Verification of Statecharts Using Data Abstraction

Steffen Helke
Brandenburg University of Technology
Cottbus-Senftenberg (BTU)
Cottbus, Germany

Florian Kammüller
Middlesex University London
London, UK

***Abstract*—We present an approach for verifying Statecharts including infinite data spaces. We devise a technique for checking that a formula of the universal fragment of CTL is satisfied by a specification written as a Statechart. The approach is based on a property-preserving abstraction technique that additionally preserves structure. It is prototypically implemented in a logic-based framework using a theorem prover and a model checker. This paper reports on the following results. (1) We present a proof infra-structure for Statecharts in the theorem prover Isabelle/HOL, which constitutes a basis for defining a mechanised data abstraction process. The formalisation is based on Hierarchical Automata (HA) which allow a structural decomposition of Statecharts into Sequential Automata. (2) Based on this theory we introduce a data abstraction technique, which can be used to abstract the data space of a HA for a given abstraction function. The technique is based on constructing over-approximations. It is structure-preserving and is designed in a compositional way. (3) For reasons of practicability, we finally present two tactics supporting the abstraction that we have implemented in Isabelle/HOL. To make proofs more efficient, these tactics use the model checker SMV checking abstract models automatically.**

***Keywords*—Statecharts; CTL; Data Abstraction; Model Checking; Theorem Proving**

I. Introduction

The Statecharts formalism [1] combines a state based automata formalism with intuitive behaviour and hierarchical and parallel state composition. In addition, each state of a state chart contains data for the modeling of real world systems. This data can be naturally changed in transitions. Thus, the Statecharts formalism supports a concise and natural presentation of large models of reactive and embedded systems. It meets high acceptance in industry in particular compared to other formal methods like Z [2] or CSP [3]. The benefits of structured state and data contained in a Statechart come at a price. The validation of properties of Statecharts is a complex endeavor. The hierarchical state structure produces intricate semantic decision problems if several state transitions are enabled simultaneously at different hierarchical levels and in several parallel states. Moreover, the presence of data in states inherits a classical problem from concurrent programming: data races, i.e. successor states are ambiguous if parallel writes happen on one state variable since it is unclear which write occurs before the other.

These issues raised by the complexity of the formalism make a mechanical support for the development of Statecharts models imperative. A natural choice for verification tools for Statecharts are model checker [4] since they implicitly address state based models of transition systems. Frameworks for model checking Statecharts exist, e.g. [5], but from our point of view they do not support the modeling of Statecharts well enough. One apparent reason is that many of these approaches omit the data contained in Statecharts to avoid the well-known state-explosion problem: since data domains may be infinite, e.g. integers, the state graph becomes infinite and model checking fails because it attempts a complete traversal. We propose a framework addressing this issue in our work that uses abstraction techniques making model checking applicable.

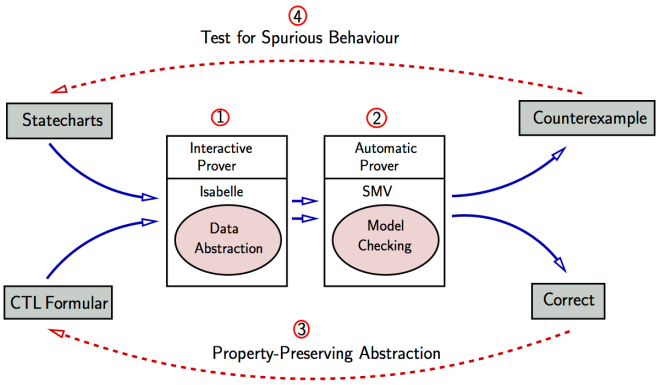

Fig. 1: Verification of Statecharts including infinite data spaces

However from the point of view of a modeler, the use of Statecharts poses more pressing practical problems. Abstracting data from a Statechart adds behaviour since it omits detail about concrete decisions between transitions. This behaviour created in such an overapproximation is called spurious behaviour. It is necessary to include in order to produce faithful abstractions that allow checking the abstraction reliably while producing check results that are valid for the concrete model. The concrete models are flattened in this abstraction process and become unreadable for the model designer who does not recognise the original data conditions. For example, if a counterexample is found that is based on spurious behaviour, this is hard to understand for the user since it is behaviour that has not been there in the concrete model.

We counter this problem by proposing abstractions that are again Statecharts with readable data conditions (for example, as conditions on transitions). To retain the convenience of model checking while still supporting a realistic Statecharts formalism that contains real data, we employ Higher Order Logic theorem proving with Isabelle/HOL [6] in combination with model checking. As illustrated in Fig. 1, we input a full data containing Statechart and a property to verify on this chart

into Isabelle (1). We then apply data abstraction to enable a representation in the model checker SMV (2). Applying SMV we either find a counterexample to the property to be verified – in which case we test for spurious behaviour (4) – or SMV accepts the model and property – which means the original Statechart has the required property since we use a property preserving abstraction (3).

The additional use of a Statecharts formalisation in Isabelle/HOL provides a rigorous quality assurance: no hand-made abstraction – commonly used when model checking Statecharts – can endanger soundness of verification results. Moreover, we conservatively embed the theory of abstraction itself in Isabelle/HOL whereby the actual abstraction process guarantees property preservation. It is not practical (though perfectly possible) to abstract a data containing Statechart in Isabelle/HOL using the logical theory. Therefore, we offer additional tactic support for abstraction that automatically produces – given suitable abstraction predicates – the abstract Statechart's representation plus the proof obligations that must be provided by the user to guarantee wellformedness (naturally being assisted by Isabelle's powerful automated proof procedures).

We use a description of Statecharts by Hierarchical Automata (HA) an established representation of Statecharts [7], [5]. HA enable the structural decomposition of Statecharts into so-called Sequential Automata (SA). This structural decomposition makes efficient and concise proofs possible but also supports data modelling. The data modelling is the central aspect of our proposed framework. The accompanying abstraction needs to be *structure-preserving* – i.e. it must preserve the Statechart's structure. This implies that the *data* abstraction must be respected as well by the abstraction of the Statechart. Abstraction has thus a compositional aspect: independent abstraction of the single SAs must be transferable to the abstraction of the HA containing these SAs. For example, the correctness condition `AbsCorrect` (see Definition 3.1 in Sec. III-D2) is a prerequisite for application of the model checking abstraction.

In this paper, after a detailed description of related work, we first present a motivating example of a safety injection system for nuclear power plants. We then give an overview of the formalisation of the abstract syntax and semantics of Statecharts in Isabelle/HOL (Sec. II). We next introduce in detail the formalisation of the abstraction theory in Isabelle/HOL (Sec. III). The abstraction theory is naturally divided in two parts: the abstraction process for a single SA (Sec. III-B) and how it is composed into an abstraction for the composite HA (Sec. III-D). Finally, we present the practical working with the framework for model checking Statecharts (Sec. IV). We introduce the tactic support and illustrate it on the running example. The Isabelle/HOL formalisation is part of the Archive of Formal Proof [8] and can be downloaded from there. The paper closes with conclusions and ideas for future work in Sec. V.

A. Related Work

The first formalisation of Statecharts in a theorem prover has been provided, to our knowledge, by Nancy Day in 1993 [9]. In this Master's project, she translated Statemate Statecharts into the input language of the HOL-Voss tool [10]. The HOL-Voss system is an integrated tool consisting of the theorem prover HOL [11] and a symbolic model checker. The focus of Day's work is not so much on the efficient formalisation within the HOL-system; rather, she aims at providing a front end for the verification of CTL formulas using the model checker that has been integrated into HOL-Voss. Although this formalisation of Statecharts does not use HA, there is a strong connection to our work. In particular, Day's approach supports the handling of data variables. The resulting effects – for example racing – are also touched on in the formalisation. However, the formalisation of data spaces has been addressed much more thoroughly in our work. By contrast, the work of Nancy Day contains no description of how temporal formula can be interpreted semantically with respect to a Statecharts specification. Day also fails to address data-abstraction concepts for Statecharts, for which our formalisation paves the way.

We are also familiar with a work on the formalisation of Statemate Statecharts in the proof assistant KIV [12]. KIV (*Karlsruhe Interactive Verifier*) [13] is an interactive theorem prover based on the Logic of Computable Functions (LCF) [14]. The authors define a sequent calculus for the verification of Statecharts specifications with infinite data spaces. However, unlike our work, the data spaces are described by a separate algebraic specification. Moreover, the approach differs in that an asynchronous macro-step semantics has been formalised for Statecharts, which differs from the synchronous step semantics used in our work.

Formalisations for UML Statecharts also exist for the theorem prover PVS [15]. In 2000, for example, Issa Traore presented a formalisation for UML Statecharts that was expressed in the input language of PVS [16]. In particular, he documents in his work how UML Statecharts can be verified using a model checker available in PVS. Based on this work, Demissie B. Aredo developed a similar – but more elaborated – formalisation [17]. In contrast to our work, this is a formalisation of UML Statecharts, which is not based on HA.

Concerning the model checking of Statecharts, we have to distinguish existing approaches according to which Statecharts semantics they use. We focus here on related work mostly with respect to the Statemate-Statecharts – for detailed information on how it relates to verification of UML state machines see [18]. However we discuss one recent work that addresses model checking of UML models, since it is fairly close to our work [19]. The authors present a CEGAR-like method for abstraction and refinement of behavioural UML systems. The behaviour of a system is there represented by the communication of a finite number of UML state machines. To verify properties effectively and automatically they use a so called model-to-model transformation, which means that - simliar to our approach - the result of abstractions and refinements can be again an UML model. However the main difference to our work is that they abstract and refine the interfaces of the state machines only, which does not work for Statemate-Statecharts. Further they use a "don't-know" value for data variables, which changes the semantics of UML state machines slightly.

