 and the conjunctive decomposition rule for modules in Corollary 6.29 in order to show each property in isolation. (Property1) over the brake and gas modules has already been established using theorem proving in Section 4.2. For (Property2) and (Property3), we have to consider the traction control and the yaw rate corrector module, respectively. Both properties can be established over a sign abstraction of the respective modules using model checking.
7.2. Verification of Adaptive Vehicle Stability Control

\[
\begin{align*}
\text{(Property 1)} & \quad \text{AG} \left( (\text{useconf}_{\text{Brake}} \neq \text{Off} \land \text{useconf}_{\text{Gas}} \neq \text{Off}) \implies \text{brake} \geq 0 \rightarrow \text{gas} = 0 \land (\text{gas} > 0 \rightarrow \text{brake} = 0) \right) \\
\text{(Property 2)} & \quad \text{AG} \left( (\text{useconf}_{\text{TractionControl}} \neq \text{Off}) \implies \text{gas} > 0 \rightarrow \text{delimited~gas} > 0 \land (\text{gas} = 0 \rightarrow \text{delimited~gas} = 0) \right) \\
\text{(Property 3)} & \quad \text{AG} \left( (\text{useconf}_{\text{YawRateCorrector}} \neq \text{Off}) \implies \text{brake} > 0 \rightarrow \text{corrected~rear~brake} > 0 \land \text{brake} = 0 \rightarrow \text{corrected~rear~brake} = 0 \right) \\
\text{(Property 1)} \land (\text{Property 2}) \land (\text{Property 3}) & \rightarrow \text{(BrakeGasExclusion)}
\end{align*}
\]

**Fig. 7.15. Requirements for (BrakeGasExclusion)**

From the validity of the three properties, we can deduce that the assumption (BrakeGasExclusion) is true in the vehicle stability control system. This allows discharging the assumption in the assume-guarantee pair (AGR). Hence, by assume-guarantee reasoning, we can conclude that the correctness property (CorrectRearBrakeGasServo) is true in the vehicle stability control system.

Using assume-guarantee reasoning and decomposition as compositional reasoning strategies, the overall verification task over the 20 system modules in the vehicle stability control system could be split into four separate verification tasks over one or two modules, respectively. This reduced the complexity for verification enormously. Sign abstractions of integer variables allowed model checking of the sub-verification tasks. The verification of (Property 1) by theorem proving over the non-abstracted brakes and gas modules documents that different verification tools can be used in the same global verification task.

**Steering AngleDelimiter**

The steering angle delimiter in the vehicle stability control system controls the steering angle at the steering angle servo. The input steering angle coming from the driver is restricted in case it exceeds the maximal possible steering angle at the current speed of the car. Otherwise, the car might enter an unstable driving situation. If the input steering angle is admissible, it is simply transferred to the steering angle servo. The steering angle delimiter module has two configurations besides the shutdown configuration: the configuration Controlled in which the steering angle is restricted depending on the current speed, and the configuration Forwarded in which
the input is simply forwarded to the output in order to compensate for a
failure of the vehicle speed input.

A safety property of the steering angle delimiter module is that the output
\texttt{delimited\_steering\_angle} never exceeds the input steering angle \texttt{steering\_angle\_Input} if the module is not in its shutdown configuration \texttt{Off}. This
can be expressed by the $L_{SAS}$ property already mentioned in Section 3.2.2.
A correctness property of the configuration \texttt{Controlled} is that the output
\texttt{delimited\_steering\_angle} is in fact smaller than the maximal admissible
steering angle at the current speed $maxAngle(v_{\text{carRef}})$. The correctness
property can be expressed in $L_{SAS}$ as

$$(\text{SteeringAngle} \ AG (useconf_{\text{SteeringAngleDelimiter}} = \text{Controlled} \rightarrow
\text{DelimiterCorrect}) ) \ delimited\_\text{steering}\_angle \leq \text{maxAngle}(v_{\text{carRef}}))$$

Both safety and correctness properties are universal properties only re-
ferring to the variables of the steering delimiter module. Hence, the mod-
ule decomposition rule in Corollary 6.29 allows lifting the validity of both
properties over the steering angle delimiter module to the overall vehicle
stability control system. As the functionality of the steering angle delim-
iter module uses an undefined function $maxAngle(v_{\text{carRef}})$, it cannot be
expressed as an SAS function description. But as demonstrated in Section
4.2.2, the functionality can be specified in the input language of a theorem
prover (cf. Figure 4.9). Hence, both properties can be established using the-
orem proving as already done in Section 4.2.2.

