On Translation Validation for System Abstractions*

Jan Olaf Blech, Ina Schaefer, Arnd Poetzsch-Heffter

{blech, inschaef, poetzsch}@informatik.uni-kl.de

Technical Report
No. 361/07

July 2007

Computer Science Department
University of Kaiserslautern

Abstract. Abstraction is intensively used in the verification of large, complex or infinite-state systems. With abstractions getting more complex it is often difficult to see whether they are valid. However, for using abstraction in model checking it has to be ensured that properties are preserved. In this paper, we use a translation validation approach to verify property preservation of system abstractions. We formulate a correctness criterion based on simulation between concrete and abstract system for a property to be verified. For each distinct run of the abstraction procedure the correctness is verified in the theorem prover Isabelle/HOL. This technique is applied in the verification of embedded adaptive systems. This paper is an extended version of our work published as [5].

* supported by the Rheinland-Pfalz Cluster of Excellence 'Dependable Adaptive Systems and Mathematical Modelling' (DASMOD)

1 Introduction

Recently, a large amount of research has addressed the verification of large, complex or infinite-state systems using model checking. Due to inherent limitations model checkers are unable to deal with such systems directly. So research concentrated on finding abstractions reducing the state space sufficiently while preserving necessary precision. However, since abstraction procedures are getting more complex it is not always clear if they are valid, i.e. that properties verified for the abstract system also hold in the concrete system. In principle, there are two approaches to guarantee correctness of abstractions: Abstraction algorithms (and their implementations!) are verified once and for all. Alternatively, abstraction results of each distinct run of the abstraction procedure are proved correct. In this work, we will propose a technique for guaranteeing abstraction correctness using the second approach.

The overall structure of our approach is depicted in Figure 1. For verifying a system abstraction, the abstraction procedure is given a concrete system comprising a property to be checked. As output an abstract system with a corresponding abstract property is produced. Furthermore, a proof script is generated capturing the actual proof that the abstraction preserves the considered property. In this direction, a correctness criterion based on simulation between abstract and concrete system is formalized. Using the proof script this criterion is checked for the considered concrete and abstract systems and properties in the theorem prover Isabelle/HOL [15]. Thus the correctness of an abstraction is verified for each run of the abstraction 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.

Our work towards runtime verification of system abstractions is inspired by a translation validation [16] based approach for compilers [11,4,17]. 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 abstractions are usually less complex and easier to establish than proofs for a general abstraction procedure. An additional advantage is that the abstraction procedure can be tailored to a particular system and property under consideration and thus match the

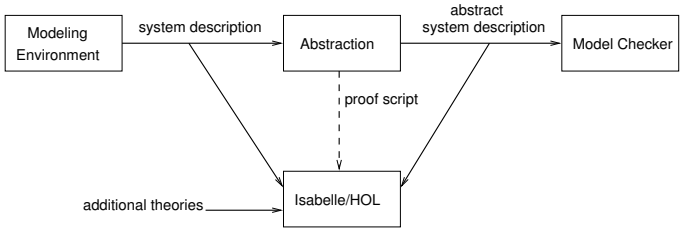

Fig. 1. Our Translation Validation Infrastructure

requirements of the concrete problem very closely while still being proved correct. Also note, that in our approach the correctness of abstractions is proved formally using a theorem prover instead of a paper-and-pencil-proof.

The proposed technique is applied in verification of embedded adaptive systems [19]. Beside potentially unbounded data domains the size of the considered systems is huge. For efficient verification by model checking, these systems have to be abstracted in a property-preserving way. We have successfully applied runtime verification of the necessary abstractions in this domain.

This paper is structured as follows: Section 2 describes the application domain of our work. In Section 3, we present a theorem on property preservation. This is used in the implementation and proving strategies in Section 4. A short evaluation is given in Section 5. We discuss related work in Section 6 before concluding in Section 7. In the appendix A, a alternative construction for the proof of the property preservation theorem can be found.

2 Adaptive System Verification

Our approach is applied to adaptive systems as used in the automotive sector. Adaptation is particularly important in the safety-critical automotive domain to meet the high demands on dependability and fault-tolerance. For example, if the sensor measuring the yaw rate of a car fails, the vehicle stability control system may adapt to a configuration, where the yaw rate is approximated by steering angle and vehicle speed. In this way, it can be guaranteed that the system is still operational even if some of the components fail in order to provide a maximum degree of safety and reliability. However, adaptation significantly complicates the development of embedded systems. The integration of formal verification into the model-based development process is important to rigorously prove that adaptation behaviour meets critical requirements. However, adaptive systems developed in a modelling environment also used for other purposes such as code generation contain a level of detail not amenable for automatic verification. This makes system abstractions indispensable and requires support for automatically verifying their correctness.

In this section, we firstly propose a model that formally captures the semantics of adaptive systems at a high level of abstraction. Secondly, we propose a temporal specification logic such that desired system properties can be formulated in a semantically exact manner. Finally, we show at an example how abstractions facilitate verification.

2.1 A Formal Model for Synchronous Adaptive Systems

Synchronous adaptive systems are composed from a set of modules where each module has a set of predetermined behavioural configurations it may adapt to. The selected configuration depends on the status of the module's environment and is determined by an adaptation aspect defined on top of the functional

behaviour. The modules are connected via links between input and output variables. However, data and adaptation flow are decoupled and do not follow the same links. Adaptations in one module may trigger adaptations in other modules by internal adaptation signals via the adaptation links. That may lead to a chain reaction of adaptations through the system. The systems are assumed to be open systems with non-deterministic input provided by an environment. Furthermore, they are modelled synchronously as then simultaneously invoked actions are executed in true concurrency.

2.2 Syntax of SAS

We define a synchronous adaptive system (SAS) [19]over a set of values Val and a set of variables Var. The smallest construction element of an SAS is a module. It contains a set of different predetermined configurations the module can adapt to. Adaptation is realised by an adaptation aspect. Before the execution of the actual functionality the adaptation aspect evaluates the configuration guards and determines the configuration to use. Note that the configuration guards may only depend on the adaptive variables in order to have a clean syntactic separation between functional and adaptive behaviour. This enables reasoning about purely adaptive properties by projecting the model onto its adaptive parts.

Definition 1 (Module and Adaptation). *An SAS module m is a tuple $m = (in, out, loc, init, confs, adaptation)$ with*

- *$in \subseteq Var$, the set of input variables, $out \subseteq Var$, the set of output variables, $loc \subseteq Var$, the set of local variables and $init : loc \to Val$ their initial values*
- *$confs = \{(guard_j, next\_state_j, next\_out_j) \mid j = 1, ..., n\}$ the configurations of the module, where*
  - *$guard_j$: the Boolean closure of constraints on $\{adapt\_in, adapt\_loc\}$ determining when configuration j is enabled with $adapt\_in$ and $adapt\_loc$ as defined below*
  - *$next\_state_j$: $(in \cup loc \to Val) \to (loc \to Val)$ the next state function for configuration j*
  - *$next\_out_j$: $(in \cup loc \to Val) \to (out \to Val)$ the output function for configuration j*

The adaptation aspect is defined as a tuple $adaptation = (adapt\_in, adapt\_out, adapt\_loc, adapt\_init, adapt\_next\_state, adapt\_next\_out)$ where

- *$adapt\_in \subseteq Var$, the set of adaptation in-variables, $adapt\_out \subseteq Var$, the set of adaptation out-variables, $adapt\_loc \subseteq Var$, the set of adaptation local state variables and $adapt\_init : adapt\_loc \to Val$ their initial values*
- *$adapt\_next\_state : (adapt\_in \cup adapt\_loc \to Val) \to (adapt\_loc \to Val)$ the adaptation next state function*
- *$adapt\_next\_out : (adapt\_in \cup adapt\_loc \to Val) \to (adapt\_out \to Val)$ the adaptation output function*

A SAS system is composed from a set of modules that are interconnected with their input and output variables. The system is an open system with an environment providing non-deterministic input. For technical reasons, we have to assume that the variable names of all modules in a composed system are pairwise disjoint. This can be easily achieved by indexing module variables with the respective module index.

Definition 2 (SAS). *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* $m_i = (in_i, out_i, loc_i, init_i, confs_i, adaptation_i)$
- $input_a \subseteq Var$ *are adaptation inputs and* $input_d \subseteq Var$ *functional inputs to the system*
- $output_a \subseteq Var$ *are adaptation outputs and* $output_d \subseteq Var$ *functional outputs from the system*
- $conn_a$ *is a function connecting adaptation outputs to adaptation inputs, system adaptation inputs to module adaptation inputs and module adaptation outputs to system adaptation outputs, i.e.* $conn_a : \bigcup_{j,k=1,\ldots,n}(adapt\_out_j \cup input_a) \rightarrow (adapt\_in_k \cup output_a)$ *where* $conn_a(input_a) \subseteq adapt\_in_k$
- $conn_d$ *is a function connecting outputs of modules to inputs, system inputs to module inputs and module outputs to system outputs, i.e.* $conn_d : \bigcup_{j,k=1,\ldots,n} (out_j \cup input_d) \rightarrow (in_k \cup output_d)$ *where* $conn_d(input_d) \subseteq in_k$.

2.3 Semantics of SAS

The semantics of SAS is defined in a two-layered approach. We start by defining the local semantics of single modules similar to standard state-transition systems. From this, we define global system semantics. A local state of a module is defined by a valuation of the module's variables, i.e. 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 and input and output variables are undefined. 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 adaptative state. The system designer should ensure that the system has a built-in default configuration 'off' which becomes applicable 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.