Our approach to the model checking of Statecharts is much in the tradition of the work of Erich Mikk. He proposes in his work a semantics for Statecharts based on Extended HA [5] and uses this as a basis for model checking. He defines and implements one translation to the input language of the

CTL model checker SMV and a second one to the input language Promela [20] of the LTL model checker SPIN [21]. In contrast to our work, Mikk's work does not address Statecharts containing data.

Jan-Juan Hiemer suggests in his dissertation [22] to verify Statemate-Statecharts with the CSP model checker FDR [3]. In a work by Bill Roscoe and Zhenzhong Wu, the translation of Statemate-Statecharts to CSP is optimised – amongst other things – by a simple treatment of data [23]. However, this approach is restricted to finite data domains. Conflicts caused by concurrent write to the data space is represented by a special error-event unlike our solution with an interleaving semantics.

Udo Brockmeyer and Gunnar Wittich use a symbolic model checker produced by Siemens for the verification of Statemate models [24], [25]. They emphasise in their work the treatment of time-constraints specified in Timed CTL – an extension of CTL by discrete time. They do not support data spaces.

B. Relation to our Previous Work

Some parts of this journal paper have been published at two conferences [26], [27] and a workshop [28]. However, apart from the doctoral dissertation [18], which is written in German, there is no publication that presents these parts in an overall context. Furthermore, in this paper we give not only an idea on how single items can be related, we revisit our previous work and present our improved concepts based on a reworked formalisation [8].

A first formalisation of Statecharts by HA was presented in [26] and relates to Sec. II. Note, that this formalisation includes no data spaces. Further, this conference paper is focused on the comparison between the traditional set-based encoding of HA and an alternative optimised variant which exploits the tree-like structure of HA using Isabelle's datatypes and primitive recursive functions. This optimised version should only be used if the main focus is on proving Statecharts properties completely within Isabelle (and not outsourcing them to other automated tools). For practical applications of Statecharts we do not recommend this.

A first idea how to formalise data spaces of HAs was given in [28]. This workshop paper deals with the challenge how to formalise a partial update on a single data partition without influencing the rest of the data space. This is realised by so called generic update function that abstract over other partitions using a lambda abstraction. However from today's perspective we believe that this encoding is too technical and is solved more elegantly by a special type for partial update-functions [8]. Note, that only this new formalisation allows us to define the properties of an abstraction function in precise manner (cf. Definition 3.1), which we did not achieve before.

Finally a first version of our abstraction theory on HA is sketched in [27]. This paper introduces the so called abstraction operators to build a structure- and property-preserving abstraction of an HA inside of Isabelle. In this journal paper we omit the detailed description of the operators. Instead, for practical reasons we focus on constructing the abstraction outside of Isabelle using more efficient algorithms, which is only very roughly described in [27].

C. Example Specification: Safety Injection System

In this section we introduce a short example specification to model the behaviour of a safety-critical system. We consider a reactive cooling system, which can be installed in a nuclear power plant. In the following sections we reuse this example to illustrate the verification technique.

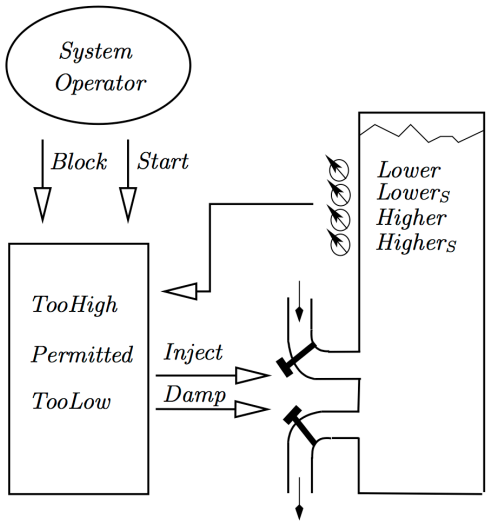

Fig. 2: Construction of a cooling system in a nuclear power plant

Fig. 2 shows the construction of the system: on the upper left is the global system operator capable of starting or blocking the system. The control unit, depicted below, once started, observes the actual cooling loop on the right via four sensors and can manipulate it via two valves. Depending on the pressure in the loop of the cooling system the valves have to be opened by the control software. The sensors *Lower* and *Higher* detect big changes in pressure whereas $Lower_S$ and $Higher_S$ detect small changes. For example, in case that the water pressure is falling under a limit (*TooLow*), the control software must generate an event (*Inject*) that causes the upper valve of Fig. 2 to open. Thereby, new cooling water comes into the loop. Similarly, the lower valve of Fig. 2 must be activated by an event (*Damp*) if overpressure occurs.

A first description of this application was published by David L. Parnas [29]. The functionality there is restricted to injecting cooling fluid into the loop. After some years, this description was rendered more precise in an SCR requirements specification [30]. Furthermore, Bultan et al. have extended the example by reducing steam in the case of overpressure [31]. Based on this work, we have developed a Statecharts specification as depicted in Fig. 3. On the top level, the specification consists of three states, which are composed in parallel. In the following, we describe the behaviour for each of these parallel states. The specification contains a data space, which consists of the single integer variable *pressure*.

The controller for both valves is represented by State *DeviceCTRL*. Initially, the system is blocked. Event *Start* indicates that the controller has to be activated. At any time, this activation can be canceled by Event *Block*. Initially, the system is situated in Substate *Idle*. Depending on the activated substate in State *PressureMode*, the contoller changes into Substate *Damp* or Substate *Safety-Injection*. If the water pressure is in a regular segment, we still are in State *Idle*. If the event

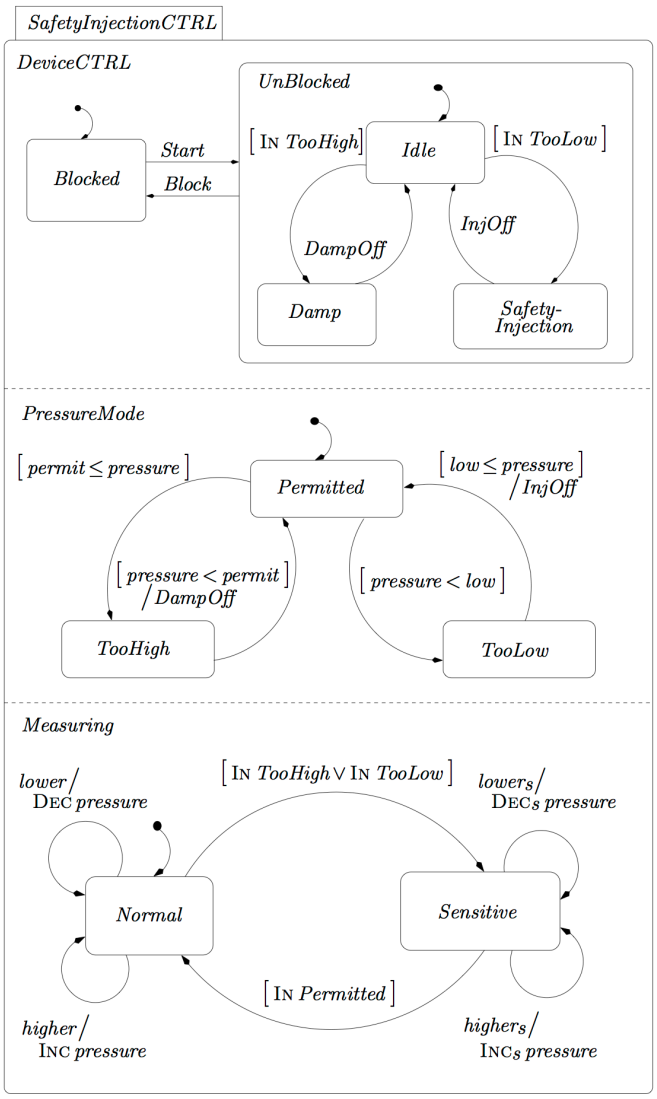

Fig. 3: Statecharts Specification of a *Safety Injection System*

DampOff occurs, State *Damp* must be left. Similarly, State *Safety-Injection* must be left, if Event *InjOff* occurs.

In the middle of Fig. 3, we model State *PressureMode*. Depending on the constants *permit* and *low*, we determine whether the value of the data variable *pressure* is in the regular segment (*Permitted*) or in a critical segment. Critical segments can be an overpressure (*TooHigh*) or a too low pressure (*TooLow*). Whenever a critical situation can be averted, the value of *pressure* returns to the regular segment. This effect comes along with generating Event *DampOff* or Event *InjOff* in order to re-close the appropriate valves.