\textit{Traction Control System}

The traction control module in the vehicle stability control system is in
charge of restricting the gas in order to prevent loss of traction at drive-
away or if the slip at the wheels exceeds a threshold. Besides the shutdown
configuration \texttt{Off}, it implements a fully functional configuration \texttt{On} and a
fail-safe configuration \texttt{SlowStart}. In the configuration \texttt{On} (cf. Figure 7.16),

```plaintext
if ( ( slipRL > 15\% \ or \ slipRR > 15\% ) )
    then delimited\_gas := old\_gas \ast \ 0.7;

else if ( ( old\_gas + max\_gas ) < gas )
    then delimited\_gas := old\_gas + 1/2 \ast max\_gas;

else delimited\_gas := gas;
```

\textbf{Fig. 7.16.} Functionality of Traction Control Module in Configuration \texttt{On}
the output gas value `delimited_gas` is set according to the slip of the wheels. If the slip of one of the rear wheels exceeds a threshold (in the example 15%), the output `delimited_gas` is set to 70% of the last output value that is stored in the local module variable `old_gas`. If the slip is smaller than the threshold, it is checked whether the input gas value exceeds the last output gas value by the maximal admissible increase. The maximal admissible gas increase is modelled by a positive constant `max_gas` that depends on the actual physics of the car. In this case, the output `delimited_gas` is set to the old gas value plus half of the maximal admissible gas value. Otherwise, the input is simply transferred to the output. The configuration `SlowStart` is used if the current wheel slip is not available. It only implements the two last cases of the functionality in configuration `On`. The output `delimited_gas` in the configuration `SlowStart` is either set to the old gas value plus half of the maximal possible gas increase if the gas increase is greater than admissible, or it is set to the input gas value `gas`.

A safety property of the traction control module is that if the module is not in its shutdown configuration, the output `delimited_gas` is always smaller or equal to the input gas value `gas`. This can be expressed as \(L_{SAS}\) property (`TractionControlSafety`):

\[
AG(\text{useconf}_{\text{TractionControl}} \neq \text{Off} \rightarrow \text{delimited\_gas} \leq \text{gas})
\]

A combined safety and correctness property of the traction control module is that the output gas value `delimited_gas` is always smaller or equal to the maximal admissible gas increase `max_gas` as long as the module is not in its shutdown configuration `Off`. This safety and correctness property can be denoted as \(L_{SAS}\) property (`TractionControlCorrect1`). The local variable `old_gas` stores the delimited gas value of the previous cycle.

\[
(\text{TractionControlCorrect1}) AG(\text{useconf}_{\text{TractionControl}} \neq \text{Off} \rightarrow (\text{delimited\_gas} - \text{old\_gas} \leq \text{max\_gas}))
\]

A correctness property of the configuration `On` is that if the slip of the wheels exceeds a certain threshold, in the example 15%, the current output gas value `delimited_gas` is smaller than the output gas value of the previous cycle `old_gas`. The \(L_{SAS}\) property (`TractionControlCorrect2`) formalises this correctness property. The variable `slipRR` corresponds to the slip of the right rear wheel. The variable `slipRL` corresponds to the slip of left rear wheel. Using the assumption over the environment of the controlled system, i.e. the physics of the car, that the slip of the wheels decreases if the gas value decreases, from this correctness property we can
if ( (old_gas + pos) < gas )
    then delimited_gas := old_gas + pos;
else delimited_gas := gas;

Fig. 7.17. Abstract Functionality of Traction Control Module in Configurations On and SlowStart

deduce that the traction control in configuration On indeed reduces the wheel slip.

\[
\begin{align*}
\text{(Traction AG)} & \quad (\text{useconf}_{\text{TractionControl}} = \text{On} \\
& \quad \land \text{slipRR} > 15\% \land \text{slipRL} > 15\%) \rightarrow \\
& \quad \text{Correct2}) \quad (\text{delimited}_{\text{gas}} < \text{old}_{\text{gas}}) \\
\end{align*}
\]

All three properties are universal properties over the traction control module. By the module decomposition rule in Corollary 6.29, we know that it suffices to show the properties over the traction control module alone in order to infer their validity over the vehicle stability control system. Since the definition of the functionality uses integer values and an underspecified constant for the maximal admissible gas value \(\max_{\text{gas}}\), the properties are not directly amenable to model checking. Instead, theorem proving can be used in order to establish their correctness using assumptions on the physics of the car.