Definition 3 (Local States and Transitions). *A local state s of an SAS module m is defined as variable assignment:*

$$s : in \cup out \cup loc \cup adapt\_in \cup adapt\_out \cup adapt\_loc \rightarrow Val$$

A local state s is called initial if $s|_{loc} = init$, $s|_{adapt\_loc} = adapt\_init$ and $s|_V = undef$ for $V = in \cup out \cup adapt\_in \cup adapt\_out$[1]. A local transition between two local states s and s' is defined as $s \rightsquigarrow s'$ iff the following conditions hold:

$$\begin{gathered} s'|_{adapt\_loc} = adapt\_next\_state(s'|_{adapt\_in} \cup s|_{adapt\_loc}) \\ s'|_{adapt\_out} = adapt\_next\_out(s'|_{adapt\_in} \cup s|_{adapt\_loc}) \\ \forall\ 0 < j < i\ .\ s'|_{in} \cup s|_{loc} \cup s'|_{adapt\_in} \cup s|_{adapt\_loc} \not\models guard_j \\ s'|_{in} \cup s|_{loc} \cup s'|_{adapt\_in} \cup s|_{adapt\_loc} \models guard_i \\ s'|_{loc} = next\_state_i(s'|_{in} \cup s|_{loc}) \text{ and } s'|_{out} = next\_out_i(s'|_{in} \cup s|_{loc}) \end{gathered}$$

The state of an SAS is the union 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 input and output is undefined. A transition between two global states is performed in three stages. Firstly, each module reads its input either from another module's output of the previous cycle or from the system inputs in the current cycle. Secondly, each module synchronously performs a local transition. Thirdly, the modules directly connected to system outputs write their results to the output variables.

Definition 4 (Global States and Transitions). *A global state σ of an 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 inputs and outputs, i.e. $\sigma = s_1 \cup \ldots \cup s_n \cup ((input_a \cup input_d \cup output_a \cup output_d) \rightarrow Val)$. A global state σ is called initial iff all local states s_i for $i = 1, \ldots, n$ are initial and the system inputs and outputs are undefined. Two states σ and σ' perform a global transition, written $\sigma \rightarrow_{glob} \sigma'$, iff*

- *for all $x, y \in Var \setminus (input_d \cup input_a)$ with $conn_d(x) = y$ or $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma(x)$, for all $x \in input_a$ and $y \in Var$ with $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$ and for all $x \in input_d$ and $y \in Var$ with $conn_d(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$*
- *for all $s_j \in \sigma$ and for all $s'_j \in \sigma'$ it holds that $s_j \rightsquigarrow s'_j$*
- *for all $x \in Var$ and $y \in output_d$ with $conn_d(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$ and for all $x \in Var$ and $y \in output_a$ with $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$*

This definition of SAS semantics induces a transition system. In the following defintion we fix the notion of an SAS transition system.

[1] For a function f and a set $M \subseteq Var$, $f|_M = \{(x, f(x)) \mid x \in M\}$ is the restriction of f to the domain M.

Definition 5 (SAS Transition System). *The transition system induced by an SAS model is defined as* $\mathcal{T}_{SAS} = (\Sigma, Init, \rightsquigarrow_{glob})$ *where*

- Σ *is the set of global SAS states*
- *Init ist the set of initial SAS states*
- $\rightsquigarrow_{glob}$ *is the global SAS transition relation*

The SAS semantics is now defined by the set of possible paths of a transition system. In each step system input is provided from the environment.

Definition 6 (Paths). *A path of* $\mathcal{T}_{SAS}$ *is defined as a sequence of global states* $\sigma_0\sigma_1 \ldots$ *where* $\sigma_0 \in Init$ *and for all* $0 \leq i$ *we have* $\sigma_i \rightsquigarrow \sigma_{i+1}$*. The set* $Paths(\mathcal{T}_{SAS}) = \{\sigma_0\sigma_1 \ldots \text{ a path of } \mathcal{T}_{SAS}\}$ *is the set of possible paths of* $\mathcal{T}_{SAS}$ *and defines the SAS semantics. Let* π_j *denote the suffix of path* $\pi = \sigma_0\sigma_1\sigma_2 \ldots$ *starting in state* σ_j*.*

2.4 Specification Logic

We define $\mathcal{L}_{SAS}$ as the language to express properties over a SAS with a set of Variables *Var* and a domain of values *Val*. $\mathcal{L}_{SAS}$ is a variant of the computational tree logic CTL* in order to express temporal properties of SAS system paths.

For reasoning about the configuration a SAS module uses in each cycle a special variable *useconf* is added to the set of Variables *Var*. It is assigned only to the indices of the configurations a module m_i implements. For a module m_i in a transition $\rightsquigarrow$ from state s_i to state s_i' *useconf* is assigned as follows:

$$s_i(useconf) = k \text{ iff } \quad s'|_{in} \cup s|_{loc} \cup s'|_{adapt\_in} \cup s|_{adapt\_loc} \models guard_k$$
$$\forall\ 0 < j < k,\ s'|_{in} \cup s|_{loc} \cup s'|_{adapt\_in} \cup s|_{adapt\_loc} \not\models guard_j$$

However, for our formal treatment, *useconf* can be treated as an ordinary variable taking values from *Val* which includes the configuration indices. In the definition of $\mathcal{L}_{SAS}$, we only allow equality between variables and values as basic relational symbol. However, this can be extended to more general terms build up by arithmetic operations. Furthermore, other relations like less or greater could be used.

Definition 7 ($\mathcal{L}_{SAS}$). *Let* $x, y \in Var$ *and* $v \in Val$*.*

$$\begin{aligned} Atoms\, a &::= x = v \mid x = y \\ StateFormula\, \varphi &::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{E}\ \psi \\ PathFormula\, \psi &::= \varphi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \mathsf{X}\ \psi \mid \psi_1\ \mathsf{U}\ \psi_2 \end{aligned}$$

We define the other CTL operators with the usual abbreviations as follows:*

$$\begin{aligned} \mathsf{F}\,\psi &\equiv true\ \mathsf{U}\ \psi \\ \mathsf{G}\,\psi &\equiv \neg\,\mathsf{F}\,\neg\psi \\ \mathsf{A}\,\psi &\equiv \neg\,\mathsf{E}\,\neg\psi \end{aligned}$$

The fragment of $\mathcal{L}_{SAS}$ where only the universal path quantifier A is used is called the universal fragment $\mathsf{A}\mathcal{L}_{SAS}$. Now we define satisfiability of $\mathcal{L}_{SAS}$-formulas over SAS models

Definition 8 (Satisfaction). *Let $\mathcal{T}_{SAS}$ be a SAS transition system. For a state formula φ, $(\mathcal{T}_{SAS}, \sigma) \models \varphi$ is defined inductively on the structure of φ.*

- $(\mathcal{T}_{SAS}, \sigma) \models \mathit{true}$ *always*
- $(\mathcal{T}_{SAS}, \sigma) \models (x = v)$ *iff* $\sigma(x) = v$ *and* $(\mathcal{T}_{SAS}, \sigma) \models (x = y)$ *iff* $\sigma(x) = \sigma(y)$
- $(\mathcal{T}_{SAS}, \sigma) \models \neg\varphi$ *iff* $(\mathcal{T}_{SAS}, \sigma) \not\models \varphi$
- $(\mathcal{T}_{SAS}, \sigma) \models \varphi_1 \wedge \varphi_2$ *iff* $(\mathcal{T}_{SAS}, \sigma) \models \varphi_1$ *and* $(\mathcal{T}_{SAS}, \sigma) \models \varphi_2$
- $(\mathcal{T}_{SAS}, \sigma) \models \mathsf{E}\psi$ *iff there exists a path* $\pi = \sigma\sigma_1\sigma_2 \ldots$ *such that* $(\mathcal{T}_{SAS}, \pi) \models \psi$.

For a path formula ψ, $(\mathcal{T}_{SAS}, \pi) \models \psi$ is defined inductively on the structure of ψ.

- $(\mathcal{T}_{SAS}, \pi) \models \varphi$ *iff* $\pi = \sigma\sigma_1\sigma_2 \ldots$ *and* $(\mathcal{T}_{SAS}, \sigma) \models \varphi$
- $(\mathcal{T}_{SAS}, \pi) \models \neg\psi$ *iff* $(\mathcal{T}_{SAS}, \pi) \not\models \psi$
- $(\mathcal{T}_{SAS}, \pi) \models \psi_1 \wedge \psi_2$ *iff* $(\mathcal{T}_{SAS}, \pi) \models \psi_1$ *and* $(\mathcal{T}_{SAS}, \pi) \models \psi_2$
- $(\mathcal{T}_{SAS}, \pi) \models \mathsf{X}\ \psi$ *iff* $\pi = \sigma\pi_1$ *and* $(\mathcal{T}_{SAS}, \pi_1) \models \psi$
- $(\mathcal{T}_{SAS}, \pi) \models \psi_1\ \mathsf{U}\ \psi_2$ *iff* $\exists k \geq 0$ *such that* $(\mathcal{T}_{SAS}, \pi_k) \models \psi_2$ *and* $\forall\ 0 \leq j \leq k$ $(\mathcal{T}_{SAS}, \pi_j) \models \psi_1$

Satisfiability of a universal state formula $\mathsf{A}\psi$ can also be defined directly by: $(\mathcal{T}_{SAS}, \sigma) \models \mathsf{A}\psi$ iff for all paths $\pi = \sigma\sigma_1\sigma_2 \ldots$ it holds that $(\mathcal{T}_{SAS}, \pi) \models \psi$. For a state formula φ we define $\mathcal{T}_{SAS} \models \varphi$ if for all $\sigma_0 \in \mathit{Init}$ we have $(\mathcal{T}_{SAS}, \sigma_0) \models \varphi$ and for a path formula ψ we define $\mathcal{T}_{SAS} \models \psi$ if $(\mathcal{T}_{SAS}, \pi_0) \models \varphi$ for all $\pi_0 \in \mathit{Paths}(\mathcal{T}_{SAS})$ starting in initial states $\sigma_0 \in \mathit{Init}$.

2.5 SAS Verification using Abstraction

Synchronous adaptive systems are in general to complex to be verified directly. Therefore, a number of transformations on SAS models must be applied to reduce this verification complexity. Abstractions are one technique used in this direction. As example how abstraction facilitates verification of synchronous adaptive systems, consider a system that consists of one module with two different configurations. If the input is bigger than a certain threshold, say 50, the module switches to its first configuration. This configuration uses a specific algorithm for computing the output. If the input is smaller than 50, the module uses configuration 2 computing the output in a different way. An important property of this example system is that everytime the input exceeds 50 configuration 1 is used in order to make sure that the appropriate algorithm is applied. This property can be stated in $\mathcal{L}_{SAS}$ as $\varphi \equiv \mathsf{AG}(\mathit{input} \geq 50 \rightarrow \mathit{useconf} = 1)$[2] with the used configuration denoted by *useconf*. For φ, the actual functionality of the system is not relevant.