Finally we describe State *Measuring*, which is depicted at the bottom of Fig. 3. In this state we determine the value of the data variable *pressure*. First, we consider the measuring procedure for the regular mode. This means that the data variable is in the regular segment and State *Normal* is activated. The pressure change is discovered by sensors of the environment. If the pressure has increased, Event *higher* is generated by an appropriate sensor. Consequently, the

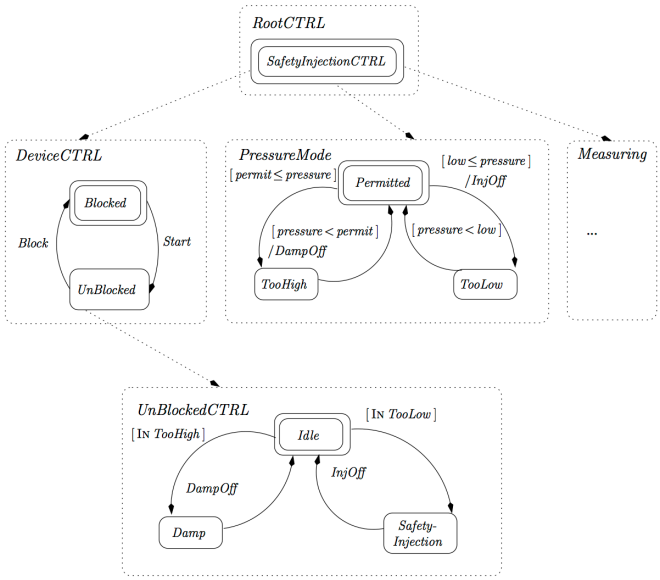

Fig. 4: Hierarchical Automaton of the *Safety Injection System*

data variable must be changed by the operator INC. Similar Event *lower* is generated for a decreased pressure and the variable must be changed by the operator DEC. Whenever *pressure* values outside of the regular segment are reached, the measuring becomes more sensitive. This is modeled by the state transition between State *Normal* and State *Sensitive*. In contrast to the measuring procedure of the regular mode, the pressure change is discovered by more sensitive sensors. In the model, this is reflected by the Event $higher_s$ and the Event $lower_s$. Consequently the data variable must be changed by the operators INC_s and DEC_s.

II. FORMALISING STATECHARTS IN ISABELLE/HOL

A Statecharts specification can be adequately represented by a Hierarchical Automaton[1] which consists of a finite set of Sequential Automata. The Hierarchical Automaton of the safety injection system in Fig. 4 gives a basic impression of this representation. We have thus decomposed the Statecharts into five Sequential Automata, which are connected by arrows in order to represent the hierarchy of the specification. In this section, we describe a formalisation of Statecharts using Hierarchical Automata in Isabelle/HOL.

We begin by introducing the definition of Hierarchical Automata. To this end, we provide a *type* for the description of Hierarchical Automata by building the type abstraction over a set. In Definition 2.1, we represent the composition of Sequential Automata as a partial function by the type constructor $\rightarrow$. Sequential Automata are basically labelled transition systems; to keep the exposition concise we omit its definition here; for the full definitions see [8].

[1]Note, we support a subset of Statechart's syntax only, e.g. an inter-level transition cannot be described by an ordinary HA and would need the expressive power of EHA (*Extended Hierarchical Automata*). However, Mikk et. al have shown[7], [5] that such an extension is straightforward and does not imply any fundamental restrictions, because this coding is structure-preserving as well.

Definition 2.1 (Hierarchical Automata (HA)): Let σ be a type of state identifiers, ϵ a type of event identifiers, and δ a type for the data space. Then a Hierarchical Automaton HA over $(\sigma, \epsilon, \delta)$ is represented by a quadruple $(\mathrm{S_A}, \mathrm{E}, \mathrm{F_{Comp}}, \mathrm{D})$, where

- $\mathrm{S_A}$ is the set of sequential automata
- E is the set of events
- $\mathrm{F_{Comp}}$ is the composition function, and
- D is the initial assignment of the data space

of the Hierarchical Automaton HA. These components must fulfil the internal consistency condition HierAutoCorrect on Hierarchical Automata. The type $(\sigma, \epsilon, \delta)$ hierauto consists of all Hierarchical Automata over $(\sigma, \epsilon, \delta)$.

$$
\begin{aligned}
(\sigma,\epsilon,\delta)\,\mathtt{hierauto} \quad \equiv_T \quad & \{\,(\mathrm{S_A}, \mathrm{E}, \mathrm{F_{Comp}}, \mathrm{D}) \mid \\
& (\mathrm{S_A} :: ((\sigma,\epsilon,\delta)\,\mathtt{seqauto\,set})) \\
& (\mathrm{E} :: \epsilon\,\mathtt{set}) \\
& (\mathrm{F_{Comp}} :: (\sigma \nrightarrow ((\sigma,\epsilon,\delta)\,\mathtt{seqauto\,set}))) \\
& (\mathrm{D} :: \delta\,\mathtt{data}). \\
& \mathtt{HierAutoCorrect}\ \mathrm{S_A}\ \mathrm{E}\ \mathrm{F_{Comp}}\ \mathrm{D}\,\} \\
& \textit{justified by}\ \mathsf{HierAutoNonEmpty}
\end{aligned}
$$

The above definition is an example of an Isabelle/HOL type definition. Type definitions are the basic building block for so-called *conservative extensions* of HOL. A new type is defined by a defining predicate (in the above case *HierAutoCorrect)* over an existing type. The new type is then given as a disjoint copy of all elements of the old type that fulfil the defining predicate. This predicate becomes a *well-formedness* condition for all elements of the new type. Two internally created bijections between the new type and the elements of the old type identified by the predicate – in the example *Abs_hierauto* and *Rep_hierauto* – enable a translation between old and new type. A type extension is conservative because new predicates and operators over the new type may only be defined based on the operators of the base type implicitly assuming the well-formedness predicate for elements of the domain of new functions.

For the above example, we omit the actual definition of *HierAutoCorrect* (see [8] for a complete presentation of all definitions). It essentially encodes that F_{Comp} is a well-formed composition with respect to S_{A} as defined in the following Definition 2.2.

Definition 2.2 (Well-Formed Composition): Let $\mathrm{S_A}$ be a set of Sequential Automata and $\mathrm{F_{Comp}}$ a composition function. The predicate IsCompFun describes the internal consistency for these components.

$$
\begin{aligned}
\mathtt{IsCompFun} \quad ::_C \quad & [(\sigma,\epsilon,\delta)\,\mathtt{seqauto\,set}, \\
& \sigma \nrightarrow ((\sigma,\epsilon,\delta)\,\mathtt{seqauto\,set})\,] \to \mathtt{bool} \\
\mathtt{IsCompFun}\ \mathrm{S_A}\ \mathrm{F_{Comp}} \quad \equiv_C \quad & \\
& \mathtt{dom}\ \mathrm{F_{Comp}} = \Big(\bigcup \mathrm{A} \in \mathrm{S_A}.\ \mathtt{States\ A}\Big) \ \wedge \\
& \Big(\bigcup \mathtt{ran}\ \mathrm{F_{Comp}}\Big) = (F - \{\mathtt{Root}\ \mathrm{S_A}\ \mathrm{F_{Comp}}\}) \ \wedge \\
& \mathtt{RootExists}\ \mathrm{S_A}\ \mathrm{F_{Comp}} \ \wedge \\
& \mathtt{OneAncestor}\ \mathrm{S_A}\ \mathrm{F_{Comp}} \ \wedge \\
& \mathtt{NoCycles}\ \mathrm{S_A}\ \mathrm{F_{Comp}}
\end{aligned}
$$

The predicate *RootExists* S_{A} F_{Comp} guarantees that there exists an unique root automaton called *Root* S_{A} F_{Comp}. The predicate *OneAncestor* S_{A} F_{Comp} reflects that each Sequential Automaton of S_{A} except the root automaton has exactly one ancestor state. The last predicate *NoCycles* ensures that the composition function does not contain any cycles.

$$
\begin{aligned}
& \mathit{NoCycles}\ S_{\mathrm{A}}\ F_{\mathrm{Comp}} \quad \equiv_C \\
& \quad \forall S : \mathbb{P}\ \Big(\bigcup \mathrm{A} : S_{\mathrm{A}}.\ \mathit{States}\ \mathrm{A}\Big)\ . \\
& \quad S \neq \varnothing \implies \\
& \quad \exists s : S.\ S \cap \Big(\bigcup A : \mathit{the}(F_{\mathrm{Comp}}\ s).\ \mathit{States}\ A\Big) = \varnothing
\end{aligned}
$$

A. Optimisation

When applying the formalisation introduced in the previous section, we frequently need selection theorems for a Hierarchical Automaton *HA* to obtain its constituents, e.g. all its defining Sequential Automata. We must provide such selection theorems in Isabelle/HOL by proving them. For these proofs, we need a special theorem, that reflects the well-formedness property of Hierarchical Automata (essentially Definition 2.2). Deriving a well-formedness property is expensive. For example, the check that there are no cycles in the composition function involves checking a predicate for each non-empty subset of the state set of *HA*. We thus obtain $2^{\#(\mathit{States\ HA})}$ different proof obligations. Clearly, this proof is inefficient even for small Hierarchical Automata.

To solve this problem, we propose a stepwise procedure to construct Hierarchical Automata from their defining Sequential Automata. Here, we need to exploit the tree-like structure of Hierarchical Automata. We define two constructors: one to build a pseudo Hierarchical Automaton from a given Sequential Automaton, and another to add a given Sequential Automaton at a specified state position to a Hierarchical Automaton extending the constituents of the Hierarchical Automaton. The idea is that this construction process guarantees wellformedness thus reducing the cost of proof. We start with the definition of a pseudo HA. The result of this construction does not contain a hierarchy and represents in fact a Sequential Automaton, which is wrapped up as a Hierarchical Automaton.

$$
\begin{aligned}
\mathit{PseudoHA\ SA\ D} \quad \equiv_C \quad & \mathit{Abs\_hierauto}(\{\mathit{SA}\}, \mathit{Events\ SA}, \\
& \mathit{EmptyMap}\ (\mathit{States\ SA}), D)
\end{aligned}
$$

The above used operator *EmptyMap* defines a composition function, in which each state in the set of states of an SA is mapped onto the empty set. The operator ensures that a pseudo HA contains only simple states.

Using the following construction operator $\boxplus$, we extend a Hierarchical Automaton by a Sequential Automaton at a specified state position.