(Property2) stating that the sign of the gas input is preserved in the traction control module can be verified over an abstracted version of the traction control module using model checking. We use sign abstraction mapping all positive values to \(pos\) and all zero values to \(zero\). Negative values do not occur in this module by design. We assume that \(\max_{\text{gas}}\) is a positive constant and substitute it by the abstract value \(pos\). Further, we make an assumption for the controlled system that if the input gas value \(\text{gas}\) is zero, i.e. the car brakes, the slip never exceeds the threshold. Conversely, if the slip exceeds the threshold the gas input must be positive such that also the output gas value is positive. So, we can omit the first case distinction in configuration On and obtain the abstract functionality of both configurations On and SlowStart as depicted in Figure 7.17.

Adding a positive value to a zero value gives a positive value. The addition of two positive values remains positive. Thus, in the abstract if the input gas value is zero, it can never become greater than the old gas value (either zero or positive) plus the maximal admissible increase. Hence, if the gas value is zero, it is simply propagated to the output. If the input gas value is positive, it can either be greater than the old gas value plus a positive constant or smaller. Hence, we have to consider both cases of
7.2. Verification of Adaptive Vehicle Stability Control

the if-condition non-deterministically. However, in both cases it holds that
the output is positive which yields that the abstraction of the preservation
property (AbstractProperty2) holds.

\[ \text{AG} \left( \left( \text{useconf}_{\text{TractionControl}} \neq \text{Off} \rightarrow \left( \text{gas} = \text{pos} \rightarrow \text{delimited}_\text{gas} = \text{pos} \right) \wedge \left( \text{gas} = \text{zero} \rightarrow \text{delimited}_\text{gas} = \text{zero} \right) \right) \right) \]

This result can be established by model checking of the abstract traction
control module. Since, we can show that the abstract module is consistently
similar to the concrete under the assumptions on the controlled system,
we know that also (Property2) holds over the concrete traction control
module.

**Yaw Rate Corrector**

The yaw rate corrector in the vehicle stability control system is used to kept
the yaw rate close to zero if the car is supposed to go straight out. This is a
simplified version of yaw rate correction that in principle can also be used
in cornering. But the yaw rate corrector module in the present version still
poses many challenges for verification. The implemented functionality is
as follows: In case the yaw rate is close to zero and a correction is not nec-
essary, the brake force outputs for the left front wheel \text{corrected}_\text{left_brake},
for the right front wheel \text{corrected}_\text{right_brake} and for the rear wheels \text{corrected}_\text{rear_brake} are simply set to the input brake value. If the yaw rate
has to be corrected, there are four different ways of influencing the brakes
at the front wheels. If the car turns to the left, the yaw rate is greater than
zero. Then either the break force at the right front wheel can be increased
or the brake force at the left front wheel can be decreased to get the car
to turn to the right. If the car turns to the right, the yaw rate is smaller
than zero. Then, the brake force at the left front wheel can be increased
or the brake force at the right front wheel can be decreased in order to
induce a left turn. The decision if brake force is increased or decreased is
made according to the brake input. If it is high, i.e. greater or equal to
95 \%, the brake force is decreased, else it is increased. In all cases, the rear
wheel brakes are not modified by the yaw rate corrector and assigned to
the input brake force. The yaw correction functionality is provided in the
configuration \text{YawCorrection}. Furthermore, the module has a fail-safe con-
figuration \text{Forward} where the brake force is simply forwarded to the servos
/* right turn increase left */
if ( v_yaw < 0 and delimited_steering_angle < 1 and
    delimited_steering_angle > -1 and brake < 95% ) then {
  corrected_rear_brake := brake;
  corrected_left_brake := brake + rightp;
  corrected_right_brake := brake;
}
/* right turn decrease right */
else if ( v_yaw < 0 and delimited_steering_angle < 1 and
          delimited_steering_angle > -1 and brake >= 95% ) then {
  corrected_rear_brake := brake;
  corrected_left_brake := brake;
  corrected_right_brake := brake - rightm;
}

Fig. 7.18. Functionality of Yaw Rate Corrector in Configuration YawCorrection on Right Turn

of the front and rear wheels. Figure 7.18 depicts an extract of the functionality of the configuration YawCorrection in case of a right turn. The local variables rightp and rightm are used to compute a factor for increasing and decreasing the brake forces.