Because the input domain in the example system is unbounded φ cannot be model checked directly. However, we can abstract the system by mapping the infinite domain of input values to a finite abstract domain while preserving the

[2] $\mathit{input} \geq 50$ is understood as an abbreviation for $\bigvee_{v \geq 50} \mathit{input} = v$ and $a \rightarrow b$ as $\neg a \vee b$.

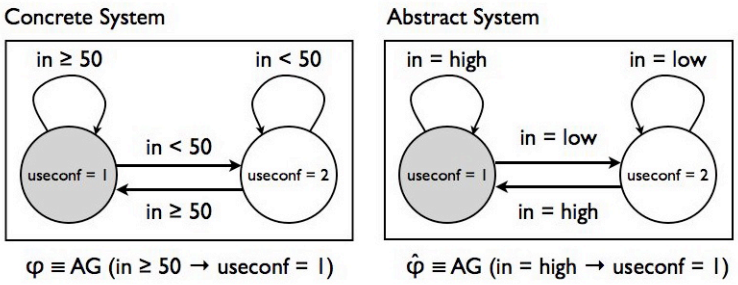

Fig. 2. Illustration of the example system

property φ. We choose the abstract domain $\widehat{Val} = \{low, high\}$. The abstraction function $h : Val \rightarrow \widehat{Val}$ is defined as $h(v) = low$ if $v < 50$ and $h(v) = high$ if $v \geq 50$. Then the abstract system will use configuration 1 if the input is $high$ and configuration 2 if it is low. Figure 2 depicts the concrete and abstract system as automata. The property φ is abstracted to $\hat{\varphi} \equiv \mathsf{AG}(input = high \rightarrow useconf = 1)$. With the approach presented in this paper we will be able to verify at runtime of the abstraction procedure that the abstraction preserves φ. This means that if we are able to verify $\hat{\varphi}$ for the abstract system φ also holds for the concrete.

3 Property Preservation by Simulation

In this section, we present the basis for the correctness criterion used in our translation validation approach. We use the fact that a property is preserved under abstraction if there is a consistent simulation between abstract and concrete system. Although our work originates from SAS in this presentation we will use general transition systems as SAS can be represented this way. Futhermore, it allows to extend the approach to a broader range of systems expressible as transition systems.

We need the notion of simulation between two transition systems to formulate a criterion for property preservation. A transition system $\mathcal{T}_{SAS}$ is simulated by an abstract transition system $\mathcal{T}_{\widehat{SAS}}$ if we can find a simulation relation $\mathcal{R}$ between the two sets of states such that firstly for all initial states of $\mathcal{T}_{SAS}$ there exists a related initial state in $\mathcal{T}_{\widehat{SAS}}$ and secondly that for any pair of related states with a transition in $\mathcal{T}_{SAS}$ there is also a transition in $\mathcal{T}_{\widehat{SAS}}$ such that the resulting states are related.

Definition 9 (Simulation between SAS). *Let $\mathcal{T}_{SAS}$ and $\mathcal{T}_{\widehat{SAS}}$ be two SAS systems. We say that $\mathcal{T}_{\widehat{SAS}}$ simulates $\mathcal{T}_{SAS}$, written as $\mathcal{T}_{SAS} \preceq \mathcal{T}_{\widehat{SAS}}$ iff there exists a simulation relation $\mathcal{R} \subseteq \Sigma \times \hat{\Sigma}$ such that*

1. *for all initial states $\sigma_0 \in Init$ there exists $\hat{\sigma}_0 \in \hat{Init}$ such that $\mathcal{R}(\sigma_0, \hat{\sigma}_0)$*

2. *for* $0 \leq i$ *and* $\sigma_i, \sigma_{i+1} \in \Sigma$ *and* $\hat{\sigma}_i \in \hat{\Sigma}$ *with* $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ *and* $\sigma_i \rightsquigarrow \sigma_{i+1}$ *the exists* $\hat{\sigma}_{i+1} \in \hat{\Sigma}$ *such that* $\hat{\sigma}_i \hat{\rightsquigarrow} \hat{\sigma}_{i+1}$ *and* $\mathcal{R}(\sigma_{i+1}, \hat{\sigma}_{i+1})$.

If a transition system $\mathcal{T}_{SAS}$ is simulated by $\mathcal{T}_{\widehat{SAS}}$ we can show that for each path in $\mathcal{T}_{SAS}$ there is a corresponding path in $\mathcal{T}_{\widehat{SAS}}$. This result is important for the preservation of temporal operators in a $\mathcal{L}_{SAS}$ 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 1 (Corresponding paths in $\mathcal{T}_{SAS}$ and $\mathcal{T}_{\widehat{SAS}}$). *Let* $\mathcal{T}_{SAS}$ *and* $\mathcal{T}_{\widehat{SAS}}$ *be two transition systems such that* $\mathcal{T}_{SAS} \preceq \mathcal{T}_{\widehat{SAS}}$ *with simulation relation* $\mathcal{R}$. *Then for every path* $\pi = \sigma_0 \sigma_1 \ldots \in \mathit{Paths}(\mathcal{T}_{SAS})$ *there exists a corresponding path* $\hat{\pi} = \hat{\sigma}_0 \hat{\sigma}_1 \ldots \in \mathit{Paths}(\mathcal{T}_{\widehat{SAS}})$ *such that* $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ *for all* $i \geq 0$.

Proof. By induction on length of path.

- Base case: For a path of length one we have $\pi = \sigma_0$ where $\sigma_0 \in \mathit{Init}$. Since $\mathcal{T}_{SAS} \preceq \mathcal{T}_{\widehat{SAS}}$ for all initial states there exists a corresponding initial state in the abstract system $\hat{\sigma}_0 \in \hat{\mathit{Init}}$ such that $\mathcal{R}(\sigma_0, \hat{\sigma}_0)$
- Induction hypothesis: For a path up to length n $\pi_n = \sigma_0 \sigma_1 \ldots \sigma_{n-1} \in \mathit{Paths}(\mathcal{T}_{SAS})$ there exists a corresponding path $\hat{\pi_n} = \hat{\sigma}_0 \hat{\sigma}_1 \ldots \hat{\sigma}_{n-1} \in \mathit{Paths}(\mathcal{T}_{\widehat{SAS}})$ such that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ for $i = 0, \ldots n-1$.
- Induction step $n \rightarrow n+1$: Let now $\pi_{n+1} = \sigma_0 \sigma_1 \ldots \sigma_n \in \mathit{Paths}(\mathcal{T}_{SAS})$. Then there exists a transition in $\mathcal{T}_{SAS}$ $\sigma_{n-1} \rightsquigarrow \sigma_n$. Furthermore, by induction hypothesis, we know that there exists a corresponding path $\hat{\pi_n} = \hat{\sigma}_0 \hat{\sigma}_1 \ldots \hat{\sigma}_{n-1} \in \mathit{Paths}(\mathcal{T}_{\widehat{SAS}})$ such that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ for $i = 0, \ldots n-1$. As $\mathcal{T}_{SAS} \preceq \mathcal{T}_{\widehat{SAS}}$ there also exists a transition $\hat{\sigma}_{n-1} \hat{\rightsquigarrow} \hat{\sigma}_n$ such that $\mathcal{R}(\sigma_n, \hat{\sigma}_n)$. Thus, there exists a corresponding path of length $(n+1)$ $\hat{\pi_{n+1}} = \hat{\sigma}_0 \hat{\sigma}_1 \ldots \hat{\sigma}_{n-1} \hat{\sigma}_n \in \mathit{Paths}(\mathcal{T}_{\widehat{SAS}})$. □

Now we are in the position to justify the criterion that allows to conclude $\mathcal{T}_{SAS} \models \varphi$ from $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ for φ and $\hat{\varphi}$ in $\mathsf{A}\mathcal{L}_{SAS}$. Liveness properties (using the existential path quantifier E) are typically lost under abstraction. The preservation result is based on simulation between the concrete and the abstract system and an additional consistency condition between concrete and abstract property. The consistency criterion intuitively expresses that the atomic propositions must be preserved under abstraction. In order to state the consistency condition we need a concretization function $\mathcal{C}$ that maps an abstract property $\hat{\varphi}$ to a corresponding property φ over the concrete system $\mathcal{T}_{SAS}$. This reflects the potentially different interpretations of variables and values in concrete and abstract system.

Definition 10 (Concretization function). *The concretization function*

$$\mathcal{C} : \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{\widehat{SAS}}] \rightarrow \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{SAS}]$$

is defined inductively on the structure of a $\mathcal{L}_{SAS}$ *formula by:*

- $\mathcal{C}(x = v) = f(x = v)$ *and* $\mathcal{C}(x = y) = f(x = y)$

- $\mathcal{C}(\neg\varphi) = \neg\,\mathcal{C}(\varphi)$
- $\mathcal{C}(\varphi_1 \wedge \varphi_2) = \mathcal{C}(\varphi_1) \wedge \mathcal{C}(\varphi_2)$
- $\mathcal{C}(\mathsf{X}\varphi) = \mathsf{X}\,\mathcal{C}(\varphi)$
- $\mathcal{C}(\varphi_1 \mathsf{U} \varphi_2) = \mathcal{C}(\varphi_1)\,\mathsf{U}\,\mathcal{C}(\varphi_2)$
- $\mathcal{C}(\mathsf{A}\varphi) = \mathsf{A}\,\mathcal{C}(\varphi)$

The function f maps atomic propositions of the abstract system to atomic propositions of the concrete system and is chosen dependant on the concrete abstraction performed.

The following theorem now summarizes under which conditions properties from the universal fragment of $\mathcal{L}_{SAS}$ are preserved under abstraction. The first two conditions state that the concrete system must be simulated by the abstract. The third condition denotes that atomic propositions must be preserved in the simulation relation. The fourth condition forces the concretisation of the abstract property to imply the original property.

