Translation Validation of System Abstractions[⋆]

Jan Olaf Blech, Ina Schaefer, Arnd Poetzsch-Heffter

Software Technology Group
University of Kaiserslautern
Germany

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 adaptive embedded systems.

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 doing the actual proof that the abstraction preserves the considered property. 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 [14]. Thus the correctness of an abstraction is verified for each run

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

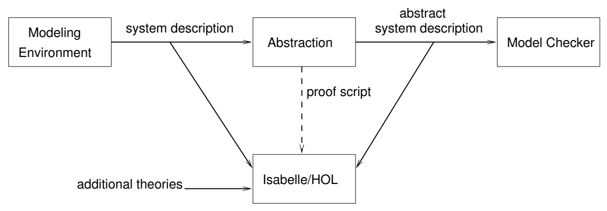

Fig. 1. Our Translation Validation Infrastructure

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 [15] based approach for compilers [12, 4, 16]. 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 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 adaptive embedded systems [1]. 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.

2 Adaptive System Verification

In the EVAS project [1], the application domain is the verification of adaptive embedded systems. The considered adaptive systems consist of a set of synchronously operating modules. Each module is equipped with a set of different predetermined behavioral variants it can adapt to depending on the status of the environment. This enhances system reliability and dependability but also increases design complexity making support for formal verification highly necessary. The systems are developed in a modeling environment also used for

$\mathsf{Module} = (\mathsf{var}, \mathsf{init}, \mathsf{configurations}, \mathsf{adaptation})$ with $\mathsf{var} \subseteq \mathsf{Var}$ and $\mathsf{init} : \mathsf{var} \to \mathsf{Val}$
$\mathsf{configurations} = \{(\mathsf{guard}_j, \mathsf{next\_state}_j, \mathsf{next\_out}_j)\}$ for $\mathsf{i} = 1, \ldots, \mathsf{n}$
guard_j: a Boolean constraint on $\mathsf{adapt\_var}$
$\mathsf{next\_state}_j, \mathsf{next\_out}_j : (\mathsf{var} \to \mathsf{Val}) \to (\mathsf{var} \to \mathsf{Val})$
$\mathsf{adaptation} = (\mathsf{adapt\_var}, \mathsf{adapt\_init}, \mathsf{adapt\_next\_state}, \mathsf{adapt\_next\_out})$
$\mathsf{adapt\_var} \subseteq \mathsf{Var}$ and $\mathsf{adapt\_init} : \mathsf{adapt\_var} \to \mathsf{Val}$
$\mathsf{adapt\_next\_state}, \mathsf{adapt\_next\_out} : (\mathsf{adapt\_var} \to \mathsf{Val}) \to (\mathsf{adapt\_var} \to \mathsf{Val})$

$\mathsf{System} = (\{\mathsf{Module}_1, \ldots, \mathsf{Module}_n\}, \mathsf{var}, \mathsf{adapt\_var}, \mathsf{conn}_a, \mathsf{conn}_d)$

Fig. 2. SAS Module and System Description

other purposes such as code generation. Hence, they contain a level of detail not amenable for automatic verification making system abstractions indispensable.

Figure 2 shows a part of our representation for synchronous adaptive systems (SAS). A full formal account can be found in [1]. A SAS module consists of a set of variables var (divided into state and output variables) together with their initial values, a set of configurations modeling the functional behavior and an adaptation aspect representing the adaptive behavior. Each configuration consists of a guard determining when this configuration is enabled and the attached state transition functions for the state and output variables. The adaptation aspect comprises a distinct set of adaptation specific variables, their initial values and state transition functions for the adaptive state and output variables. This explicit account of adaptive and functional behaviour allows to reason about functional and adaptive aspects in isolation as well as in combination. The semantics of SAS modules is similar to ordinary transition systems with the difference that a transition between two module states evolves in two stages: First, the adaptation aspect computes the new valuation of adaptive state and output variables. Then, the configuration with valid guard is selected and the respective state and output transition functions are executed. A SAS system is composed from a set of modules by connecting their functional and adaptive variables and the system's functional and adaptive variables by functional and adaptive connection functions, conn_a and conn_d resp.

As an example of how abstraction facilitates verification of synchronous adaptive systems, consider a system that consists of one module with two different configurations. Every time 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 every time the input exceeds 50 configuration 1 is used in order to make sure that the appropriate algorithm is employed. This property can be stated in a variant of the temporal logic CTL*[11] as $\varphi \equiv \mathsf{AG}(input \geq 50 \to useconf = 1)$ modeling the used configuration by a variable $useconf$. For φ, the actual functionality of the system is irrelevant.