A safety property over the yaw rate corrector is that the sum of the brake forces at the front and at the rear wheels is always between two and three times of the input brake force if the module is not shut down. This property relies on the assumption that the input brake force is always in the range between 0 and 100 interpreted as the percentage of the brake force to be used. Furthermore, the factors for increasing and decreasing the brake force have to be computed such that the final output brake force value is also in this range. Then, it can be shown that the property (YawCorrectionSafety) holds for the yaw rate module.

\[
\text{AG} \left( \text{useconf}_{\text{yawRateCorrector}} \neq \text{Off} \right) \rightarrow \\
(2 \times \text{brake} \leq \\
 \text{corrected\_rear\_brake} \
+ \text{corrected\_left\_brake} \
+ \text{corrected\_right\_brake} \
\leq 3 \times \text{brake})
\]

A correctness property (YawCorrectionCorrect) of the yaw rate corrector module is that the yaw rate in the respective direction decreases if the yaw correction functionality is activated. The variable \( v_{\text{yaw\_old}} \) denotes the yaw rate of the previous cycle while the variable \( v_{\text{yaw\_new}} \) denotes the yaw rate in the current cycle.
The proof of this property requires a number of assumptions on the controlled system depicted in Figure 7.19. Variables with suffix \textit{old} denote the value of the previous cycle while variables with suffix \textit{new} refer to the current value. The first pair of assumptions reflects the correspondence between wheel speeds and brake forces. If the brake force at one wheel increases, the wheel speed of this wheel decreases, and if the brake force decreases, the wheel speed increases. The second pair of assumptions describes how the yaw rate changes if the wheel speeds of the two front wheels change. The first assumption refers to a right turn and states that the value of the yaw rate increases, if the difference of the wheel speeds between left and right wheel decreases. The second assumption states that in case of a left turn the yaw rate decreases if the difference of the wheels speeds between right and left wheel decreases.

The proof of the correctness property (\textit{YawCorrectionCorrect}) uses a case distinction whether the car turns right or left when a yaw rate correction is necessary. In the case of a right turn, we know by the functional description of the configuration \textit{Controlled} that either the brake force at the left front wheel increases or that the brake force at the right front wheel decreases. By the assumptions on (\textit{Brake and Wheel Speed}) we know that either the wheel speed of the left front wheel decreases or that the wheel speed at the right front wheel increases such that the difference of the wheel speeds between left front wheel and right front wheel decreases. By the first assumption on (\textit{Wheel Speed and Yaw Rate}), we can deduce that in case of a decreasing difference between the wheel speeds

\[
AG \left\{ \left( useconf_{YawRateCorrector} = \text{Controlled} \right) \rightarrow \right.
\]

\[
(YawCorrectionCorrect)
\]

\[
\left( v_{\text{yaw old}} < 0 \rightarrow v_{\text{yaw new}} > v_{\text{yaw old}} \right)
\]

\[
\land \left( v_{\text{yaw old}} > 0 \rightarrow v_{\text{yaw new}} < v_{\text{yaw old}} \right)
\]
of the left and the right front wheel the yaw rate increases. The reasoning for the case of a left turn is analogue. Thus, the correctness property (\texttt{YawCorrectionCorrect}) holds.

Since the previous safety and correctness properties for the yaw rate corrector module heavily rely on assumptions on the characteristics of the controlled system and use infinite data domains, direct model checking is not possible. However, (\texttt{Property3}) expressing that the sign of the input brake force is preserved in the yaw rate corrector module can be verified by model checking. As the property is universal, it suffices to consider the yaw rate corrector module alone. Using module slicing, the assignments to the brake forces of the front wheels can be removed as they do not influence the brake force at the rear wheels. This leaves an assignment of the input brake force to the rear wheel brake force output in each configuration. By sign abstraction, the domain of the brake force input and the domain of the brake force output at the rear wheels can be reduced to the abstract domain \{\texttt{pos}, \texttt{zero}\}. By model checking, we can establish that the abstract property (\texttt{AbstractProperty3}) holds over the sliced and abstracted module.

\[
\begin{align*}
\text{AG} \left( \texttt{useconf}_{\text{YawRateCorrector}} \neq \texttt{Off} \rightarrow \right. \\
\left. (\text{brake} = \texttt{pos} \rightarrow \text{corrected\_rear\_brake} = \texttt{pos}) \right. \\
\left. \land (\text{brake} = \texttt{zero} \rightarrow \text{corrected\_rear\_brake} = \texttt{zero}) \right)
\end{align*}
\]

Using the results for property preservation under combined model transformations in Section 6.1.1 and the result on module decomposition in Corollary 6.29, we can establish (\texttt{Property3}) for the vehicle stability control system. This property shows that a combination of slicing and abstraction is required to be able to complete a verification task by model checking.

\subsection*{7.2.2 Verification of Adaptation Behaviour}

The adaptation behaviour of the vehicle stability control system is finite by design of the system. Therefore, adaptation properties can be verified by model checking without prior abstraction. Compositional reasoning strategies are not applicable for verification of the generic adaptation properties (cf. Figure 3.9), since these properties are not universal, except for the third generic adaptive property. Instead, model reduction by slicing implemented in the AMOR framework can be used to increase the efficiency of model checking while preserving the validity of the generic adaptation properties. To this end, we investigated how slicing at different levels of granularity
7.2. Verification of Adaptive Vehicle Stability Control

influences the required analysis times for the generic adaptation properties as well as for application-specific adaptive properties. For the third generic adaptation property and for universal application-specific adaptive properties, we further compared slicing techniques and the decomposition strategy with respect to the required analysis times, since decomposition preserves the validity of universal properties.