Theorem 1 (Property-Preservation of ACTL*). *Let $\mathcal{T}_{SAS} = (\Sigma, Init, \rightsquigarrow)$ and $\widehat{\mathcal{T}_{SAS}} = (\widehat{\Sigma}, \widehat{Init}, \hat{\rightsquigarrow})$ be two transition systems, φ a $\mathsf{A}\mathcal{L}_{SAS}$ formula over $\mathcal{T}_{SAS}$ and $\hat{\varphi}$ an $\mathsf{A}\mathcal{L}_{SAS}$ formula over $\mathcal{T}_{\widehat{SAS}}$. Then it holds that*

$$\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi} \text{ implies } \mathcal{T}_{SAS} \models \varphi$$

if there exists a simulation relation $\mathcal{R} \subseteq \Sigma \times \widehat{\Sigma}$ and a concretization mapping $\mathcal{C} : \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{\widehat{SAS}}] \rightarrow \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{SAS}]$ such that the following conditions hold:

1. *Initial Simulation: for all $\sigma_0 \in Init$ there exists $\hat{\sigma}_0 \in \widehat{Init}$ such that $\mathcal{R}(\sigma_0, \hat{\sigma}_0)$*
2. *Step Simulation: for all $i \geq 0$, $\sigma_i, \sigma_{i+1} \in \Sigma$ and $\hat{\sigma}_i \in \widehat{\Sigma}$ with $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ and $\sigma_i \rightsquigarrow \sigma_{i+1}$ there exists $\hat{\sigma}_{i+1} \in \widehat{\Sigma}$ such that $\hat{\sigma}_i \hat{\rightsquigarrow} \hat{\sigma}_{i+1}$ and $\mathcal{R}(\sigma_{i+1}, \hat{\sigma}_{i+1})$.*
3. *Consistency: for all $\hat{a} \in Atoms(\hat{\varphi})$ if $\mathcal{R}(\sigma, \hat{\sigma})$ and $(\mathcal{T}_{\widehat{SAS}}, \hat{\sigma}) \models \hat{a}$ then $(\mathcal{T}_{SAS}, \sigma) \models \mathcal{C}(\hat{a})$*
4. *Implication: $\mathcal{T}_{SAS} \models \mathcal{C}(\hat{\varphi}) \rightarrow \varphi$.*

Proof. The proof proceeds in two steps: Firstly, we show that if conditions (1) - (3) are satisfied it holds that $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$. By condition (4), we will obtain the overall theorem. The proof of the first step is by induction of the structure of the formula $\hat{\varphi}$. The base case uses the consistency condition. The induction step for temporal operators and path quantifiers uses the path lemma.

For state formulas:

- $\hat{\varphi} = \hat{a}$ where $\hat{a}$ atomic: directly by consistency condition as for all initial states $\mathcal{R}(\sigma_0, \hat{\sigma_0})$.
- $\hat{\varphi} = \neg\varphi_1$: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $\mathcal{T}_{\widehat{SAS}} \not\models \varphi_1$. By induction hypothesis $\mathcal{T}_{SAS} \not\models \mathcal{C}(\varphi_1)$ and thus $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.
- $\hat{\varphi} = \varphi_1 \wedge \varphi_2$: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $\mathcal{T}_{\widehat{SAS}} \models \varphi_1$ and that $\mathcal{T}_{\widehat{SAS}} \models \varphi_2$. By induction hypothesis $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi_1)$ and $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi_2)$. This implies that $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.

– $\hat{\varphi} = \mathsf{A}\varphi_1$: Assume $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$. Then for all paths $\hat{\pi}_0 \in \mathit{Paths}(\mathcal{T}_{\widehat{SAS}})$ we have that $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi}_0) \models \varphi_1$. It holds that $(\mathcal{T}_{SAS}, \sigma_0) \models \mathcal{C}(\mathsf{A}\varphi_1)$ if for all paths $\pi_0 \in \mathit{Paths}(\mathcal{T}_{SAS})$ starting in σ_0 have the property that $(\mathcal{T}_{SAS}, \pi_0) \models \mathcal{C}(\varphi_1)$. By Path Lemma 1, for each path $\pi_0 \in \mathit{Paths}(\mathcal{T}_{SAS})$ there exists a corresponding paths $\hat{\pi}_0$. By assumption $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi}_0) \models \varphi_1$ and by induction hypothesis we have that $(\mathcal{T}_{SAS}, \pi_0) \models \mathcal{C}(\varphi_1)$ which yields $(\mathcal{T}_{SAS}, \sigma_0) \models \mathcal{C}(\mathsf{A}\varphi_1)$. Thus, we conclude $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.

For path formulae:

– $\hat{\varphi} = \varphi_1$, where $\hat{\varphi}$ is a path formula and φ is a state formula: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $(\mathcal{T}_{\widehat{SAS}}, \hat{\sigma}_0) \models \varphi_1$. By induction hypothesis $(\mathcal{T}_{SAS}, \sigma_0) \models \mathcal{C}(\varphi_1)$ and thus $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.
– $\hat{\varphi} = \neg\psi_1$ where ψ_1 path formula: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $\mathcal{T}_{\widehat{SAS}} \not\models \psi_1$. By induction hypothesis $\mathcal{T}_{SAS} \not\models \mathcal{C}(\psi_1)$ and thus $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.
– $\hat{\varphi} = \psi_1 \wedge \psi_2$ where ψ_1, ψ_2 path formulae: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $\mathcal{T}_{\widehat{SAS}} \models \psi_1$ and that $\mathcal{T}_{\widehat{SAS}} \models \psi_2$. By induction hypothesis $\mathcal{T}_{SAS} \models \mathcal{C}(\psi_1)$ and $\mathcal{T}_{SAS} \models \mathcal{C}(\psi_2)$. This implies that $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.
– $\hat{\varphi} = \mathsf{X}\,\psi$: $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ implies that $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi}_1) \models \psi_1$. It holds that $(\mathcal{T}_{SAS}, \pi_0) \models \mathcal{C}(\mathsf{X}\varphi_1)$ if $(\mathcal{T}_{SAS}, \pi_1) \models \mathcal{C}(\varphi_1)$. By Path Lemma 1, for each path $\pi_1 \in \mathit{Paths}(\mathcal{T}_{SAS})$ there exists a corresponding paths $\hat{\pi}_1$. If $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi}_1) \models \psi_1$ by induction hypothesis $(\mathcal{T}_{SAS}, \pi_1) \models \mathcal{C}(\varphi_1)$. This yields $\mathcal{T}_{SAS} \models \mathcal{C}(\varphi)$.
– $\hat{\varphi} = \psi_1 \,\mathsf{U}\, \psi_2$: As $\hat{\varphi}$ is a path formula $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi_0}) \models \psi_1 \,\mathsf{U}\, \psi_2$. By definition of the until operator there is a k such that $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi_k}) \models \psi_2$ and for all $0 \leq j < k$ $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi_j}) \models \psi_1$. It holds that $(\mathcal{T}_{SAS}, \pi_0) \models \mathcal{C}(\psi_1 \,\mathsf{U}\, \psi_2)$ if there is a k such that $\pi_k \models \psi_2$ and for all $0 \leq j < k$ $\pi_j \models \psi_1$. By Path Lemma 1, for each path $\pi \in \mathit{Paths}(\mathcal{T}_{SAS})$ there exists a corresponding paths $\hat{\pi}$ and in particular for each π_k there exists a corresponding $\hat{\pi}_k$. By assumption $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi_k}) \models \psi_2$ and for all $0 \leq j < k$ $(\mathcal{T}_{\widehat{SAS}}, \hat{\pi_j}) \models \psi_1$. By induction hypotheses $(\mathcal{T}_{SAS}, \pi_k) \models \mathcal{C}(\psi_2)$ and for all $0 \leq j < k$ $(\mathcal{T}_{SAS}, \pi_j) \models \mathcal{C}(\psi_1)$. This yields $\mathcal{T}_{SAS} \models \mathcal{C}(\psi_1 \,\mathsf{U}\, \psi_2)$ □

In order to see that existentially quantified path formulae are not preserved under abstraction we construct a counter example: We have a concrete system $\mathcal{T}_S = (\Sigma, \mathit{Init}, \rightsquigarrow)$ where $\Sigma = \{\sigma_1, \sigma_2\}$, $\mathit{Init} = \{\sigma_1\}$ and $\rightsquigarrow = \{(\sigma_1, \sigma_2), (\sigma_2, \sigma_2)\}$. The set of paths of $\mathcal{T}_S$ is given by $\mathit{Paths}(\mathcal{T}_S) = \{\pi = \sigma_1\sigma_2 \ldots\}$. The abstract system is defined by $\mathcal{T}_{S'} = (\Sigma', \mathit{Init}', \rightsquigarrow')$ where $\Sigma' = \{\sigma_1', \sigma_2', \sigma_3'\}$, $\mathit{Init}' = \{\sigma_1'\}$ and $\rightsquigarrow' = \{(\sigma_1', \sigma_2'), (\sigma_2', \sigma_2'), (\sigma_1', \sigma_3'), (\sigma_3', \sigma_3')\}$. The set of paths is given by $\mathit{Paths}(\mathcal{T}_{S'}) = \{\pi^{1'} = \sigma_1'\sigma_2' \ldots, \pi^{2'} = \sigma_1'\sigma_3' \ldots\}$. The two systems are depicted in Figure 3 with the valid atomic propositions in each state.

The concrete system is simulated by the abstract $\mathcal{T}_S \preceq \mathcal{T}_S'$ 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 $\mathcal{T}_S$ we have $\mathcal{R}(\sigma_1, \sigma_1')$. For the transition $\sigma_1 \rightsquigarrow \sigma_2$, we have $\mathcal{R}(\sigma_1, \sigma_1')$ and there is σ_2' such that $\sigma_1' \rightsquigarrow' \sigma_2'$ and $\mathcal{R}(\sigma_2, \sigma_2')$. For the transition $\sigma_2 \rightsquigarrow \sigma_2$, we have $\mathcal{R}(\sigma_2, \sigma_2')$ and there is σ_2' such that $\sigma_2' \rightsquigarrow' \sigma_2'$ and $\mathcal{R}(\sigma_2, \sigma_2')$.

