# Verification of Hierarchical State/Event Systems Using Reusability and Compositionality*

Gerd Behrmann[1], Kim G. Larsen[1],
Henrik R. Andersen[2], Henrik Hulgaard[2], and Jørn Lind-Nielsen[2]

[1] BRICS***, Aalborg University, Denmark
[2] Department of Information Technology, DTU, Denmark

**Abstract.** We investigate techniques for verifying hierarchical systems, i.e., finite state systems with a nesting capability. The straightforward way of analysing a hierarchical system is to first flatten it into an equivalent non-hierarchical system and then apply existing finite state system verification techniques. Though conceptually simple, flattening is severely punished by the hierarchical depth of a system. To alleviate this problem, we develop a technique that exploits the hierarchical structure to *reuse* earlier reachability checks of superstates to conclude reachability of substates. We combine the reusability technique with the successful compositional technique of [13] and investigate the combination experimentally on industrial systems and hierarchical systems generated according to our expectations to real systems. The experimental results are very encouraging: whereas a flattening approach degrades in performance with an increase in the hierarchical depth (even when applying the technique of [13]), the new approach proves not only insensitive to the hierarchical depth, but even leads to improved performance as the depth increases.

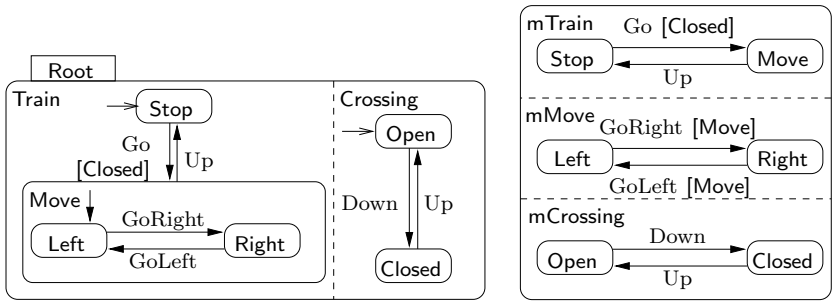
## 1 Introduction

Finite state machines provide a convenient model for describing the control-part (in contrast to the data-part) of embedded reactive systems including smaller systems such as cellular phones, hi-fi equipment, cruise controls for cars, and large systems as train simulators, flight control systems, telephone and communication protocols. We consider a version of finite state machines called state/event machines (SEMs). The SEM model offers the designer a number of advantages including automatic generation of efficient and compact code and a platform for formal analysis such as model-checking. In this paper we focus and contribute to the latter.

In practice, to describe complex systems using SEMs, a number of extensions are often useful. In particular, rather than modeling a complex control

* Supported by CIT, The Danish National Center of IT Research.
*** BRICS: Basic Research in Computer Science, Center of the Danish National Research Foundation

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 163–177, 1999.




(a) Hierarchical system. (b) Flattened system.

**Fig. 1.** (a) A hierarchical model of a toy train. The system is composed of a number of serial, parallel and primitive states. (b) The model after it has been flattened.

as a *single* SEM, it is often more convenient to use a *concurrent* composition of several component SEMs each typically dealing with a specific aspect of the control. Here we focus on an additional *hierarchical* extension of SEMs, in which states of component SEMs are either *primitive* or *superstates* which are themselves (compositions of) SEMs. Figure 1(a) illustrates a hierarchical description of a system with two components, a Train and a Crossing. Inside the Train the state Move is a superstate with the two (primitive) states Left and Right. Transitions within one component may be guarded with conditions on the substates of other components. E.g., the 'Go'-transition may only be fired when the machine Crossing is in the substate Closed.

The STATECHART notation is the pioneer in hierarchical descriptions. Introduced in 1987 by David Harel [10] it has quickly been accepted as a compact and practical notation for reactive systems, as witnessed by a number of hierarchical specification formalisms such as MODECHARTS [11] and RSML [12]. Also, hierarchical descriptions play a central role in recent object-oriented software methodologies (e.g., OMT [15] and ROOM [16]) most clearly demonstrated by the emerging UML-standard [8]. Finally, hierarchical notations are supported by a number of CASE tools, such as STATEMATE [2], OBJECTIME [3], RATIONAL-ROSE [4], and in the forthcoming visualSTATE™version 4.0 [1].

Our work has been performed in a context focusing on the commercial product visualSTATE™ and its hierarchical extension. This tool assists in developing embedded reactive software by allowing the designer to construct and manipulate SEM models. The tool is used to simulate the model, checking the consistency of the model, and from the model automatically generate code for the hardware of the embedded system. The *consistency checker* of visualSTATE™ is in fact a *verification tool* performing a number of generic checks, which when violated in-

dicate likely design errors. The checks include checking for absence of deadlocks, checking that all transitions may fire in some execution, and similarly checking that all states can be entered.

In the presence of concurrency, SEM models may describe extremely large statespaces[1] and, unlike in traditional model checking, the number of checks to be performed by visualSTATE™ is at least linear in the size of the model. In this setting, our previous work [13] offers impressive results: a number of large SEM models from industrial applications have been verified. Even a model with 1421 concurrent SEMs (and $10^{476}$ states) has been verified with modest resources (less than 20 minutes on a standard PC). The technique underlying these results utilises the ROBDD data structure [9] in a *compositional analysis* which initially considers only a few component-machines in determining satisfaction of the verification task and, if necessary, gradually includes more component-machines.