Definition 2.3 (Extension of a HA by a SA): Let HA be a Hierarchical Automaton, S a state and SA a Sequential Automaton. Then the extension of HA by SA in the state S is defined as follows.

$$
\begin{aligned}
\_ \boxplus \_ \quad ::_C \quad & [(\sigma,\epsilon,\delta)\,\mathtt{hierauto}, \sigma * (\sigma,\epsilon,\delta)\,\mathtt{seqauto}] \\
& \to (\sigma,\epsilon,\delta)\,\mathtt{hierauto} \\
\_ \boxplus \_ \quad \equiv_C \quad & (\lambda\,\mathtt{HA}\ (\mathrm{S}, \mathrm{SA})\ . \\
& \quad \mathsf{let}\ S'_{\mathrm{A}} = \{\,\mathrm{SA}\,\} \cup (\mathrm{SAs\ HA}); \\
& \quad\quad \mathrm{E}' = \mathtt{Events\ HA} \cup \mathtt{Events\ SA}; \\
& \quad\quad \mathrm{F}'_{\mathrm{Comp}} = \mathtt{CompFun\ HA} \oplus (\mathrm{S}, \mathrm{SA}) \\
& \quad\quad \mathrm{D}' = \mathtt{InitData\ HA} \\
& \quad \mathsf{in}\ \mathtt{Abs\_hierauto}\,(S'_{\mathrm{A}}, \mathrm{E}', \mathrm{F}'_{\mathrm{Comp}}, \mathrm{D}'))
\end{aligned}
$$

In Definition 2.3, we use the operator $\oplus$ to extend the composition function F_{Comp} by a Sequential Automaton *SA* on

```
SafetyInjectionSystem ::_C (string, string, ds) hierauto
SafetyInjectionSystem ≡_C
        (PseudoHA RootCTRL (LiftInitData [V_0 15]))
          ⊞ ("SafetyInjectionCTRL", DeviceCTRL)
          ⊞ ("SafetyInjectionCTRL", PressureMode)
          ⊞ ("SafetyInjectionCTRL", Measuring)
          ⊞ ("UnBlocked", UnBlockedCTRL)
```

Fig. 5: Stepwise construction for the model of the *Safety Injection System*

a state S. The extension is well-formed if and only if two conditions hold. First, S must be contained in the domain of F_{Comp}, and second, S must not be a state of SA. These conditions ensure that the construction using the operator $\oplus$ yields a cycle-free composition function.

In Fig. 5 we present the definition of the running example of the safety injection system using construction operators. The operator $InitDS$ represents an initial data space assignment. *Pressure* is there assigned to 15.

The benefit of such a construction is that we can derive theorems automatically reflecting selection theorems of constructed Hierarchical Automata. To achieve such automation, we have implemented a tactic in Isabelle/HOL. The tactic decomposes a constructed Hierarchical Automaton into its components and derives for each component a selection theorem, e.g for the selection of Sequential Automata. These selection theorems are subsequently reused to derive one selection theorem of the whole Hierarchical Automaton. To decompose a constructed Hierarchical Automaton, we use theorems reflecting the well-formedness property of Hierarchical Automata under construction.

A pseudo HA is always well formed.

$$HierAutoCorrect\ \{SA\}\ Events\ SA\ EmptyMap\ (States\ SA)\ D$$

Accordingly, we have derived for the construction operator $\boxplus$ a theorem of well-formedness. However, this property is not always satisfied. We have framed the conditions, which have to be proved.

Theorem 2.4 (Well-Formed $\boxplus$ - Construction):

$$\frac{\boxed{States\ SA \cap States\ HA = \emptyset} \qquad \boxed{S \in States\ HA}}{HierAutoCorrect\ S'_A\ E'\ F'_{Comp}\ D'}$$

$$S'_A = \{SA\} \cup (SAs\ HA) \qquad E' = Events\ HA \cup Events\ SA$$

$$F'_{Comp} = CompFun\ HA \oplus (S, SA) \qquad D' = InitData\ HA$$

The well-formedness theorems help to identify and construct the conditions used in the construction of elements of the type of Hierarchical Automata. To round off the smooth construction with the optimised constructor operators we only need the following set of theorems that helps to lift the selection operators to constructed HAs. To access the SA contained in a pseudo HA we use the following initial theorem.

$$SAs\ (PseudoHA\ SA\ D) = \{SA\}$$

For the general case, the previously identified well-formedness assumptions (see Theorem 2.4) help to select SA in a constructed HA.

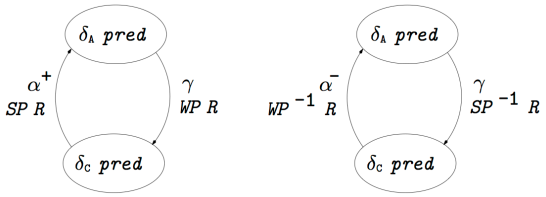

Fig. 6: Galois Connections based on predicate transformers

Theorem 2.5 (Selecting SA of a HA Construction):

$$\frac{States\ SA \cap States\ HA = \emptyset \qquad S \in States\ HA}{SAs\ (HA \boxplus (S, SA)) = \{SA\} \cup SAs\ HA}$$

With the formalisation of these original construction operators we end here the presentation of the basic formalisation of Statecharts in Isabelle/HOL omitting the semantics. It follows – as the basic syntax – quite closely the work by Mikk [5]. Its Isabelle/HOL representation is detailed in [8]. We next consider the abstraction process of Statecharts as a preparation for model checking.

III. Data Abstraction

Our formalisation in Isabelle/HOL [8] contains a theory of the temporal logic CTL and a theory of the specification language Statecharts. Both theories are integrated semantically. In principle, this formalisation enables the derivation of CTL properties on a Statecharts specification. However, the drawback is that the user has to prove these properties in an interactive way. Our approach combines Isabelle/HOL with the model checker SMV to automate the reasoning. The core of this combination is a data abstraction theory for Statecharts, which we introduce in this section.

A. Property Preserving Abstraction

We investigate property preserving data abstraction for Statecharts to preserve CTL formulas. In the literature we find two major approaches for transition systems: over- and underapproximation.

In an overapproximation the abstract model can contain new behaviour, but old behaviour cannot be lost. That is, properties of the universal fragment of CTL ($\forall$CTL), that are valid on all paths of the overapproximated abstract model, must hold on the paths of the concrete model. By contrast in an underapproximation, new behaviour cannot be added, but old behaviour can be lost. Accordingly, properties of the existential fragment of CTL ($\exists$CTL) are preserved by underapproximated abstract models. In the work of Dams [32] two automata representing these different kinds of abstractions are generated. Depending on the property that has to be verified the appropriate model has to be chosen.

The theoretical basis for over- and underapproximations are Galois connections [33]. For complete lattices, C and A, a pair of monotone maps, $\alpha ::_C C \to A$ and $\gamma ::_C A \to C$ define a Galois connection, written $GaloisCorrect\ A\ C\ \alpha\ \gamma$, iff $\alpha \circ \gamma \leq_A id_A$ and $id_C \leq_C \gamma \circ \alpha$. Fig. 6 represents two instantiated Galois connections relating predicates on an abstract (δ_A) and a concrete (δ_C) data space. Since they are predicates, these elements are ordered by implication.

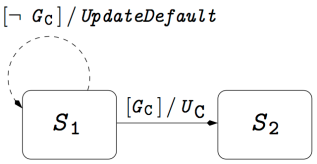

Fig. 7: Implicit behaviour of a SA

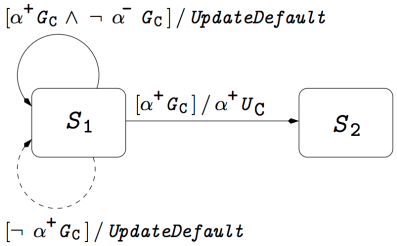

Fig. 8: Overapproximation of a SA

On the left side we define the Galois connection

$$GaloisCorrect\ (\delta_A\ pred)\ (\delta_C\ pred)\ \alpha^+\gamma$$

for overapproximation to weaken a concrete predicate pc by the abstraction α^+as is reflected in the Galois property $pc \implies \gamma(\alpha^+ pc)$. Correspondingly on the right, the Galois connection

$$GaloisCorrect\ (\delta_C\ pred)\ (\delta_A\ pred)\ \gamma\ \alpha^-$$

for underapproximation strengthens concrete predicates. This is reflected in the Galois property $\gamma(\alpha^- pc) \implies pc$.

Given an abstraction function $R ::_C \delta_C \to \delta_A$ mapping elements of the concrete data space into the abstract data space, we can generally define α^+, α^- and γ. The following definitions of strongest post- and weakest precondition form two Galois connections for any such R (cf. Fig. 6).

$$\begin{aligned} SP\ R\ P &\equiv_C \lambda a.\ \exists c.\ (R\,c) = a \wedge P\,c \\ WP\ R\ P &\equiv_C \lambda c.\ \forall a.\ (R\,c) = a \implies P\,a \\ \\ SP^{-1}\ R\ P &\equiv_C \lambda c.\ \exists a.\ (R\,c) = a \wedge P\,a \\ WP^{-1}\ R\ P &\equiv_C \lambda a.\ \forall c.\ (R\,c) = a \implies P\,c \end{aligned}$$

In our theory, the abstraction function is total. Hence, the weakest precondition coincides with the inverse strongest postcondition ($WP\ R = SP^{-1}\ R$). Furthermore, a Galois connection includes that α^+can be expressed by α^- which is reflected by the property $SP\ R\ (\neg\ P) = \neg\ (WP^{-1}\ R\ P)$.

To avoid the detailed definitions of the abstraction functions, we use in the next section an abbreviation. Analogous to the depiction of Fig. 6, we abbreviate $SP\ R$ and $WP^{-1}\ R$ by $\alpha^+$and α^- omiting the parameter R.