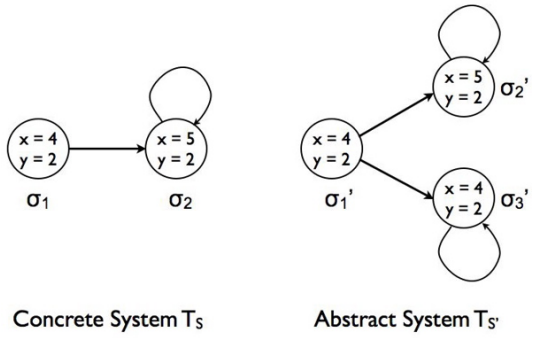

Fig. 3. Graphical Representation of Example Systems

With respect to property preservation, the following can be observed: As the atomic propositions of both systems are the same, the concretization function is the identity function. The abstract system satisfies the properties $\mathsf{AG}(y = 2)$ and $\mathsf{EF}(x = 4)$. However, for the concrete system $\mathcal{T}_S \models \mathsf{AG}(y = 2)$ and $\mathcal{T}_S \not\models \mathsf{EF}(x = 4)$. The reason for this lies in the corresponding paths. From Lemma 1, we know that for each path in $\mathcal{T}_S$ there is a corresponding path in $\mathcal{T}_{S'}$. In this example, π corresponds to $\pi^{1'}$. As $\mathsf{AG}(y = 2)$ holds on all paths of $\mathcal{T}_{S'}$ it in particular holds on the corresponding path $\pi^{1'}$. Hence, $\mathsf{AG}(y = 2)$ also holds in the concrete system. For the existential path property $\mathsf{EF}(x = 4)$, we have that $(\mathcal{T}_{S'}, \pi^{2'}) \models \mathsf{F}(x = 4)$. However, there is no path in $\mathcal{T}_S$ that corresponds to $\pi^{2'}$. Hence, $\mathsf{EF}(x = 4)$ does not hold in the concrete system.

Theorem 1 constitutes the necessary conditions for the correctness criterion in our translation validation approach. It differs from other approaches using property-preservation by simulation [7,2,10] therein that states of the underlying system model are characterized by variable assignments and that atomic propositions in the applied logic are constraints over these assignments. This requires a concretization function but simplifies to work with systems where states are described by valuations of variables such as in SAS. The same result can also be proved by giving a transformation of SAS to labeled Kripke structures and applying results from [7]. This construction can be found in Appendix A.

Furthermore, Theorem 1 is formulated in a very general fashion that allows to instantiate it with a number of different kinds of abstractions. In this direction, it can be used to justify the domain abstraction approach proposed in [6]. The concrete transition system is defined over a concrete data domain D, either very large or infinite. Thus, the system can only be model checked very inefficiently if at all. So the concrete domain is abstracted to an abstract domain $\widehat{D}$ by an homomorphic abstraction function $h : D \to \widehat{D}$. In order to prove that a property φ is preserved under this form of domain abstraction we have to establish a simulation relation between Σ and $\widehat{\Sigma}$ satisfying the conditions of Theorem 1. This is the relation defined by $(\sigma, \hat{\sigma}) \in \mathcal{R}$ if $\hat{\sigma}(x) = h(\sigma(x))$ for all $x \in Var$. The

concretization function $\mathcal{C}$ for an atomic proposition maps the formula $x = \hat{v}$ for $x \in \mathit{Var}$ and $\hat{v} \in \widehat{D}$ to the disjunction over all concrete values that are mapped to the abstract value $\hat{v}$, i.e

$$\mathcal{C}(x = \hat{v}) = \bigvee_{h(v)=\hat{v}} (x = v)$$

The concretization function is further defined by induction on the formula structure preserving it. This form of abstraction is also applied in the example of Section 2.
Another abstraction procedure that can be handled with Theorem 1 is the omission of variables that are not necessary for the considered property, similar to dead code elimination in compiler optimization. Here, the abstract system $\mathcal{T}_{\widehat{SAS}}$ only contains a subset of the variables of $\mathcal{T}_{SAS}$, i.e. $\widehat{\mathit{Var}} \subseteq \mathit{Var}$ while the rest of the system remains the same. The simulation relation between two states can be defined as $\mathcal{R}(\sigma, \hat{\sigma})$ iff $\sigma(x) = \hat{\sigma}(x)$ for all $x \in \widehat{\mathit{Var}}$. The concretization function is simply the identity function since the interpretation of the atomic propositions does not change if the abstraction is carried out correctly. Beside these two abstraction procedures we aim at extending our work to more complicated and powerful abstractions (cf. Future Work in Section 7).

4 The Translation Validation Infrastructure

In this section, we describe the different steps that have to be accomplished in order to verify a system abstraction correct in Isabelle/HOL[15]. Firstly, we have to generate an Isabelle/HOL description of both the concrete and the abstract system. Secondly, we have certain properties to be preserved in the abstracted system. These properties have to be represented in Isabelle as well. This comprises the formalization of atomic propositions for both concrete and abstract system and the formalization of the concretization function from Definition 10. Thirdly, we have to formalize a criterion stating the correctness of an abstraction in Isabelle corresponding to the conditions of Theorem 1. Finally, we need a proof script that proves that the concrete and abstract system description fulfill the correctness criterion. Note that instead of the more general transition relation in Theorem 1 we use explicit state transition functions in the Isabelle formalization corresponding to the SAS system specification (cp. Definitions 1 and 2).

4.1 Representing Systems in Isabelle

In our implementation, Isabelle representations of concrete and abstract system are generated right before and after a run of the abstraction procedure. Concrete and abstract systems are represented using the same datatypes. We use a shallow embedding of our system description language into the Isabelle/HOL theorem prover. This means that we formalize the semantics of a system directly

within Isabelle's Higher Order Logic constructs. Since the semantics is basically defined via state transition functions we use Isabelle syntax to directly encode these functions. In contrast, a deep embedding would require to formalize the syntax of the system description language in Isabelle[3] and define a semantics on top of the syntactical elements. Some of the SAS specifications are not entirely formulated as executable programs. Instead the functionality of a single configuration may only be characterized via pre- and postconditions. Due to the more abstract nature of shallow embeddings such issues are much easier to deal with in our approach. We also believe that we can adopt to changes in the underlying datatypes faster if we do not formalize them in Isabelle directly.

Thus, to generate Isabelle system semantics representations we need to convert a SAS description directly into Isabelle (state transition) functions. Furthermore, we generate datatypes representing system states to serve as arguments for these functions. Due to the finite number of variables in each system we encode states as tuples of values rather than in a mapping function. This simplifies conducting the proofs. Variable references are encoded as selectors to such tuples. We do not distinguish between different kinds of variables (like e.g. input, output, adaptation and ordinary variables in Definitions 1 and 2) in the state encoding. Input is implicitly regarded as a stream of input elements. One element after the other is consumed during system execution. Initial states are encoded as functions assigning initial values to an arbitrary state.

A SAS module is divided into an adaptation aspect for adaptive behavior and functional configurations. Before evaluating the functionality of a configuration the adaptive part is evaluated (cp. Definition 1: $\mathsf{adapt\_next\_state}$ and $\mathsf{adapt\_next\_out}$). The actual functionality of a configuration ($\mathsf{next\_state}_j$ and $\mathsf{next\_out}_j$) is selected using a guard formula. In our semantics framework we encode this behavior by evaluating the Isabelle representation for $\mathsf{adapt\_next\_state}$ and $\mathsf{adapt\_next\_out}$ first. Then we make a case distinction on the guard formulas (several if-clauses) selecting the appropriate Isabelle representation for the configuration functions $\mathsf{next\_state}_j$ and $\mathsf{next\_out}_j$ to be evaluated. Connectors between different modules are encoded as copy operations between variables of different modules within the state transition functions.

The generation of the system state transition function is done using a visitor pattern on the datatypes representing the input systems. While visiting parts of the system description corresponding parts for the state transition function are emitted in Isabelle/HOL syntax. These parts are composed to a large state transition function representing a system's semantics within Isabelle/HOL.

In systems with more than one module, we generate Isabelle representations for each module. Since we deal with synchronous systems, modules do not affect each other during a single transition. Hence, we can evaluate the modules' state transition functions one after the other. Evaluation order does not matter. An addition to this, we generate Isabelle representations for the connections between modules which are functions themselves. All these functions are composed into a

[3] see e.g. [20] for a comparison between deep and shallow embedding in an Isabelle/HOL environment

single state transition function representing a system's semantics. This technique works for concrete and abstract systems equally well.

A generated example Isabelle representation of a module is depicted in Figures 4, 5 and 6. It describes a wheel speed sensor. Figure 4 shows the datatype the module's state transtion function opperates with. It defines a record which is realized in Isabelle as a tuple representation with named components. The initialization function for formalizing initial states is shown in Figure 5. The $\%f.f$ is an Isabelle notation for a lambda-style function definition. In this case it defines a function that takes an argument f and returns the f thus defining an identity function. With this function, we define an arbitrary initial state. Figure 6 shows the state transition function for this module. It comprises two configurations: `V_WheelCalculation` where the vehicle speed is calculated from the speed of the wheels and the default configuration `Off`. In the first configuration `V_WheelCalculation` also the values for the two adaptive output variables `wheel_revolution_measuredRR_7_available_resolution` and `v_wheelRR_7_calculated_slipProbability` specifying parameters for this configuration are computed. The o operator composes different independently generated functions to the state transition function.The complete system description containing this module consists of 24 modules and implements a traction control system used as case study.

```
record v_wheelRR_params =
   aux :: int
   useCONFIG :: int
   wheel_revolution_measuredRR_7::"int"
   wheel_revolution_measuredRR_7_quality::"int"
   wheel_revolution_measuredRR_7_available_resolution::"int"
   v_wheelRR_7::"int"
   v_wheelRR_7_quality::"int"
   v_wheelRR_7_calculated_slipProbability::"int"
   v_wheelRR_7_calculated_resolution::"int"
```

Fig. 4. Generated Datatype for State Representation

```
constdefs init_v_wheelRR:: " v_wheelRR_params => v_wheelRR_params"
 "init_v_wheelRR == ( % f. (f  ))"
```