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

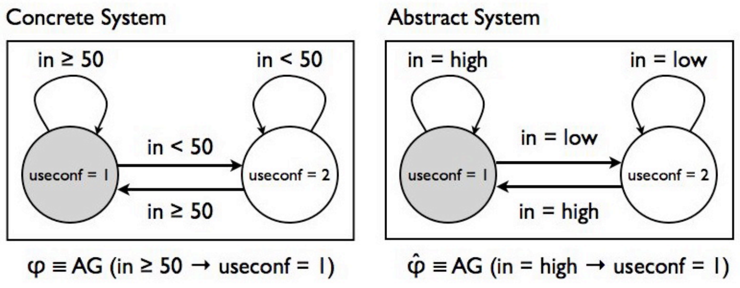

Fig. 3. Illustration of the example system

infinite domain of input values to a finite abstract domain while preserving the property under consideration. 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 3 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.

We apply our approach to adaptive systems in the automotive sector. An adaptive system implementing the ABS (antilock braking system) consists of a large number of different modules and hundreds of different variables ranging over unbounded domains. While in the simple example, the correctness of the abstraction can be easily seen, in real-world examples abstractions become very complex and require support for automatically verifying their correctness.

3 Property Preservation by Simulation

In this section, we present the basis for the correctness criterion used in our translation validation approach. It uses the fact that a property is preserved under abstraction if there is a consistent simulation between abstract and concrete system. In this presentation, we will use general transition systems as SAS semantics are defined in this way. Futhermore, this allows to extend the approach to a broader range of systems expressible as transition systems. For a more detailed formal account and proofs, see the extended version of this paper [5].

Definition 1 (Transition System). *A transition system $\mathcal{T} = (\Sigma, Init, \rightsquigarrow)$ is defined by Σ, the set of states $\sigma : Var \rightarrow Val$ for a set of variables Var and a set of values Val, $Init \subseteq \Sigma$, the set of initial states and $\rightsquigarrow \subseteq \Sigma \times \Sigma$, the transition relation. A path of $\mathcal{T}$ is defined as a sequence of states $\pi = \sigma_0 \sigma_1 \ldots$*

where $\sigma_0 \in Init$ and $\sigma_i \rightsquigarrow \sigma_{i+1}$ for all $i \geq 0$. The set $Paths(\mathcal{T})$ denotes the set of possible paths of $\mathcal{T}$.

We use a variant of the temporal logic CTL*[11] to express properties over computation paths of $\mathcal{T}$. The atomic propositions are constraints on variables, e.g. $x = y$ or $input \leq 50$. Besides Boolean negation, conjunction and disjunction we have temporal operators, e.g. $\mathsf{X}\varphi$ ("next") denoting that φ holds in the next state or $\mathsf{G}\varphi$ ("globally") denoting that φ holds on all states of a path. Additionally, we have path quantifiers $\mathsf{E}\varphi$ and $\mathsf{A}\varphi$. $\mathsf{E}\varphi$ denotes that there exists a computation path on which φ holds. $\mathsf{A}\varphi$ denotes that for all computation paths φ holds. Atomic propositions are interpreted over a state σ by evalutating the variable assignments, e.g. $(\mathcal{T}, \sigma) \models (x = y)$ iff $\sigma(x) = \sigma(y)$. Boolean and CTL* operators are interpreted standardly. $\mathcal{T} \models \varphi$ denotes that φ holds on paths starting in the initial states. $Atoms(\varphi)$ returns the set of atomic propositions used in a CTL* formula φ. ACTL* denotes the fragment of CTL* where only the universal path quantifier A is used.

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

Definition 2 (Simulation of transition systems). *Let $\mathcal{T}$ and $\widehat{\mathcal{T}}$ be two transition systems. We say that $\widehat{\mathcal{T}}$ simulates $\mathcal{T}$, denoted $\mathcal{T} \preceq \widehat{\mathcal{T}}$, iff there exists a simulation relation $\mathcal{R} \subseteq \Sigma \times \widehat{\Sigma}$ such that*

1. *for all $\sigma_0 \in Init$ there exists $\hat{\sigma}_0 \in \widehat{Init}$ such that $\mathcal{R}(\sigma_0, \hat{\sigma}_0)$*
2. *for $0 \leq i$ and $\sigma_i, \sigma_{i+1} \in \Sigma$ and $\hat{\sigma}_i \in \widehat{\Sigma}$ 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 $\hat{\sigma}_i \widehat{\rightsquigarrow} \hat{\sigma}_{i+1}$ and $\mathcal{R}(\sigma_{i+1}, \hat{\sigma}_{i+1})$.*

If a transition system $\mathcal{T}$ is simulated by $\widehat{\mathcal{T}}$ we can show that for each path in $\mathcal{T}$ there is a corresponding path in $\widehat{\mathcal{T}}$. This result is important for the preservation of temporal operators in a CTL* formula. The proof proceeds by induction on the length of a path.