B. Overapproximation of SA

Statecharts, HA, and accordingly SA belong to the family of synchronous languages. Synchronous languages build on a synchronous step semantics, which we have formalised for HA in Isabelle/HOL [8]. It includes a formalisation of SA. The semantics of SA is there a special case of the semantics of HA because an SA can be viewed as an HA without hierarchy (cf. the definition of a pseudo HA in Sec. II-A). Based on this decomposition, we present in this section the overapproximation of SA independently from HA. Reusing this theory, we introduce in the next section the overapproximation of HA.

One special property of synchronous languages is that in each semantical status – synchronised by a global clock – the system performs a defined calculating step. Semantical statuses of SA, where no transitions fire, perform a trivial calculating step, in which the data variables are assigned to the previous value. This effect can be interpreted as complementation by implicit transitions. On the left side of Fig. 8, this is depicted by a dashed self-transition, where the guard $\neg\ G_C$ is constructed as the negated guard of the exiting transition. The action-part $UpdateDefault$ represents that the data variables are assigned to the previous value. Note that $UpdateDefault$ will be an unwanted effect in the case of writing data variables by a synchronous executed transition, which must be prioritised. Such synchronous executed transition could arise from an SA which is composed in parallel to the considered SA. A more detailed discussion, how the SAs inside a HA mutually affect each other, can be found in Sec. III-D.

As a rule, the guard of the implicit transition must be constructed as the conjunction of negated guards of all exiting transitions. Overapproximating an SA, we construct an identical structured SA. We adopt the control states and abstract the transitions. Abstracting transitions, we abstract guards and updates separately.

In general, it is impossible to construct an abstracted guard which exactly describes a concrete guard G_C. First, we propose to weaken G_C by α^+ using overapproximation. Building such weaker guards adds new behaviour to the model but deletes some implicit behaviour simultaneously caused by the special semantics of synchronous languages. The reason is that the guard of the implicit transition $\neg\ \alpha^+ G_C$ is automatically stronger. Therefore – secondly – we must add a suitable self transition, to adjust this unwanted effect. The guard of this self transition must be constructed by a conjunction of the overapproximated guard of G_C and the negated underapproximated guards of all exiting transitions. This procedure is illustrated on the right side of Fig. 8.

Abstracting the example, only one exiting transition has to be considered. First, we abstract the guard G_C by $\alpha^+$and introduce a self transition. The guard of this self transition is constructed by a conjunction of the overapproximated guard of the exiting transition $\alpha^+ G_C$ and the negation of the underapproximated guard of the exiting transition $\neg\ \alpha^-\ G_C$. The latter can be expressed by α^+using the theorem of subsection III-A, so that finally we obtain the following guard for the self transition.

$$[\alpha^+ G_C \wedge \alpha^+ \neg\ G_C]$$

In Fig. 8, only one exiting transition exists for State S_1. For *n* transitions, the guard of the self-transition would be constructed in the following way.

$$[\alpha^+ G_C^1 \wedge \neg\ \alpha^-\ G_C^1 \wedge \alpha^+ G_C^2 \wedge \neg\ \alpha^-\ G_C^2 \wedge \ \ldots\ \wedge \alpha^+ G_C^n \wedge \neg\ \alpha^-\ G_C^n]$$

Building traditional overapproximation of updates [34] is compatible with SAs because a weaker update adds new behaviour to the system but old behaviour cannot be lost. Accordingly on the right side of Fig. 8, the update is overapproximated

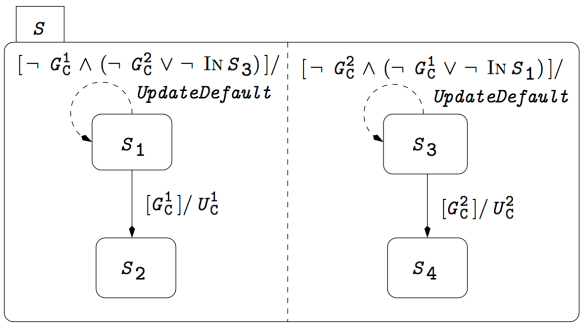

Fig. 9: Implicit behaviour of a HA

by α^+. However, in general, abstracting updates results in non-constructive predicates that violate the action language of Statecharts. More precisely, for each U_C we obtain in general more than one abstract update by α^+.

We must restrict to overapproximation because SAs do not allow a reduction by underapproximation. This is caused by the special semantics of synchronous languages, which can be interpreted as a complementation by implicit transitions. This complementation restricts the possibility for reducing behaviour fundamentally. Consider the example in Fig. 8 on the left side. If we propose to build a stronger guard of G_C by α^-, we obtain a weaker guard for the implicit transition. That is, we add new behaviour to the abstract model, which is unsound for underapproximation. Hence, for the example on the left side of Fig. 8 we cannot build an underapproxmiation, where the abstract model is again a SA. The only way out is to reduce nondeterministic branches to deterministic ones but this is not sufficient for a general procedure. Consequently, the result of an underapproximation cannot usually be expressed by a SA. Hence our abstraction technique is restricted to preserve properties of the universal fragment of CTL. If the overapproximation is too rough, we will obtain undesirable counterexamples, describing so called spurious behaviour. Because of the reason described above, we cannot refine the model directly. Instead in such cases we start from scratch and apply the well-known counterexample guided abstraction and refinement loop (CEGAR) [35] generating a new and more detailed overapproximation.

C. Semantical Characteristics of HAs

We are interested in developing a data abstraction technique which can be applied in a compositional manner. This means that we decompose a given HA into its defining SAs. After that, we abstract each SA independently. Finally, we compose the abstracted SAs to an abstracted HA.

In this section we decribe such compositional abstraction techniques for HAs. Because of the rather complex semantics of HAs this is a challenge: in general, a compositional procedure does not yield a valid overapproximation for a given HA.

1) Implicit behaviour of HAs: Implicit behaviour occurs in synchronous modelling languages whenever a transition cannot fire at the beginning of a clock cycle. In this situation the statechart executes a trivial calculation step that restores the data state. In Sec. III-B the implicit behaviour has been represented in the model by a dashed arrow (cf. Fig. 8). This special transition only fires if no other transition of the model is enabled. As is shown in Fig. 9, we can model the implicit behaviour of a HA explicitly in a similar fashion. Note, however, that with respect to compositional abstraction, the guard of the self-transition depends on context information that lies outside the SA in which the self-loop is defined.

Considering the SA on the left side of Fig. 9, we observe that the guard of the implicit self-transition of the control state S_1 holds, if and only if the guard G_C^1 of the transition exiting S_1 does not hold. In addition a predicate of the parallelly composed SA must hold. Either the control state S_3 is not active or the guard of the transition exiting S_3 is not valid. More generally, in all composed SAs of a HA there must not be any transition that is enabled. The modelling of implicit behaviour of HAs, shown in Fig. 9, becomes more complicated, because usually we have more than one local state in a SA. The concept of partial default update functions – presented in the next subsection – can avoid this effect, because they abstract from the dependencies between parallelly composed SAs (see Fig. 10).

2) Partitions on Data Spaces and Partial Updates: In general, the data space of a HA consists of a finite number of disjoint partitions. Update-functions can be defined in such a way that they do not write on all partitions. The semantics of HAs determines the values of partitions after transition execution also in cases in which a transition does not write on the partition.

On the left side of Fig. 10 we have modelled a data space consisting of two partitions D_C^1 and D_C^2. The update-function of the transition between the states S_1 and S_2 assigns D_C^1 using an auxiliary function U_{PC}^1 and does not write on partition D_C^2. Note, the second partition is assigned to $None$. This indicates in our formalisation that the partition is unwritten. That is, the update-function is partial. In one step of calculation of HA, transitions of several parallel SAs can be executed synchronously. If a transition does not write on a partition (e.g. D_C^2), the first question is whether there is another synchronously executed transition writing on this partition. If this is the case, the value of the synchronously executed transition is selected. In case of a concurring write of several transitions on one partition (so-called racing) the resulting conflict is resolved by introducing a non-determinism (interleaving semantics). In contrast, if there is no transition writing on a partition, the semantics assigns to this partition the value prior to execution of the transition. Note, that we provide a complete formal semantics of HAs including data spaces and partial update-functions in Isabelle/HOL [8].

Summarising the above, it is, in general, not possible to decide locally inside a SA, whether implicit behaviour for writing a partition on the data space will occur. Hence it is challenging to define a compositional abstraction technique, whereby a partial update-function can be abstracted independently from synchronously executed updates. In the next section we propose a procedure abstracting update-functions only based on local informations available inside a SA. Therefore, we demand special properties from the abstraction functions in order to construct overapproximations of HAs in a compositional manner.

D. Overapproximation of HA

We present in this section a data abstraction technique for HA, which is charaterized by three properties. First, it is a structure-preserving abstraction. This means that the result of the abstraction is represented by a HA, whose structure is identical to the input-HA. Second, our abstraction is an over-approximation preserving properties of the universal fragement of CTL. Third, the technique can be applied in a compositional way.

1) Partial Default Update-Functions: First, we adress the modelling of implicit behaviour inside a SA. Consider again the example of Fig. 10, which is depicted on the left side.