Fig. 5. Generated Initialization Function

```
constdefs v_wheelRR :: " v_wheelRR_params => v_wheelRR_params => v_wheelRR_params "
"v_wheelRR ==  ( % g . (
if (
 (  wheel_revolution_measuredRR_7_quality g )  =
       wheel_revolution_measuredRR_7_quality_available)
then (
      (* Priority 1 *)
      ( % f. f (|useCONFIG := v_wheelRR__V_WheelCalculation|) )
   o
      ( (% f . f (| v_wheelRR_7_quality  := v_wheelRR_7_quality_calculated|) ) o
(*Begin Cascade *) (
     if (
      (  wheel_revolution_measuredRR_7_available_resolution g )  =
            wheel_revolution_measuredRR_7_available_resolution_low )
     then
     (
       ( (% f . f (| v_wheelRR_7_calculated_resolution  :=
                        v_wheelRR_7_calculated_resolution_low|) ) )
     )
     else if (
      (  wheel_revolution_measuredRR_7_available_resolution g )  =
            wheel_revolution_measuredRR_7_available_resolution_middle )
     then
     (
      ( (% f . f (| v_wheelRR_7_calculated_resolution  :=
                        v_wheelRR_7_calculated_resolution_middle|) ) )
     )
     else if (
      (  wheel_revolution_measuredRR_7_available_resolution g )  =
            wheel_revolution_measuredRR_7_available_resolution_high )
     then
     (
      ( (% f . f (| v_wheelRR_7_calculated_resolution  :=
                        v_wheelRR_7_calculated_resolution_high|) ) )
     )
     else
     (
      ( ( % f . f ) )
     )
   )
(*End Cascade *)
o
   (% f . f (| v_wheelRR_7_calculated_slipProbability  :=
                  v_wheelRR_7_calculated_slipProbability_none|) )
      )
)   (* end conf *)
else
(
      ( % f. f (|useCONFIG := v_wheelRR__Off|) )
   o
      ( (% f . f (| v_wheelRR_7_quality  := v_wheelRR_7_quality_unavailable|) ) )
)
) )"
```

Fig. 6. Generated State Transition Function

4.2 System Properties and Concretization Function

System properties are represented using the specification logic from Definition 7. Due to its definition as context free grammar a deep embedding using Isabelle/HOL's datatype constructors is an adequate choice. We leave the set of atoms from Definition 7 parametric to the syntax definition of the specification logic in Isabelle. The deep embedding requires to define a semantics on top of the datastructures representing specification logic formulas. This is done defining a function taking a system, a formula representing a property in the specification logic and a function stating whether a certain atomic property is fulfilled at a given state of the system. The defined semantics function returns a truth value indicating whether the system fulfills the given propertiy. Since the formalized logic is parameterized with a set of atoms and a function interpreting atoms in a given state it can be used for both concrete and abstract system properties using different instantiations.

The concretization function between atomic propositions of the abstract system and atomic propositions of the concrete system is defined in Isabelle/HOL corresponding to Definition 10. In most cases this function may be automatically generated from a given simulation relation and vice versa.

4.3 Formalizing Abstraction Correctness in Isabelle

For proving that an abstraction is valid we need a formalization of property preservation in Isabelle/HOL. Such a formalized correctness criterion (Figure 7) has to fulfill the conditions stated in Theorem 1. The first two conditions (in both the theorem and the figure) correspond to the simulation between the two systems. These first two conditions are formalized once for all systems. With a slight generalization they can also be applied for the verification of compiler optimization phases (cf. [4,11]).

```
constdefs systemequivalence ::
   (state => state) => (state' => state') => state => state' =>
     (state => state' => bool) => concprop => absprop => concfun => bool
   "systemequivalence nextstate nextstate' s0 s0' R c a C ==
    R s0 s0' &
    ALL s s'. R s s' --> R (next s) (next s') &
    consistency(R,C) & implies(C (a),c)"
```

Fig. 7. Correctness Criterion

The third condition in Theorem 1 requires that the simulation relation preserves consistency. We are free to chose the notion of consistency by instantiating the concretization function $\mathcal{C}$. However, we have to ensure that the fourth condition of Theorem 1 still holds. In order to establish condition 4 in Theorem 1, one

can formulate properties to be checked in terms of the abstractions in the first place. In our case studies, however, properties are usually formulated in terms of the concrete system. Hence, one has to verify that the concretization of the abstract property implies the concrete property. Note that conditions (1) and (2) do not depend on concrete properties to be checked. Conditions (3) and (4) are independent of a concrete system. Only the first three conditions depend on the simulation relation. This allows to reprove only parts of the overall correctness proof if we have minor changes in either the system description or the properties that shall be preserved during abstrations.We found it especially usefull to prove the condition (4) independent of the first three conditions.

Figure 8 shows a small extract from a typical simulation relation for a domain abstraction. It takes two states `A` and `B` of concrete and abstract system, respectively, and ensures that whenever the variable `in1` in the concrete system has a value less than 50 then the value of `in1` in the simulating abstract system must be `low`. In the complete simulation relation for a system, we encode a condition for every variable abstraction being performed. In contrast to this fragment of a simulation relation for domain abstractions the simulation relation for omission of variables is even simpler. Here, no condition is put on an omitted variable in the relation.

```
constdefs inputequivalence :: "S1 => S2 => bool"
"inputequiv A B == ( ( (in1 A = low) = (in1 B <= 50) ) & ..."
```

Fig. 8. Simulation Relation

The simulation relation for a concrete system can be generated by the abstraction procedure or adjusted by hand. It reflects the performed abstractions. Note that the concretization function $\mathcal{C}$ in Theorem 1 directly corresponds to the simulation relation. In our example simulation relation, the abstract value on the left side of the equation is the argument of $\mathcal{C}$ whereas the concrete value on the right side refers to the result of the concretization.

4.4 Proving Abstractions Correct

To conduct the correctness proof we still need a proof script. In our current implementation we first prove additional lemmata implying the actual correctness criterion. The `simu_step_helper` lemma for abstraction of variable domains and omission of variables as well as its proof is depicted in Figure 9. It is generic for many systems. The formalization of the lemma is shown in the first line. The rest is the proof script computing the proof for this lemma. A proof script can be considered as a kind of program that tells the theorem prover how to conduct a proof. It comprises the application of several tactics (`apply`) which can be regarded as subprograms in the proving process. In the proving process the theorem prover symbolically evaluates state transition functions (`M1`,`M1'`) on

```
lemma simu_step_helper: "(funequiv A B) & (inputequiv A B) & (funequiv' A B)
                                  --> (funequiv (M1' A A) (M1 B B))"
apply (clarify, unfold funequiv_def inputequiv_def, clarify  )
apply (unfold M1_def, unfold M1'_def)
apply (erule subst)+
apply (unfold funequiv'_def funequiv_def inputequiv_def )
apply clarify
apply (rule conjI, simp) +
apply simp
done
```

Fig. 9. Proof Script

symbolic states. These symbolic states are specified by their relation to each other. The theorem prover checks that the relation between the states still holds after the evaluation of the transition functions. The predicates `funequiv` and `inputequiv` together imply system equivalence and in general do highly depend on the chosen simulation relation. In the scenarios examined so far, however, we have developed a single highly generic proof script that proves the correctness in all scenarios. For more complicated scenarios the proof script might need adaptation. This was the case in the original compiler scenario where adaptations could be done fully automatically [4].

5 Evaluation of our Framework

The AMOR (Abstract and MOdular verifieR) tool prototypically implements the technique proposed in this paper for domain abstractions and omitting variables. We have successfully applied it in several case studies in the context of the EVAS project [1] and proved that interesting system properties were preserved by abstractions. Our largest example with domain abstractions contained amongst others 39 variables with infinite domains. We generated Isabelle representations for systems consisting of up to 2600 lines of Isabelle code. In some of these scenarios, model checking was not possible without abstractions. Thus, our technique bridges a gap in the verification process between a system model representation in a modelling environment (used e.g. for code generation) and an input representation for verification tools.

6 Related Work

While previously correctness of abstractions was established by showing soundness for all possible systems, for instance in abstract interpretation based approaches [8,9], our technique proves an abstraction correct for a specific system and property to be verified. In this direction, we adopted the notion of translation validation [16,21] to correctness of system abstractions. Translation validation

focuses on guaranteeing correctness of compiler runs. After a compiler has translated a source 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 abstraction mechanism, the source program by the original system and the target program by the abstract system. Isabelle/HOL[15] serves as checker in our case. In the original translation validation approach[16] 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 was 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. This approach is known as credible compilation [18] or certifying compilation [11]. Translation validation in general is not limited to simulation based correctness criteria. However, also for compiler and transformation algorithm verification simulation based correctness criteria can be used (see e.g. [3] for work with a similar Isabelle formalization of simulation).

Simulation for program correctness was originally introduced by [12]. Since, property preservation by simulation has been studied also for different fragments of CTL* and the μ-calculus. The authors in [7,2,10] use Kripke structures as their underlying system model 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 the same. However, this complicates the treatment of systems defined by valuations of variables such as SAS. In [6], the authors use a system model similar to ours, but this work is restricted to data domain abstraction while our technique can be applied for different abstraction mechanisms. Abstract interpretation based simulations as used in [2,10] are also less general than generic simulation relations considered here.

Related work on formalizing state transition systems (I/O automata) in Isabelle is done by Müller and Nipkow in [14,13]. Reasoning about combining model checking and theorem proving techniques is done in [14]. In [13], a methodology is presented to prove abstractions correct with respect to preservation of formulas. They use a notion of correctness based on output trace inclusion which is equivalent to simulation for abstractions. Correctness with respect to trace inclusion is proved for a class of abstraction functions. In contrast, we provide a methodology to prove that concrete abstractions fall into a well established class of correct abstractions whereas their work concentrates on establishing such a class.

7 Conclusion

In this paper, we presented a technique for proving correctness of system abstractions using a translation validation approach. Based on property-preservation by simulation we formalized a correctness criterion in Isabelle. With the help of generic proof scripts we are able to verify abstractions correct at runtime of the

abstraction procedure. Our technique was successfully applied in various case studies verifying data domain abstractions and omission of variables.