Verification Complexity Reduction by Slicing

For evaluation of verification complexity reduction by slicing [SPH08], we reduced the original vehicle stability control system with respect to the generic adaptation properties (cf. Figure 3.9) for each module \( i \) by system, module and adaptive slicing as well as by a combination of adaptive and system slicing. Furthermore, we reduced the original system with respect to each of the three fail-safe properties of the core controller modules (cf. Figure 3.10) by system, module and adaptive slicing as well as by a combination of adaptive and system slicing. We measured the times necessary for each reduction and the times for the analysis of the respective system and property by model checking using the Averest model checker Version 1.9.2 and the NuSMV model checker Version 2.4.3.\(^2\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Model</td>
<td>232</td>
<td>n/a</td>
<td>9.22</td>
<td>263.35</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Adaptive Sliced Model</td>
<td>148</td>
<td>0.007</td>
<td>2.78</td>
<td>143.12</td>
<td>23778.66</td>
<td>447.55</td>
</tr>
<tr>
<td>System Sliced Model</td>
<td>112</td>
<td>0.140</td>
<td>2.10</td>
<td>9.57</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Adaptive-System Sliced M.</td>
<td>48</td>
<td>0.043</td>
<td>1.53</td>
<td>12.03</td>
<td>7.35</td>
<td>7.75</td>
</tr>
<tr>
<td>Module Sliced Model</td>
<td>38</td>
<td>0.043</td>
<td>1.49</td>
<td>11.90</td>
<td>2.85</td>
<td>4.75</td>
</tr>
</tbody>
</table>

Table 7.1. Analysis of Slicing for Adaptive Properties

Table 7.1 shows the average sizes of the systems in terms of the number of variables before and after reduction by slicing in different levels of detail and the average times required for the reductions. Furthermore, it contains the average analysis times required by the Averest and the NuSMV model checkers for the generic adaptation properties for each module and for the fail-safe properties of the controllers modules over systems reduced with the same slicing technique. As the functional behaviour of the vehicle

\(^2\) All results were computed on a Dual-Core AMD Opteron Server with 2.0 GHz and 4 GB RAM.
stability control system uses integer variables, the original system cannot be represented in NuSMV. The same holds for the system sliced version of the system that still contains integer variables for functional data. Hence, we have to apply adaptive slicing removing functional behaviour in order to translate the system to NuSMV. Thus, we could only measure analysis times in NuSMV for the adaptive sliced, the adaptive-system sliced and the module sliced versions of the system.

In the experiments, it can be observed that the size of the model expressed by the average number of variables decreases, the more detailed the reduction is. As expected, module slicing provides the best possible model reduction with respect to the number of system variables, cutting them by 85%. Furthermore, we see that the time for computing the reduction increases, the more detailed the analysis is. In this direction, adaptive slicing allows large model reductions without a detailed analysis by design of SAS models. The reason why system slicing in this example takes longer than module slicing is that the number of interconnections in the considered example system is rather high whereas the modules themselves do not contain many variables. In a more loosely coupled system, system slicing can outperform module slicing. Furthermore, if we perform adaptive slicing followed by system slicing, the time for the reduction is about the same as for module slicing because functional connections do not have to be considered for system slicing.

With respect to the average analysis times required by the Averest and NuSMV model checkers, it can be observed that all reductions allow a significant speed up in verification time. For instance, the average analysis time for the fail-safe properties decreases by 95% from the original to the module sliced system in the Averest model checker and by 99% from the adaptive sliced to the module sliced system in the NuSMV model checker. For the generic adaptation properties as well as for the fail-safe properties, it holds that the smaller the model is, the more efficient verification is possible. The verification of the generic adaptation properties with NuSMV on the average takes more than 6.5 hours (= 234000 s) per verification task on the adaptively sliced system. One of the verification tasks could not be completed at all because NuSMV ran out of memory. In this direction, slicing makes model checking of the generic adaptation properties with NuSMV at all feasible since the properties can be established on a module sliced system within seconds.