Applying the technique of Sec. III-B, we obtain the self-transition on the right side of Fig. 10. Note that in contrast to the dashed self-transition on the left side of Fig. 10, the guard of the abstrated self-transition is constructed only using local informations of the SA. Hence, the guard is weaker than a precise approximation which could be constructed using context informations of SAs, that are composed in parallel. A more precisely approximated guard for our example is the following predicate.

$$[\alpha^+ G_{\mathrm{C}}^1 \wedge \neg\ \alpha^-\ G_{\mathrm{C}}^1 \wedge (\neg\ (\alpha^+ G_{\mathrm{C}}^2 \wedge \neg\ \alpha^-\ G_{\mathrm{C}}^2) \vee \neg\ \mathrm{In}\, S_3)]$$

Consequently, locally constructed self-transitions will be more often executed than self-transitions, which are labeled with precise approximated guards. Nevertheless, this effect is invisible in our semantics because we have used as action a so-called partial default update-function (`PUpdateDefault`). This update-function has a slightly different effect in comparison to `UpdateDefault`. The partial default update-function represents that data partitions are only *potentially* assigned to the previous value. In the case of writing a data partition by synchronously executed transitions, the semantics of `PUpdateDefault` does not have a writing effect on this data partition. Consequently, in this case executing the self-transition is invisible in the semantical states of the abstracted HA.

2) Data Structure Preserving Abstraction Functions: To tackle the problem of Sec. III-C2 we propose to use an abstraction function that preserves the structure of the data space. Therefore, we demand for each concrete data partition a unique counterpart in the abstract data space. Furthermore each concrete partition has to be mapped into the domain of its abstract counterpart using a given abstraction function. Note, that this mapping must be done independently from other partitions of the data space. To ensure this requirement, we force the user of our abstraction technique to prove some special properties of the abstraction function.

Again we consider the example of Fig. 10 on the left side. There we have used a partial update-function writing the first partition of the data space only. To construct a precise overapproximation of this update-function inside a SA, we demand that the binary structure of the data space must be preserved by the abstraction function. This means that the concrete data partitions D_{C}^1 and D_{C}^2 will be mapped into the abstract data partitions D_{A}^1 and D_{A}^2 independently, which is illustrated in the middle of Fig. 10. Assuming that we are interested in using a predicate abstraction, D_{A}^1 and D_{A}^2 would be characterized by boolean variables. In this case, each boolean variable corresponds exclusively to D_{A}^1 or D_{A}^2. Furthermore, each boolean variable reflects the validity of a predicate on the corresponding concrete data partition.

Assuming that both data partitions are declared as integers, we can define an abstraction function for a predicate abstraction in the following way.

$$R\,[D_{\mathrm{C}}^1, D_{\mathrm{C}}^2] \equiv_{df} [D_{\mathrm{C}}^1 \leq D_{\mathrm{C}}^2, D_{\mathrm{C}}^2 \leq 2]$$

In this example, each abstract data partition is represented by a single boolean variable. For example the partition D_{A}^1 is represented by a boolean variable, which is evaluated by the predicate $D_{\mathrm{C}}^1 \leq D_{\mathrm{C}}^2$. Accordingly, D_{A}^2 is represented by a variable, which is evaluated by $D_{\mathrm{C}}^2 \leq 2$. It is obvious that the abstraction function does not map the data partition D_{C}^1 into D_{A}^1 independently from D_{C}^2. Consequently, it is not possible to calculate a precise overapproximation of the update-function U_{PC}^1 locally. The reason is that we calculate the abstract update-function by simulating U_{PC}^1. Therefore, we must map the result of U_{PC}^1 into the abstract data space. However, we have not much information about D_{C}^2 after executing U_{PC}^1 because this partition is locally unwritten. This means that the value of D_{C}^2 can be the previous value of D_{C}^2 or any other value, which is written by a synchronous executed transition. The only way out would be to construct the simulation using type properties of D_{C}^2 resulting in an unprecise approximation. Hence we have defined a special type of wellformed abstraction functions in order to avoid these unprecise approximations.

Definition 3.1 (Abstraction Function): Let δ_{C} be a type of the concrete data space and δ_{A} a type of the abstract data space. Then an abstraction function R over $(\delta_{\mathrm{C}}, \delta_{\mathrm{A}})$ is represented by a triple $(L, D_{\mathrm{C}}, D_{\mathrm{A}})$, where

- *L is the list of functions for the abstraction of data partitions,*
- *D_{C} is the concrete data space, and*
- *D_{A} is the abstract data space*

of the abstraction function R. These components must fulfil the internal consistency condition `AbsCorrect` on abstraction functions. The type $(\delta_{\mathrm{C}}, \delta_{\mathrm{A}})\,abs$ consists of all abstraction functions over $(\delta_{\mathrm{C}}, \delta_{\mathrm{A}})$.

$$\begin{aligned}(\delta_{\mathrm{C}}, \delta_{\mathrm{A}})\,abs \quad \equiv_{\tau} \quad & \{\,(L, D_{\mathrm{C}}, D_{\mathrm{A}}) \mid \\ & (L :: (\delta_{\mathrm{C}} \rightarrow \delta_{\mathrm{A}})\,list) \\ & (D_{\mathrm{C}} :: \delta_{\mathrm{C}}\,dataspace) \\ & (D_{\mathrm{A}} :: \delta_{\mathrm{A}}\,dataspace). \\ & AbsCorrect\ L\ D_{\mathrm{C}}\ D_{\mathrm{A}}\} \\ & \quad justified\ by\ \textit{AbsNonEmpty}\end{aligned}$$

The predicate $AbsCorrect\ L\ D_{\mathrm{C}}\ D_{\mathrm{A}}$ guarantees that the number of concrete data partitions, the number of abstract data partitions, and the number of elements in the function list are identical. Furthermore, the predicate requires for each function of the list that the range of this function is contained in the domain of its corresponding abstract data partition.

$$\begin{aligned}AbsCorrect\ L\ D_{\mathrm{C}}\ D_{\mathrm{A}} \quad \equiv_{c} \\ & \#D_{\mathrm{C}} = \#D_{\mathrm{A}} \wedge \#D_{\mathrm{C}} = \#L \wedge \\ & \forall\ i < \#D_{\mathrm{C}}.\ ran(L!\,i) \subseteq dom\,D_{\mathrm{A}}\,i\end{aligned}$$

Note that the operator ! selects the abstraction function of the i-th partition from the list L. Furthermore, the operator # gives the number of partitions for a data space corresponding to the number of abstraction functions for a list.

Based on this definition of wellformed abstraction functions, we have defined in our Isabelle/HOL-formalisation an

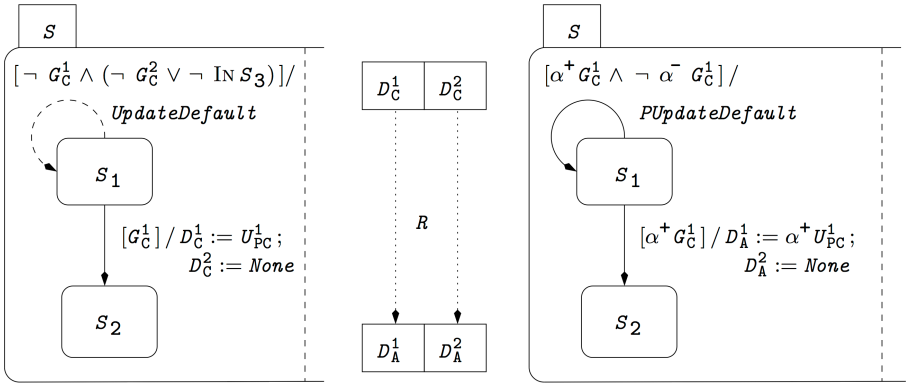

Fig. 10: Structure-preserving abstraction using partial update-functions

operator $AbsBy^{+}_{HA}$ constructing an overapproximation from a given HA and a given abstraction function. This operator implements the ideas of Sec. III-B and Sec. III-D in a compact manner. Basically, we decompose the HA in its defining SA and build the overapproximation of each SA independently. In order to construct the overapproximation of a SA, we have to abstract guards and update-functions of each transition of the SA. Additionally, we have to introduce self-transitions in the abstracted SA if the abstracted guard of a transition becomes weaker than the original one.

Furthermore, we have defined an operator $AbsBy^{-}_{CTL}$ constructing the underapproximation for a given CTL-formula and a given abstraction function. In comparison to defining abstracted hierarchical automata, $AbsBy^{-}_{CTL}$ is constructed directly and straightforwardly. First, the operator traverses a formula in an inductive manner getting access to all atomic propositions that are defined on the infinite data space. Second, the operator constructs underapproximations from these atomic propositions using the operator α^{-} (cf. Sec. III-A).

IV. Model Checking Statecharts

Based on the abstraction theory of Sec. III we have designed and implemented two tactics to verify Statecharts more efficiently. Fig. 11 gives an overview of the framework describing the architecture at an abstract level[2]. On the left side, the theorem prover is depicted including all developed Isabelle/HOL-theories. In principle, proof obligations on Statecharts can be derived inside Isabelle/HOL based on these theories. However, proofs are not fully automated in the prover. They are often complex and exhausting processes involving many interactive steps. To overcome this, we have developed two tactics external to Isabelle/HOL that are invoked via the oracle interface – a specific Isabelle/HOL interface for plugging in additional support tools. The first tactic can be used to check CTL-properties for Statecharts that are defined on finite data spaces. This tactic uses the model checker SMV [36], [37] checking the properties automatically. The second tactic can be used to abstract Statecharts that are defined on infinite data spaces. This tactic implements an automatic predicate abstraction and invokes SVC[3][39] evaluating propositional properties including arithmetic. Note, the second tactic is limited to formulas of the universal fragment of CTL because it includes a property-preserving abstraction algorithm supporting this subclass only.