For future work, we want to apply our technique to further and more complex abstraction procedures. In particular, we want to focus on abstractions of hierarchical systems where simple stepwise simulation relations will no longer be sufficient. Additionally, we are planning to investigate the interplay between modularisation and abstraction in order to further reduce verification effort.

References

1. R. Adler, I. Schaefer, T. Schuele, and E. Vecchie. From model-based design to formal verification of adaptive embedded systems. In *9th International Conference on Formal Engineering Methods (ICFEM 2007), Boca Raton, FL*, LNCS. Springer, November 2007.
2. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. of CAV '92*, pages 260–273, London, UK, 1993. Springer-Verlag.
3. J. O. Blech, L. Gesellensetter, and S. Glesner. Formal Verification of Dead Code Elimination in Isabelle/HOL. In *Proc. of SEFM*, pages 200–209, September 2005.
4. J. O. Blech and A. Poetzsch-Heffter. A certifying code generation phase. In *Proc. of COCV 2007, Braga, Portugal*, ENTCS, March 2007.
5. J. O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation validation for system abstractions. In *7th Workshop on Runtime Verification (RV'07), Vancouver, Canada*, March 2007.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, September 1994.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252. ACM Press, January 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL*, pages 269–282. ACM Press, January 1979.
10. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
11. M. J. Gawkowski, J. O. Blech, and A. Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, TU Kaiserslautern, November 2006.
12. R. Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI*, pages 481–489, 1971.
13. O. Müller. I/O Automata and Beyond: Temporal Logic and Abstractions in Isabelle. In *Theorem Proving in Higher Order Logics*, LNCS. Springer, 1998.
14. O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
16. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, volume 1384 of *LNCS*. Springer, 1998.
17. A. Poetzsch-Heffter and M. J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.

18. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
19. I. Schaefer and A. Poetzsch-Heffter. Using Abstraction in Modular Verification of Synchronous Adaptive Systems. In *Proc. of "Workshop on Trustworthy Software", Saarbrücken, Germany, May 18-19*, 2006.
20. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Theorem Proving in Higher Order Logics*, LNCS. Springer, 2004.
21. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.

A Proof of Property Preservation with Kripke Structures

Another way to justify Theorem 1 is by reducing SAS to Kripke structures (labeled transition systems) and applying the results from [7]. So let us first recall the results used in [7]. Systems are modelled by Kripke structures as transition systems where states are labeled by atomic propositions of the logic. The labeling denotes the atomic propositions valid in a state.

Definition 11. *A Kripke structure $M = (AP, S, S_0, R, L)$ is defined as a five tuple where*

- *AP is the set of atomic propositions*
- *S the set of states and S_0 the initial states*
- *$R \subseteq S \times S$ the transition relation*
- *$L : S \to \mathcal{P}(AP)$ the labeling function assigning sets of atomic propositions to states.*

The syntax of a CTL* formula φ over M is defined analogously to Definition 7. However, the atoms used in this formulas are taken from the set of atomic propositions AP. The satisfiablitiy of CTL* formulas over Kripke structures is defined analogosly to Defintion 8 with a difference for the atomic propostions. An atomic proposition $p \in AP$ is true in a state s if it is contained in the set of labels: For a state $s \in S$ and an atomic proposition $p \in AP$ it holds that $(M, s) \models p$ iff $p \in L(s)$.

Simulation between two Kripke structures is defined in a similar manner as for SAS. However, besides requirements on the transition relation also conditions are imposed on the labeling of states. Firstly, the set of atomic propositions of the original system must be a subset of the atomic propositions of the abstracted system. Secondly, the labeling of two states included in the simulation relation must coincide on the common set of atomic propositions.

Definition 12 (Simulation Relation). *Given two Kripke structures M and M' with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a simulation relation between M and M' iff for all states $s \in S$ and $s' \in S'$ with $H(s, s')$ the following conditions holds:*

1. *$L(s) \cap AP' = L'(s')$*
2. *For every state s_1 such that $R(s, s_1)$, there is a state s'_1 with property that $R'(s', s'_1)$ and $H(s_1, s'_1)$.*

Adding a condition on the initial states of the two considered systems gives the notion of simulation between two Kripke structures.

Definition 13 (Simulation between Kripke structures). *M' simulates M ($M \preceq M'$) if there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s'_0 in M' for which $H(s_0, s'_0)$ holds.*

The following theorem states the property preservation result for universal CTL* formulae based on simulation from [7].

Theorem 2 (Property Preservation on Kripke structures [7]). *Let $M \preceq M'$. Then for every ACTL* formula f with atomic propositions in AP', $M' \models f$ implies $M \models f$.*

Our proof for property preservation by simulation for SAS now proceeds as follows: For a given concrete system $\mathcal{T}_{SAS}$ and an abstract system $\mathcal{T}_{\widehat{SAS}}$ and a property $\hat{\varphi}$ over $\mathcal{T}_{\widehat{SAS}}$ and a concretization $\mathcal{C}(\hat{\varphi})$ of the abstract property over $\mathcal{T}_{SAS}$ we perform the following steps:

1. We define the atomic propositions AP of a Kripke structure as pairs consisting of an atom of the abstract property $\hat{\varphi}$ and its concretisation via $\mathcal{C}$. We call this set of atomic propositions $AP(\hat{\varphi})$.
2. We show how to map the property $\hat{\varphi}$ to a property f using pairs of atomic propositions from $AP(\hat{\varphi})$ as atomic propositions.
3. We show how to map the concrete SAS $\mathcal{T}_{SAS}$ and the abstract SAS $\mathcal{T}_{\widehat{SAS}}$ to a Kripke structures M and M' where the states are labeled with atomic propositions from $AP(\hat{\varphi})$
4. We show that if the abstract system $\mathcal{T}_{\widehat{SAS}}$ satisfies the abstract property $\hat{\varphi}$ also the Kripke structure M' satisfies the property f.
5. We show that if the Kripke structure M satisfies the property f the concrete system $\mathcal{T}_{SAS}$ satisfies the concretized property $\mathcal{C}(\hat{\varphi})$.
6. We show that if the concrete system $\mathcal{T}_{SAS}$ and the abstract system $\mathcal{T}_{\widehat{SAS}}$ consistently simulate each other with respect to the atomic propositions of $\hat{\varphi}$ also M and M' simulate each other. Consistent simulation means that atomic propositions are preserved by the simulation relation. This is important in order to conclude simulation for Kripke structures which requires coinciding labels of similar states.
7. Using the result from [7] and our previous considerations we can prove the overall theorem.

We start by defining the atomic propositions. The concretization function maps a property over the abstract system $\mathcal{T}_{\widehat{SAS}}$ defined in terms of $\widehat{Var}$ and $\widehat{Val}$ to a property φ over the concrete system defined in terms of Var and Val, i.e. $\mathcal{C} : \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{\widehat{SAS}}] \rightarrow \mathsf{A}\mathcal{L}_{SAS}CTL*[\mathcal{T}_{SAS}]$ as defined in Definition 10. Those formulas are equivalent in the sense of the abstraction. The following definition formalizes the operator $Atoms$ returning the set of atoms of a formula $\varphi \in \mathsf{A}\mathcal{L}_{SAS}$.

Definition 14 (Atoms). *Let φ be a formula form $\mathsf{A}\mathcal{L}_{SAS}$ over a set of variables Var and values Val. We define the $Atoms(\varphi)$ on the structure of φ as*

- $Atoms(a) = a$ *, for a an Atom of* $\mathcal{L}_{SAS}$
- $Atoms(\neg\varphi) = Atoms(\varphi)$
- $Atoms(\varphi_1 \wedge \varphi_2) = Atoms(\varphi_1) \cup Atoms(\varphi_2)$
- $Atoms(\mathsf{X}\varphi) = Atoms(\varphi)$
- $Atoms(\varphi_1 \mathsf{U} \varphi_2) = Atoms(\varphi_1) \cup Atoms(\varphi_2)$
- $Atoms(\mathsf{A}\varphi) = Atoms(\varphi)$

The set of atomic propositions $AP(\hat{\varphi})$ generated from a property $\hat{\varphi}$ is now defined as the set of pairs consisting of an atom from the abstract property $\hat{\varphi}$ and its concretization via $\mathcal{C}$.

Definition 15 (Definition of atomic propositions as pairs). *Let $\hat{\varphi}$ be a CTL* property over the abstract system $\mathcal{T}_{\widehat{SAS}}$ and $\mathcal{C} : \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{\widehat{SAS}}] \to \mathsf{A}\mathcal{L}_{SAS}[\mathcal{T}_{SAS}]$ a concretization function for properties over $\mathcal{T}_{\widehat{SAS}}$ to properties over $\mathcal{T}_{SAS}$. The set of atomic propositions $AP(\hat{\varphi})$ induced by $\hat{\varphi}$ is defined as*

$$AP(\hat{\varphi}) = \bigcup_{\hat{a} \in Atoms(\hat{\varphi})} (\hat{a}, \mathcal{C}(\hat{a}))$$

As an example for this construction of pair of corresponding atomic propositions consider data domain abstraction where a set of concrete values $\mathcal{V} \subseteq Val$ is mapped to an abstract value $\hat{v} \in \widehat{Val}$. An abstract atomic proposition is for instance $\hat{\varphi} \equiv (x = \hat{v})$. The concretization is $\mathcal{C}(x = \hat{v}) = \bigvee_{v \in \mathcal{V}} (x = v)$. The set of atomic propositions induced by $\hat{\varphi}$ is then $AP(\hat{\varphi}) = \{(x = \hat{v}, \bigvee_{v \in \mathcal{V}} x = v)\}$.

We can now define the abstract property $\hat{\varphi}$ in terms of the atomic propositions defined by $AP(\hat{\varphi})$ by substituting each atomic proposition by the corresponding pair of this atomic proposition and its concretization.