Comparing the performance of NuSMV and Averest, we see that model reductions improve the efficiency of model checking with NuSMV to a much
greater extend. For instance, for the generic adaptation properties the average analysis time with NuSMV is reduced by 60% from the adaptive-system sliced to the module sliced system while for Averest the analysis time is approximately the same. The reduction for the adaptively sliced system to the module sliced system is even more dramatic. The performance of Averest is due to more sophisticated internal model reductions because Averest can exploit the structural properties of synchronous systems. This also explains why in all considered examples Averest performs better with respect to the average analysis times than NuSMV. The average analysis time in Averest for the fail-safe properties on a system sliced version of the system is unexpectedly small. The reason is that coincidentally this specific verification task allows an efficient representation of the models in the internal BDD packages of the model checker which does not apply in general and cannot be efficiently predicted in advance.

**Verification Complexity Reduction by Decomposition**

The third generic adaptation property and the fail-safe properties of the controller modules are universal properties such that verification by decomposition allows lifting the result to the overall system. To this end, we compared verification complexity reductions by decomposition to reductions by slicing with respect to required analysis times. Environment assumptions are not necessary to establish the properties. We reduced the original system with respect to the third generic adaptive property for each module \(i\) with adaptive and module slicing. Further, we decomposed the system such that it only contained the considered module \(i\). For the fail-safe properties over the controller modules, we reduced the original system with adaptive and module slicing and decomposed it into the controller modules. We measured the times required by the Averest and NuSMV model checkers for analysing the respective property over the original system, over the adaptive sliced system, over the module sliced system and over the modules in the decomposition.

Table 7.2 shows the average sizes of the original system, of the sliced systems and of the modules in terms of their variables. Further, it presents the average times required by Averest and NuSMV for the analysis of the third generic adaptation property and the fail-safe properties of the controller modules on the systems reduced with the same technique. Again, we could not translate the original system to NuSMV as it contains integer variables for functional data such that these analysis results are missing.
In our experiments, we observe that decomposition reduces the size of the verification task expressed in terms of the number of variables substantially. On the average, each module has 10 variables which is only about a forth of the variables in the module sliced system. The average analysis times required by Averest and NuSMV on the decomposed modules compared to the analysis times required for a module sliced system are considerably smaller. Again, model reductions are more important for the NuSMV model checker as verification times are minimised to a much greater extent than for Averest. For instance, the average analysis time for the third generic adaptation property with the NuSMV model checker decreases by a factor of 43.000 from the adaptive sliced system to the modules in the decomposition. For the Averest model checker, verification of the third generic adaptive property is about 25 times faster comparing the analysis of the adaptive sliced system with the analysis of the modules in the decomposition. In general, the results show that whenever a universal property can be established over a decomposition of a system, verification complexity is reduced more significantly compared to reductions achievable by slicing techniques.

### 7.2.3 Results of Case Study

The AMOR integration framework for model-based development and formal verification proposed in this thesis has been evaluated together with the development of the adaptive vehicle stability control system (cf. Chapter 2). For verification of functional and adaptive properties considering generic as well as application-specific aspects of the system model, we have applied the compositional reasoning strategies and model transformations implemented in the AMOR framework in order to bridge the gap between high-level modelling concepts and input for verification tools.
In our case study, it turned out that compositional reasoning strategies and data domain abstractions simplified verification tasks regarding functional properties to an extended such that formal verification became at all feasible. Properties expressed over the complete system model could be split into a number of less complex local verification tasks that could be handled separately. Data domain abstractions made those verification tasks amenable to automatic verification by model checking. Using the results on property-preservation for compositional reasoning and abstractions, the validity of the properties could be lifted to the complete system. Without prior verification complexity reductions, it would have been impossible to verify those properties. Furthermore, characteristics of the controlled system influencing the validity of the considered properties could be handled by environment assumptions as proposed in Section 6.2.5. Some properties were jointly established by theorem proving and model checking, for instance if functionality was not fully specified. This proves the necessity of an independent intermediate layer allowing to produce verification input for different verification tools in order to exploit their strengths for specific verification tasks.

For properties of the adaptation behaviour, verification complexity reductions by decomposition and slicing could increase the efficiency of formal verification enormously. Whereas the Averest model checker could deal with the original system model in a reasonable amount of time due to its specialised procedures for synchronous systems, verification with the NuSMV model checker heavily required model reductions before generation of verification input in order to make the analysis at all feasible. In the AMOR framework, system models are captured at a high-level of abstraction which allows using the internal model structure for analysis of irrelevant system parts with respect to a property. In the case study, adaptive and system slicing have proven to be valuable techniques for model reduction offering a significant decrease in analysis time without a detailed model analysis. Decomposition exploiting the module structure present in the intermediate SAS model representation can further reduce the analysis effort considerably if applicable.