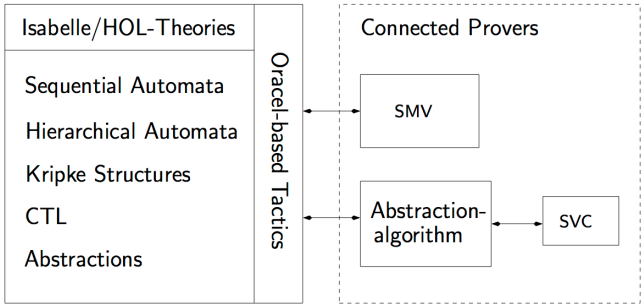

Fig. 11: Framework architecture: Isabelle-theories and connected provers

Applying the first tactic, a user of Isabelle/HOL is able to check whether a temporal formula F is satisfied by the behaviour decription of a given HA representing a finite Statecharts specification.

$$HA \models_{HA} F$$

Therefore, the tactic analyses the internal term structure of the proof state in Isabelle/HOL and translates the collected information into the input language of SMV [28]. Of course, it may happen that SMV will not be able to verify the proof obligation successfully. Consequently in this case, the tactic fails. Otherwise the tactic returns true.

If a user is interested in verifying an infinite Statecharts specification, he has to apply the second tactic. In a first step, the proof obligation is reduced to the following proof state assuming that a property-preserving abstraction is used.

$$(HA\ AbsBy^{+}_{HA}\ R) \models_{HA} (F\ AbsBy^{-}_{CTL}\ R)$$

Abstracted counterparts to HA and F are defined using the operators $AbsBy^{+}_{HA}$ and $AbsBy^{-}_{CTL}$ for a given abstraction

[2]Note, the whole abstraction theory including tactics is not part of [8]. However, interested readers are welcome to request the sources by email to the authors.

[3]The current version is ported to Isabelle 2013-2 and uses CVC3 [38], because SVC will be not longer supported by all plattforms.

function R (cf. Sec. III-D). If we want to prove the proof state in Isabelle/HOL, we must exploit these definitions. In contrast to this, the tactic ignores the definitions and uses an algorithm external to Isabelle/HOL that constructs the abstraction more efficiently. Note, the tactic is applicable only if the following assumptions are satisfied.

1) R respects the structure of the data space satisfying the wellformedness property of Definition 3.1,
2) R defines a predicate abstraction interpreting the infinite data space as a finite set of atomic propositions on it, whereby each of them is represented as a boolean variable inside the abstract data spaces, and
3) F is a formula of $\forall$CTL.

For the design of the abstraction algorithm, we reuse existing work for abstracting ordinary transition systems [34] and adapt this approach to hierarchical state systems. Accordingly, we also use adjunction theorems of Galois connections defining α^+ and α^- different to our Isabelle/HOL-definitions of Sec. III-A.

$$\alpha^+ P_C \equiv_{df} \bigwedge \{ P_A \mid P_C \implies_P \gamma P_A \}$$
$$\alpha^- P_C \equiv_{df} \bigvee \{ P_A \mid \gamma P_A \implies_P P_C \}$$

The identifier P_C and P_A represent predicates on the concrete and abstract data space. An implementation of the overapproximation α^+ is sufficient, because α^- can be expressed using α^+ (cf. Sec. III-D). To calculate the overapproximation of a predicate P_C, we have to implement two steps. First, we check the proof obligation $P_C \implies_P \gamma P_A$ for all P_A of the abstract predicate type using SVC. Second, we conjugate all predicates, where the check was successful.

To implement this procedure, we need an algorithm calculating the predicate transformer γ for a given predicate efficiently. Thanks to the guaranteed properties of predicate abstraction, we are able to define γ as a substitution. For a given abstract predicate we replace a contained boolean variable by it's corresponding atomic proposition, which is defined on the concrete data space only. If we do so for all contained variables, we obtain a concrete predicate as a result.

We refer to the Safety Injection System (cf. Example I-C) for an illustration of the substitution. There we have defined a single integer variable *pressure*, which is used in both guards and actions of the Statechart (cf. Fig. 3). Consequently we need to abstract these predicates algorithmically. Assuming a user of the tactic likes to interpret *pressure* by the atomic propositions $pressure < 10$ and $pressure < 20$ we introduce two boolean variables B_1 and B_2 for the abstract data space representing, whether the propositions are satisfied or not.

For example, if we like to determine the overapproximation of the guard $permit \leq pressure$ we must check for all elements of the abstract predicate type, whether they ensure the guard or not. For example, we must check for the abstract predicate $B_1 \vee_P B_2$ the following statement[4].

$$20 \leq pressure \implies_P \gamma\,(B_1 \vee_P B_2)$$

First, we replace B_1 and B_2 by their corresponding predicates.

$$\begin{aligned} & (B_1 \vee_P B_2)\,[pressure < 20/B_1, pressure < 10/B_2] \\ \Leftrightarrow\ & (pressure < 20 \vee_P pressure < 10) \end{aligned}$$

Afterwards, we evaluate the statement above to false. The overapproximation of the guard is then built as the conjunction of all possible predicates that pass the test (cf. adjunction theorem defining α^+).

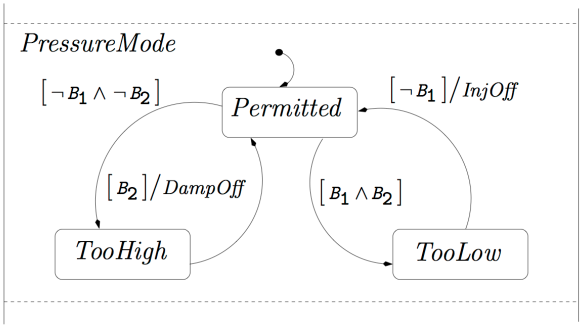

Fig. 12: Abstracted specification of the SA *PressureMode*

Note, the abstract predicate type represents atomic propositions only. Consequently, we can use a finite set of formulas based on the disjunctive normal form to represent the whole predicate space. Nevertheless, we do not check all predicates of this set, because the complexity is too high. Assuming k represents the number of boolean variables used to represent the propositions inside the abstract data space, we obtain 2^{2^k} proof obligations.

To overcome this problem, we use a qualified strategy selecting predicates, which was proposed by Saïdi and Shankar [34]. They proved that it is sufficient for calculating an overapproximation, to do the check for all disjunctions of literals, whereby a literal is a boolean variable or the negation of it. So the complexity can be reduced to at most $3k - 1$ proof obligations. Fig. 13 shows the selection for a set of atomic propositions, which are constructed using two boolean variables.

To sum up, this selection strategy helps to calculate the overapproximation of a predicate more efficiently. For example, to overapproximate a guard G, we build the conjunction of all possible disjunctions of literals that pass the test for G. Afterwards, we simplify the result and convert it into disjunctive normal form.

So far, we have introduced how propositional predicates defined on an infinite data space can be abstracted using a predicate abstraction. For the abstraction of a whole HA, we replicate the idea of Sec. III-D2 of decomposing the HA in it's defining SA. Subsequently we abstract each SA independently. In contrast to defining $AbsBy^+_{HA}$, we calculate overapproximations in both guards and action predicates algorithmically.

Fig. 12 shows the abstracted SA *PressureMode* of the Safety Injection System. Inside the original SA *PressureMode* (cf. Fig. 3), the data variable *pressure* is used for guards only. Hence, we transfer actions into the abstract model without changing them. Furthermore, all guards can be precisely expressed using the boolean variables B_1 and B_2. Accordingly, we are not introducing self-transitions to represent implicit behaviour. Such behaviour must be modeled explicitly only if the abstracted guard is weaker than the original counterpart.

To calculate the abstraction of the SA *Measuring*, we need to abstract all actions of the self-transitions (cf. Fig. 3). The only part of an action, that involves data variables, is

[4]For the sake of simplicity we assume, that 20 is assigned to the constant *permit*.

the update-function. Consequently, we have to abstract the update-functions INC *pressure* and DEC *pressure*. Therefore, we interpret an update-function as a binary predicate defined over the pre and post state. Following common notational conventions, we write the post state of (boolean) variable B_1 as B_1'. Unfortunately in this particular example, we are not able to determine a precise overapproximation of the update-functions because the preceded guards give no information on the data variable *pressure*. Consequently, incrementing or decrementing *pressure* maps into the whole abstract data space. The only effect, which we omit here, is $B_1' \wedge \neg\ B_2'$ because this configuration is obviously not satisfiable. Accordingly, the abstraction of an update-function results in the following predicate.

$$(B_1' \wedge B_2') \vee (\neg\ B_1' \wedge B_2') \vee (\neg\ B_1' \wedge \neg\ B_2')$$

The predicate is a disjunction with three disjoint parts. We represent each part as an abstract update-functions. Note, that the notation of Statecharts allows one update-function per transition only. Consequently, we have to duplicate the self-transitions in the example obtaining three abstracted self-transitions for each (cf. Fig. 14).

V. CONCLUSIONS

We have presented here an approach to model checking of Statecharts with data that combines a full formalisation of the original Statmate semantics of Statecharts in Isabelle/HOL with abstraction techniques, also formalised in Isabelle/HOL, to finally check the abstracted Statecharts in the model checker SMV and additionally use an integration with the SVC validity checker to solve proof obligations resulting from the abstraction process. The main issue of this paper is to give an overall impression of this project that is the core of Steffen Helke's PhD thesis [18].

We finalise this paper with a few concluding remarks concerning the presented parts. The formalisation of the syntax and semantics of Statecharts in Isabelle/HOL has only been reported on partially in this paper; the semantics has been left out. We only wanted to give a gist of the level of explicitness we have used to represent Statecharts in Higher Order Logic. We used type definitions for SA and HA thereby making wellformedness conditions implicit. More importantly, we used explicit polymorphic HOL types to represent the data contained in a Statechart. This makes the representation very concise and also very efficient. Since the data types of our object (the Statecharts) are (generic data) types of the logic HOL, we can exploit a lot of the existing proof infrastructure for proving about Statecharts. On the other hand, explicit formalisations about types become quite tricky. We have, however, illustrated in this work that it is just about possible to formalise notions of partitions of data space in this model.