Definition 16 (Construction of f over $AP(\hat{\varphi})$ from $\hat{\varphi}$). *Let $\hat{\varphi}$ be a $\mathcal{L}_{SAS}$ formula and let $AP(\hat{\varphi})$ be the set of atomic propositions consisting of pairs of atomic propositions and their conretization. The mapping trans transforms the property $\hat{\varphi}$ into a property f over $AP(\hat{\varphi})$ as follows:*

- $trans(a) = (a, \mathcal{C}(a))$ *, for* $a \in Atoms(\hat{\varphi})$.
- $trans(\neg\varphi) = \neg\, trans(\varphi)$
- $trans(\varphi_1 \wedge \varphi_2) = trans(\varphi_1) \wedge trans(\varphi_2)$
- $trans(\mathsf{X}\varphi) = \mathsf{X}\, trans(\varphi)$
- $trans(\varphi_1 \mathsf{U} \varphi_2) = trans(\varphi_1)\, \mathsf{U}\, trans(\varphi_2)$
- $trans(\mathsf{A}\varphi) = \mathsf{A}\, trans(\varphi)$

A SAS transition system $\mathcal{T}$ can be encoded as a labeled Kripke structure over an appropriately chosen set of atomic propostions. For the property preservation result we instantiate the set of atomic propositions with the set of pairs of corresponding propositions $AP(\hat{\varphi})$ induced by the abstract property $\hat{\varphi}$. However, we have to define, when a state σ of $\mathcal{T}$ satisfies such a pair of propositions. For this purpose, we introduce the satisfaction relation $\models_\times$.

Definition 17. *A state $\sigma : Var \to Val$ of a transition system $\mathcal{T}$ satisfies a pair of atomic prospositions $(a, \mathcal{C}(a))$, i.e. $\sigma \models_\times (a, \mathcal{C}(a))$ iff either a is defined on σ and $\sigma \models a$ or $\mathcal{C}(a)$ is defined on σ and $\sigma \models a$ where for variables x, y and values v an atomic proposition $(x = v)$ is defined on σ if $x \in Dom(\sigma)$ and $v \in Val$ and an atomic proposition $(x = y)$ is defined on σ if $x, y \in Dom(\sigma)$.*

We can now encode SAS transition systems as labeled Kripke structures over $AP(\hat{\varphi})$ for an abstract property $\hat{\varphi}$ as follows:

Definition 18 (Encoding of SAS as Kripke structures). *We define the encoding trans of a SAS transition system* $\mathcal{T}_{SAS} = (\Sigma, Init, \rightsquigarrow)$ *as a Kripke structure* $trans(\mathcal{T}_{SAS}) = (AP, S, S_0, R, L)$ *over a set of atomic propositions* $AP(\hat{\varphi})$ *by*

- $S = \Sigma$
- $S_0 = Init$
- $R = \rightsquigarrow$
- *for* $s \in S$: $L(s) = \{p \in AP(\hat{\varphi})\}$ *iff* $(\mathcal{T}_{SAS}, s) \models_{\times} p$

The following lemma states that if the abstract system $\mathcal{T}_{\widehat{SAS}}$ satisfies the abstract property $\hat{\varphi}$ the transformed system $M' = trans(\mathcal{T}_{\widehat{SAS}})$ satisfies f in terms of a Kripke structure.

Lemma 2. *Let* $M' = trans(\mathcal{T}_{\widehat{SAS}})$ *and* $f = trans(\hat{\varphi})$. *If* $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$ *then* $M' \models f$

Proof. As the structure of the formula is preserved in the translation from $\hat{\varphi}$ to f it suffices to consider satisfaction of atomic propositions. For each atomic proposition in $a \in Atoms(\hat{\varphi})$ it can easily be seen by construction of M' that if a state $\hat{\sigma} \models a$ the corresponding pair of atomic propositions is contained in the labeling of this state, i.e. $(a, \mathcal{C}(a)) \in L(\hat{\sigma})$. □

In the following lemma we prove that if the encoding of the concrete system $M = trans(\mathcal{T}_{SAS})$ satisfies the property $f = trans(\hat{\varphi})$ the concrete system $\mathcal{T}_{SAS}$ itself satisfies the concretization of the abstract property $\mathcal{C}(\hat{\varphi})$.

Lemma 3. *Let* $M = trans(\mathcal{T}_{SAS})$ *and* $f = trans(\hat{\varphi})$. *If* $M \models f$ *then* $\mathcal{T}_{SAS} \models \mathcal{C}(\hat{\varphi})$

Proof. The proof proceeds by induction on the structure of the formula f and follows almost immediately by construction of f and M. For an atomic proposition of M, we know that it is a pair $(a, \mathcal{C}(a))$. If a state s of M satisfies $(a, \mathcal{C}(a))$, $(a, \mathcal{C}(a)) \in L(s)$. Then by construction of M, $s \models_{\times} (a, \mathcal{C}(a))$ in $\mathcal{T}_{SAS}$. By definition of $\models_{\times}$, $(\mathcal{T}_{SAS}, s) \models \mathcal{C}(a)$. The induction step follows from the fact the the structure of the formula is not changed by the applied transformations. □

In order to apply the property preservation results from [7], it now remains to show that the transformed Kripke structures M and M' simulate each other. While for simulation between Kripke structures it is required that similar states have coinciding labels simulation between SAS models does not consider properties in the first place. Therefore, we introduce the notion of consistent simulation. Two SAS models consistently simulate each other, if the simulation relation preserves the atomic propositions. As we consider an abstract and a concrete system, we refine the notion such that if an abstract state satisfies an abstract atomic proposition a similar concrete state must satisfy its concretization.

Definition 19 (Consistent Simulation). *Let $\mathcal{T}_{SAS}$ and $\mathcal{T}_{\widehat{SAS}}$ be two SAS such that $\mathcal{T}_{SAS} \preceq \mathcal{T}_{\widehat{SAS}}$ and $\mathcal{C} : \mathsf{AL}_{SAS}[\mathcal{T}_{\widehat{SAS}}] \to \mathsf{AL}_{SAS}[\mathcal{T}_{SAS}]$ a concretization function. We say that $\mathcal{T}_{\widehat{SAS}}$ consistently simulates $\mathcal{T}_{SAS}$ with respect to a set of atomic propositions AP, denoted as $\mathcal{T}_{SAS} \preceq_{[AP]} \mathcal{T}_{\widehat{SAS}}$, if for all $\sigma \in \Sigma$ and $\hat{\sigma} \in \widehat{\Sigma}$ with $R(\sigma, \hat{\sigma})$ and for all $a \in AP$ it holds that if $(\mathcal{T}_{\widehat{SAS}}, \hat{\sigma}) \models a$ then also $(\mathcal{T}_{SAS}, \sigma) \models \mathcal{C}(a)$.*

The following lemma states that if $\mathcal{T}_{\widehat{SAS}}$ consistently simulates $\mathcal{T}_{SAS}$ then M' simulates M in terms of labeled Kripke structures.

Lemma 4 (Simulation Lemma). *Let $\mathcal{T}_{SAS}$ and $\mathcal{T}_{\widehat{SAS}}$ be two SAS and $M = trans(\mathcal{T}_{SAS})$ and $M' = trans(\mathcal{T}_{\widehat{SAS}})$ the Kripke structure encoding with respect to a property $\hat{\varphi}$ over $\mathcal{T}_{\widehat{SAS}}$. Then it holds: If $\mathcal{T}_{SAS} \preceq_{[Atoms(\hat{\varphi})]} \mathcal{T}_{\widehat{SAS}}$ then $M \preceq M'$.*

Proof. By assumption $\mathcal{T}_{SAS} \preceq_{[Atoms(\hat{\varphi})]} \mathcal{T}_{\widehat{SAS}}$. Hence there exists a simulation relation $\mathcal{R}$ between Σ and $\hat{\Sigma}$. As the states of M and M' are by definition equal to Σ and $\hat{\Sigma}$, respectively, it only remains to show that the labeling of a state s in M and a state s' in M' with $\mathcal{R}(s, s')$ satisfies $L(s) = L'(s')$. As $\mathcal{T}_{SAS} \preceq_{[Atoms(\hat{\varphi})]} \mathcal{T}_{\widehat{SAS}}$ is a consistent simulation, we know that for $(s, s') \in \mathcal{R}$ it holds that if $s' \models a$ implies $s \models \mathcal{C}(a)$. By construction of M we have $L(s) = \{p \in AP(\hat{\varphi})\}$ iff $(\mathcal{T}_{SAS}, s) \models_{\times} p$ and by construction of M' we have $L'(s') = \{p \in AP(\hat{\varphi})\}$ iff $(\mathcal{T}_{\widehat{SAS}}, s') \models_{\times} p$. This yields $L(s) = L'(s')$. □

From the previous lemmata, we can now conclude Theorem 1. For a given concrete SAS $\mathcal{T}_{SAS}$ and a given abstract SAS $\mathcal{T}_{\widehat{SAS}}$ and a given property φ over $\mathcal{T}_{SAS}$ and a given abstract property $\hat{\varphi}$ over $\mathcal{T}_{\widehat{SAS}}$ we construct the Kripke structures $M = trans(\mathcal{T}_{SAS})$ and $M' = trans(\mathcal{T}_{\widehat{SAS}})$ and the property $f = trans(\hat{\varphi})$. Assuming that $\mathcal{T}_{SAS}$ and $\mathcal{T}_{\widehat{SAS}}$ consistently simulate each other with respect to the atomic propositions of $\hat{\varphi}$, i.e. $\mathcal{T}_{SAS} \preceq_{[Atoms(\hat{\varphi})]} \mathcal{T}_{\widehat{SAS}}$, we conclude by Lemma 4 that $M \preceq M'$. The first three conditions in Theorem 1 correspond to the definition of consistent simulation with respect to $Atoms(\hat{\varphi})$. By assumption the abstract system satisfies the abstract property $\mathcal{T}_{\widehat{SAS}} \models \hat{\varphi}$. Thus, Lemma 2 gives that $M' \models f$. Now we can apply Theorem 2 from [7] yielding $M \models f$. From Lemma 3, we conclude that $\mathcal{T}_{SAS} \models \mathcal{C}(\hat{\varphi})$. The final thing to show is that the concretized property $\mathcal{C}(\hat{\varphi})$ implies the concrete property φ which is assumed in the fourth condition of Theorem 1 and we are done.