Lemma 1 (Corresponding paths in $\mathcal{T}$ and $\widehat{\mathcal{T}}$). *Let $\mathcal{T}$ and $\widehat{\mathcal{T}}$ be two transition systems such that $\mathcal{T} \preceq \widehat{\mathcal{T}}$ with simulation relation $\mathcal{R}$. Then for every path $\pi = \sigma_0\sigma_1 \ldots \in Paths(\mathcal{T})$ there exists a corresponding path $\hat{\pi} = \hat{\sigma}_0\hat{\sigma}_1 \ldots \in Paths(\widehat{\mathcal{T}})$ such that $\mathcal{R}(\sigma_i, \hat{\sigma}_i)$ for all $i \geq 0$.*

Now we are in the position to justify the criterion that allows to conclude $\mathcal{T} \models \varphi$ from $\widehat{\mathcal{T}} \models \hat{\varphi}$ for φ and $\hat{\varphi}$ in ACTL*. Existential properties are typically lost under abstraction. The 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 an corresponding property φ over the concrete system $\mathcal{T}$. It is defined on atomic propositions and compatibly lifted to ACTL* formulas. This reflects the potentially different interpretations of variables in concrete and abstract system. The concrete choice of simulation relation and concretization mapping depends on the abstraction procedure used.

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

$$\widehat{\mathcal{T}} \models \hat{\varphi} \textit{ implies } \mathcal{T} \models \varphi$$

iff there exists a simulation relation $\mathcal{R} \subseteq \Sigma \times \widehat{\Sigma}$ and a concretization function $\mathcal{C} : \mathsf{A}CTL^*[\widehat{\mathcal{T}}] \to \mathsf{A}CTL^*[\mathcal{T}]$ such that the following conditions hold:

1. *Initial Simulation: for all $\sigma_0 \in \mathit{Init}$ there exists $\hat{\sigma}_0 \in \widehat{\mathit{Init}}$ 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 $\hat{a} \in \mathit{Atoms}(\hat{\varphi})$ if $\mathcal{R}(\sigma, \hat{\sigma})$ and $(\widehat{\mathcal{T}}, \hat{\sigma}) \models \hat{a}$ then $(\mathcal{T}, \sigma) \models \mathcal{C}(\hat{a})$*
4. *Implication: $\mathcal{T} \models \mathcal{C}(\hat{\varphi}) \to \varphi$.*

The proof 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. This theorem 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 eases to work with systems where states are described by valuations of variables such as in SAS.

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 mapped 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 $x \in \mathit{Var}$. 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 compatibly lifted to ACTL* formulas. This form of abstraction is also applied in the example of Section 2.
Another abstraction procedure that can be mapped to this theorem is omitting variables that are irrelevant for the considered property, similar to dead code elimination in compiler optimization. Here, the abstract system $\widehat{\mathcal{T}}$ only contains a subset of the variables of $\mathcal{T}$, i.e. $\widehat{Var} \subseteq Var$ 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 $x \in \widehat{Var}$. 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. Besides these two abstraction procedures we aim at extending our work to more complicated and powerful abstractions (see Future Work in Section 7).

4 The Translation Validation Infrastructure

In this section, we describe the different steps for verifying a system abstraction correct in Isabelle/HOL[14]. Firstly, we have to generate an Isabelle/HOL description of both the concrete and the abstract system. Secondly, 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 (cf. Figure 2). We chose the higher order theorem prover Isabelle/HOL for its greater degrees of freedom in specification allowing shorter and more elegant formalizations.

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 to a shallow embedding, a deep embedding would require to formalize the syntax of the system description language in Isabelle[1] and define a semantics on top of the syntactical elements. Some of the SAS specifications are not entirely formulated as executable programs. Instead they are only 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.

[1] see e.g. [18] for a comparison between deep and shallow embedding in an Isabelle/HOL environment

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 system 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 (`adapt_var, var` cp. Fig 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 (adapt_next_state and adapt_next_out) is evaluated. 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 adapt_next_state and 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. 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 single state transition function representing a system's semantics. This technique works for concrete and abstract systems equally well.

4.2 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 4) 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, 12]).

```
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. 4. 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.

Figure 5 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 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 designed 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. 5. 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.

4.3 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

```
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. 6. Proof Script

criterion. The `simu_step_helper` lemma is a generic part for proving abstraction of variable domains and omission of variables correct. The lemma as well as its proof is depicted in Figure 6. 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 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. For the case studies examined so far, we have developed a single highly generic proof script (which the lemma `simu_step_helper` is a part of) that proves the correctness in all scenarios containing domain abstractions and omission of variables. 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. Examined system representations had 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 modeling environment (used e.g. for code generation) and an input representation for verification tools. The time to conduct the proofs did not turn out to be a problem contrary to our translation validation work on compilers [4].

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 [15, 19] 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[14] serves as checker in our case. In the original translation validation approach[15] 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 [17] or certifying compilation [12]. 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 [13]. Since, property preservation by simulation has been studied 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 labeled with atomic propositions or atomic propositions are labeled with states. This reduces the consistency condition to checking that the labeling 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.

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/HOL. 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 modularization 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 *Proc. of ICFEM 2007*, 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. On Translation Validation for System Abstractions. Technical Report 361-07, TU Kaiserslautern, July 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. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam, 1990.
12. 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.
13. R. Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI*, pages 481–489, 1971.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of TACAS*, volume 1384 of *LNCS*. Springer, 1998.
16. A. Poetzsch-Heffter and M. J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.
17. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proc. of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
18. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Theorem Proving in Higher Order Logics*, LNCS. Springer, 2004.
19. 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.