At the same time, the above described – slightly "shallow" – embedding of data containing Statecharts, is ideally suited for the design of automated tactics. We have here, thus, presented the abstraction techniques we employ to use the type information efficiently to produce proof obligations rising from abstracting Statecharts. Although fairly shallow, the formalisation of Statecharts is deep enough to enable meta theoretical proofs. We have omitted those as they are not in the centre of interest here.

Finally, we showed how we integrate, on the practical side, model checkers and validity checkers to round off the verification process.

In our current research we try to further extend the practicality of model checking Statecharts. Two current projects that show the emphasis of our current and future work are a front-end tool for lay-outing Statecharts before they are fed into the verification process and another project build on MDD (*Model Driven Development*) to integrate the tool chain for Statecharts.

REFERENCES

[1] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[2] J. Spivey, *The Z Notation – A Reference Manual*, 2nd ed. Prentice Hall, 1992.

[3] B. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall, 2005.

[4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT-Press, 2000.

[5] E. Mikk, Y. Lakhnech, and M. Siegel, "Hierarchical Automata as Model for Statecharts," in *Proceedings of Asian Computing Science Conference (ASIAN)*, ser. LNCS. Springer, 1997, vol. 1345.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[7] E. Mikk, "Semantics and Verification of Statecharts," Ph.D. dissertation, Christian Albrechts Universität Kiel, Germany, 2000.

[8] S. Helke and F. Kammüller, "Formalizing statecharts using hierarchical automata," *Archive of Formal Proofs*, 2010.

[9] N. Day, "A Model Checker for Statecharts," Department of Computer Science, University of British Columbia, Tech. Rep. TR 93–35, 1993.

[10] J. Joyce and C.-J. Seger, "The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover," in *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*, ser. LNCS, J. Joyce and C.-J. Seger, Eds. Springer, 1994, vol. 780, pp. 185–198.

[11] M. Gordon and T. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] A. Thums, G. Schellhorn, F. Ortmeier, and W. Reif, "Interactive verification of statecharts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. LNCS, H. Ehrig, Ed. Springer, 2004, vol. 3147, pp. 355–373.

[13] W. Reif, "The KIV System: Systematic Construction of Verified Software," in *International Conference on Automated Deduction (CADE 1992)*, ser. LNCS, vol. 2392. Springer, 1992, pp. 753–757.

[14] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF - A Mechanised Logic of Computation*, ser. LNCS. Springer, 1979, vol. 78.

[15] S. Owre, J. Rushby, , and N. Shankar, "PVS: A Prototype Verification System," in *International Conference on Automated Deduction (CADE)*, ser. LNCS, D. Kapur, Ed., vol. 607. Springer, 1992, pp. 748–752.

[16] I. Traore, "An Outline of PVS Semantics for UML Statecharts," *Journal of Universal Computer Science*, vol. 6, no. 11, pp. 1088–1108, 2000.

[17] D. Aredo, "Formal Development of Open Distributed Systems: Integration of UML and PVS," Ph.D. dissertation, Department of Informatics, University of Oslo, Norway, 2004.

[18] S. Helke, "Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion," Ph.D. dissertation, Fakultt IV, Technische Universität Berlin, Germany, 2007.

[19] Y. Meller, O. Grumberg, and K. Yorav, "Verifying behavioral UML systems via CEGAR," in *Integrated Formal Methods (IFM)*, ser. LNCS, vol. 8739. Springer, 2014, pp. 139–154.

[20] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann, "Implementing Statecharts in Promela/SPIN," in *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*. IEEE Computer Society, 1999, pp. 90–101.

Normal form (2^{2^k})

$B_1 \wedge B_2$	$B_1 \vee (\neg\ B_1 \wedge B_2)$
$\neg\ B_1 \wedge B_2$	$B_1 \vee (\neg\ B_1 \wedge \neg\ B_2)$
$\neg\ B_1 \wedge \neg\ B_2$	$B_2 \vee (\neg\ B_1 \wedge \neg\ B_2)$
$B_1 \wedge \neg\ B_2$	$B_2 \vee (B_1 \wedge \neg\ B_2)$
$B_1 \vee B_2$	$(B_1 \wedge \neg\ B_2) \vee (\neg\ B_1 \wedge B_2)$
$\neg\ B_1 \vee B_2$	$(B_1 \wedge B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$
$\neg\ B_1 \vee \neg\ B_2$	$(B_1 \wedge \neg\ B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$
$B_1 \vee \neg\ B_2$	$(\neg\ B_1 \wedge B_2) \vee (\neg\ B_1 \wedge \neg\ B_2)$

Approximation $(3^k - 1)$

$B_1 \vee B_2$	B_1
$\neg\ B_1 \vee B_2$	B_2
$\neg\ B_1 \vee \neg\ B_2$	$\neg\ B_1$
$B_1 \vee \neg\ B_2$	$\neg\ B_2$

Fig. 13: Selection strategy for $k = 2$

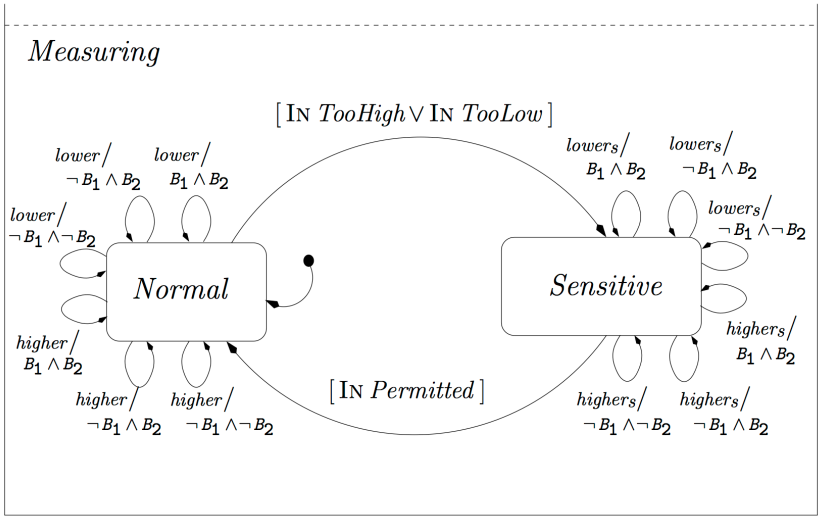

Fig. 14: Abstracted specification of the SA *Measuring*

[21] G. Holzmann, *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.

[22] J.-J. Hiemer, "Statecharts in CSP – Ein Prozessmodell in CSP zur Analyse von Statemate Statecharts," Ph.D. dissertation, Technische Universität Berlin, Germany, 1998.

[23] B. Roscoe and Z. Wu, "Verifying Statemate Statecharts Using CSP and FDR," in *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*, ser. LNCS, Z. Liu and J. He, Eds. Springer, 2006, vol. 4260.

[24] U. Brockmeyer and G. Wittich, "Real-Time Verification of Statemate Designs," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 1998, pp. 537–541.

[25] ——, "Tamagotchis Need Not Die — Verification of STATEMATE Designs," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer, 1998, vol. 1384, pp. 217–231.

[26] S. Helke and F. Kammüller, "Representing Hierarchical Automata in Interactive Theorem Provers," in *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, R. Boulton and P. Jackson, Eds. Springer, 2001, vol. 2152, pp. 233–248.

[27] ——, "Structure Preserving Data Abstractions for Statecharts," in *Proceedings of Formal Techniques for Networked and Distributed Systems (FORTE)*, ser. LNCS, F. Wang, Ed. Springer, 2005, vol. 3731, pp. 305–319.

[28] ——, "Verification of Statecharts Including Data Spaces," in *TPHOLs 2003: Emerging Trends Proceedings*, ser. Technischer Report 189, D. Basin and B. Wolff, Eds. Albert-Ludwigs-Universität Freiburg, 2003, pp. 177–190.

[29] P. Courtois and D. Parnas, "Documentation for Safety Critical Software," in *Proceedings of the 15th International Conference on Software Engineering*, 1993, pp. 315–323.

[30] R. Bharadwaj and C. Heitmeyer, "Verifying SCR Requirements Specifications using State Exploration," in *Proceedings of the ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.

[31] T. Bultan, R. Gerber, and C. League, "Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 1998, pp. 113–123.

[32] D. Dams, "Abstract Interpretation and Partition Refinement for Model Checking," Ph.D. dissertation, Eindhoven University of Technology, Niederlande, 1996.

[33] A. Melton, D. Schmidt, and G. Strecker, "Galois Connections and Computer Science Applications," in *Category Theory and Computer Programming*, ser. LNCS, D. Pitt, S. Abramsky, A. Poigne, and D. Rydeheard, Eds., vol. 240. Springer, 1986, pp. 299–312.

[34] H. Saïdi and N. Shankar, "Abstract and Model Check While You Prove," in *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, ser. LNCS, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 443–454.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2000, pp. 154–169.

[36] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[37] ——, "SMV homepage: The Cadence SMV Model Checker. http://www.kenmcmil.com/smv.html."

[38] C. Barret and C. Tinelli, "CVC3 homepage: Cooperating Validity Checker 3. http://www.cs.nyu.edu/acsys/cvc3/."

[39] C. Barrett, D. Dill, and J. Levitt, "Validity Checking for Combinations of Theories with Equality," in *Formal Methods In Computer-Aided Design*, ser. LNCS, M. Srivas and A. Camilleri, Eds., vol. 1166. Springer, 1996, pp. 187–201.