To sum up, the case study has shown that the integration framework proposed in this thesis eases the use of formal verification in the model-based development process for adaptive embedded systems. System models and their properties are formally captured at a high level of abstraction. Compositional reasoning strategies and model transformations by slicing and abstraction reduce verification complexity before input for verification
tools is generated. This facilitates more efficient analysis of models with respect to their properties. In certain cases, the applied reductions make formal verification at all possible. The independent intermediate model representation further allows using different verification techniques and tools such that the technique and tool most suitable for a verification task can be employed.
Chapter 8

Conclusion and Outlook

"The best way to predict the future is to invent it."

(Alan Kay)

Embedded systems increasingly control safety-critical functionality in many domains, for example in automotive, medical or industrial control applications. In this context, adaptation has become state-of-the-art to meet the high demands on safety and availability by reconfiguring system behaviour in case of changing environment conditions as well as in case of failures. However, adaptation significantly complicates system design due to the fact that adaptation of one part of a system may cause adaptations in other parts. Model-based development is one approach to deal with the increased complexity adaptation imposes on system design by providing means to focus on adaptation and functionality of a system in isolation. Furthermore, models of adaptive systems can be analysed already at early stages of their design. This allows discovering conceptual errors and design flaws, before a system is actually implemented. In this context, formal verification of models with respect to their properties is a promising approach to increase the reliability of adaptive systems. However, modelling concepts use a high level of abstraction for representing relevant system aspects, while input for verification tools is based on low-level mathematical concepts. Hence, there is a need to bridge the gap between modelling concepts and verification tools in order to alleviate the use of formal verification in the model-based design process.
8.1 Contributions

In this direction, this thesis has presented an integration of model-based development for adaptive embedded system with existing verification techniques [ASSV07]. The integration relies on a formal semantics-based intermediate representation of the modelling concepts at a high level of abstraction by synchronous adaptive system (SAS). SAS allow formulating system models as well as their properties in a semantically exact manner. Using interactive theorem proving and static analysis, model properties can be checked directly. In order to make models amenable to automatic verification by model checking, model complexity is reduced by transformations. The transformations, including slicing in different levels of detail and data domain abstractions, are formally verified to be property-preserving using a translation validation approach. Furthermore, compositional reasoning strategies reduce verification effort significantly by splitting global verification tasks into less complex local tasks. If the transformed models are small enough to be handled by automatic verification techniques, verification input for verification tools can be generated.

In this thesis, we have made the following contributions towards an integration of model-based development with formal verification for adaptive embedded systems:

- We have defined synchronous adaptive systems (SAS) as a formal semantics-based model for adaptive embedded systems capturing MARS modelling concepts for adaptive systems at a high level of abstraction [SPH06b, ASSV07]. SAS models constitute the basis of the intermediate model representation in the proposed integration framework. Furthermore, we have presented $L_{SAS}$ as a temporal logic for expressing properties of synchronous adaptive systems.
- Based on the formal intermediate representation, model properties can be verified directly. To this end, we have described how to analyse structural consistency of models by static analysis. Further, we have established properties of SAS models by theorem proving. Therefore, we have given a translation of synchronous adaptive systems and their properties into the Isabelle/HOL [NPW02] theorem prover.
- In order to make models amenable to formal verification by model checking, we have presented model transformations on SAS for verification complexity reduction. Slicing techniques [SPH08] in different levels of granularity offer a fine-grained control over the analysis effort required for model reduction and exploit structural information still
present in the intermediate model representation. Data domain abstractions [SPH06b] can be applied to transform an infinite-state model into an abstract finite-state model that can be efficiently model checked. The model transformations are proved correct using a translation validation approach [BSPH07] seamlessly integrated into our framework.

- Splitting a global verification task into a number of less complex local tasks eases verification effort significantly. To this end, we have proposed compositional reasoning strategies [SPH06a] for synchronous adaptive systems. The strategies range from simple decompositions of systems and properties, over assume-guarantee reasoning and interface abstraction to special compositional proof rules for adaptive systems. By a maximal model construction, we have reduced assume-guarantee reasoning for SAS models to a standard model checking problem. Combining compositional reasoning with model transformations offers additional potential for verification complexity reduction. Analogue to assume-guarantee reasoning, we have presented an approach to deal with the characteristics of the controlled system in verification of controller properties in the embedded systems domain.

- We have prototypically implemented the proposed integration in the AMOR framework [ASSV07]. This has been empirically evaluated in the development of an adaptive vehicle stability control system [ASS08]. At this case example, we have demonstrated that the proposed integration framework improves usability of formal verification in the model-based design process for adaptive embedded systems. Using the independent intermediate layer, we have shown that verification tasks can be completed by different state-of-the-art verification techniques, i.e. by static analysis, theorem proving and model checking. Model transformations and compositional reasoning strategies significantly improve the efficiency of formal verification. Slicing and decompositions, for instance, allow a dramatic decrease in analysis time for generic adaptation properties. In some cases, model reductions and transformations are necessary to make models at all amenable to automatic verification by model checking. In the case example, this holds in particular for functional properties of the vehicle stability control system.

### 8.2 Future Work

The integration of formal verification into the model-based design process for adaptive embedded systems presented in this thesis offers many
prospects for future work. The scope of verification tasks that can be handled for adaptive embedded system can be widened. In this direction, the considered system models can be augmented to comprise more features requiring additional verification complexity reduction techniques. In this thesis, MARS and SAS models are modular systems without hierarchy. Hierarchical specification mechanisms can be used to build a system from a set of predefined components being systems themselves. An extension of synchronous adaptive systems to hierarchical synchronous adaptive systems would enable reuse of system components and already established properties in a hierarchical system composition. Furthermore, the model of adaptation behaviour can be widened. While currently functionality and adaptation behaviour are clearly separated, the modelling concepts can be augmented to allow for a closer interconnection. One idea would be to use all configurations of a module in parallel and to merge the results in order to increase reliability of the overall module output. Another possibility would be to allow run time adaptation of configuration priorities in order to find more appropriate adaptation behaviour in a particular environment situation.

Model transformations for verification complexity reduction can be improved in order to increase the reduction potential of system models. While the presented slicing techniques for model reduction only consider the variables contained in a property as a slicing criterion, a more detailed analysis of the property structure [HDZ00] could be used in order to offer a greater reduction. Abstractions of SAS models could further be expanded from data domain abstractions to more rigorous abstractions. Predicate abstraction [GS97] of SAS models could increase the possible verification complexity reduction. In this direction, a set of suitable abstraction predicates for adaptive systems has to be found for capturing adaptive system properties beyond valuations of variables in a state. Furthermore, the application of predicate abstraction to SAS would facilitate using counterexample-guided abstraction refinement [CGJ+03] in SAS verification. Finding abstractions suitable for verification of a model property automatically alleviates the usage of formal verification in the design process.

With respect to usability of formal verification in the model-based design process, the proposed integration framework can be extended to offer a closer interconnection of the employed verification tools. To this end, a theorem prover could provide guidance in the application of compositional reasoning strategies for splitting a global proof goal into smaller subgoals.
8.2. Future Work

Afterwards, a model checker could be invoked as a special proof tactic to discharge the subgoals automatically. Approaches in this direction have already been undertaken, for instance the integration of a software model checker into Isabelle/HOL [DMSS05] as a special proof tactic. Additionally, the compositional strategies themselves can be improved. In assume-guarantee reasoning, it may be hard to find an assumption that is strong enough to prove the required guarantee. The integration of work on automatic learning of assumptions for assume-guarantee reasoning [Ang87, JGP03] could help to overcome this problem. The reduction potential by interface abstraction could be enlarged if a system part could be abstracted with respect to a safe approximation of the behaviour at its interface. Such overapproximations of the interface behaviour could for instance be obtained by implementing an abstract interpretation [CC77, CC79] of synchronous adaptive systems.

A more intuitive specification formalism for model properties beyond mere temporal logic would be advantageous in order to alleviate the use of formal verification. This specification formalism should allow a system designer to express desired properties of a system at modelling level in terms of the intuitive modelling concepts. As a starting point in this direction, life sequence charts (LSC) [DH99] could be adapted to the characteristics of adaptive systems and translated to the specification logic of the integration framework. Additionally, the results of the verification process have to be propagated back to the modelling level in order to give the system designer feedback where exactly a model violates the desired properties. In this direction, an analysis of the output of the verification tools and a re-translation to the modelling concepts would be required.
References


<table>
<thead>
<tr>
<th>References</th>
<th>Title</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


About the Author

Ina Schaefer (*1977) studied Computer Science with a minor in Mathematics at Rostock University from 1997-2003. In the academic year 2000/2001, she was a visiting student at the University of Oxford, UK. In September 2003, she received her Diploma in Computer Science (Dipl.-Inf.).

From 2003 - 2005, she worked as a researcher in the Programming Logics Department, headed by Prof. Dr. H. Ganzinger, at the Max-Planck-Institute for Computer Science, Saarbrücken. Since 2006, she is a researcher in the Software Technology Group, headed by Prof. Dr. A. Poetzsch-Heffter, at Kaiserslautern University.

Her research interests are in the area of formal methods and verification with a particular focus on integrating formal techniques into the development processes of software systems.