Now facing hierarchical SEMs, one can obtain an equivalent concurrent composition of ordinary SEMs by flattening it, that is, by recursively introducing for each superstate its associated SEM as a concurrent component. Figure 1(b) shows the flattening of the hierarchical SEM in Fig. 1(a) where the superstate Move has given rise to a new component mMove. Thus, verification of hierarchical systems may be carried out using a flattening preprocessing. E.g., demonstrating that the primitive state Left is reachable in the hierarchical version (Figure 1(a)), amounts to showing that the flattened version (Figure 1(b)) may be brought into a system-state, where the mMove-component and the mTrain-component are simultaneously in the states Left and Move.

Though conceptually simple, verification of hierarchical systems via flattening is, as we will argue below (Section 2) and later experimentally demonstrate, severely punished by the hierarchical depth of a system; even when combined with our successful compositional technique of [13] for ordinary SEMs.

To alleviate this problem, we introduce in this paper a new verification technique that uses the hierarchical structure to *reuse* earlier reachability checks of superstates to conclude reachability of substates. We develop the reusability technique for a hierarchical SEM model inspired by STATECHART and combine it with the compositionality technique of [13]. We investigate the combination experimentally on hierarchical systems generated according to our expectations from real systems.[2] The experimental results are very encouraging: whereas the flattening approach degrades in performance with an increase in the hierarchical depth, it is clearly demonstrated that our new approach is not only insensitive to the hierarchical depth, but even leads to improved performance as the depth increases. In addition, for non-hierarchical (flat) systems the new method is an instantiation of, and performs as well as, the compositional technique of [13].

[1] The so-called *state-explosion problem*.

[2] In short, we expect that transitions and dependencies between parts of a well-designed hierarchical system are more likely to occur between parts close to each other rather than far from each other in the hierarchy.

**Related Work**

R. Alur and M. Yannakakis' work on *hierarchical Kripke structures* offers important worst case complexity results for both *LTL* and *CTL* model checking [5]. However, their results are restricted to *sequential* hierarchical machines and use the fact that abstract superstates may appear in several instantiations. In contrast we provide verification results for general hierarchical systems with both sequential and parallel superstates without depending on multiple instantiations of abstract superstates.

Park, Skakkebæk and Dill [14] have found an algorithm for automatic generation of invariants for states in RSML specifications. Using these invariants it is possible to perform some of the same checks that we provide for hierarchical SEMs. Their algorithm works on an approximation of the specification, and uses the fact that RSML does allow internal events sent from one state to another.

## 2 Flattening and Reusability

To see why the simple flattening approach is vulnerable to the hierarchical depth, consider the (schematic) hierarchical system of Fig. 2(a). The flattened version of this system will contain (at least) a concurrent component $mS_i$ for each of the superstates $S_i$ for $0 \leq i \leq 100$. Assume, that we want to check that the state $u$ is reachable. As reachability of a state in a hierarchical system automatically implies reachability of all its superstates, we must demonstrate that the flattened system can reach a state satisfying the following condition:[3]

$$mS_{100}@u \wedge mS_{99}@S_{100} \wedge mS_{98}@S_{99} \wedge \ldots \wedge mS_0@S_1 \ .$$

Consequently, we are faced with a reachability question immediately involving a large number of component SEMs, which in turn means that poor performance of our compositional technique [13] is to be expected. Even worse, realizing all the checks of visualSTATE™ means that we must in similarly costly manners demonstrate reachability of the states $x$, $y$, $z$ and $v$. All these checks contain $mS_{99}@S_{100} \wedge mS_{98}@S_{99} \wedge \ldots \wedge mS_0@S_1$ as common part. Hence, we are in fact repeatedly establishing reachability of $S_{100}$ as part of checking reachability of $x, y, z, u$ and $v$. As this situation may occur at all (100) levels, the consequence may be an exponential explosion of our verification effort.

Let us instead try to involve the hierarchical structure more actively and assume that we have *already* in some previous check demonstrated that $S_{100}$ is reachable (maybe from an analysis of a more abstract version of the model in which $S_{100}$ was in fact a primitive state).

How can we *reuse* this fact to simplify reachability-checking of, say, $u$? Assume first a simple setting (Figure 2(a)), where $S_{100}$ is only activated by transitions to $S_{100}$ itself (and not to substates within $S_{100}$) and transitions in $S_{100}$ are only dependent (indicated by the guard $g$) on substates within $S_{100}$ itself. In this

[3] Here $mS@T$ denotes that the component $mS$ is in state $T$.

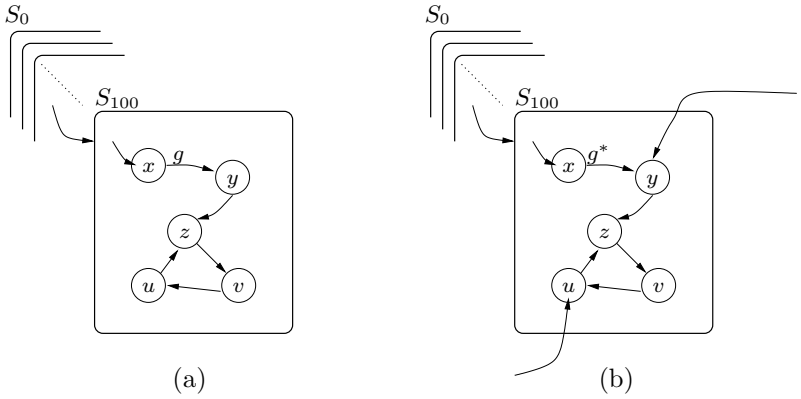



**Fig. 2.** Simple and complex substates.

case we may settle the reachability question by simply analysing $S_{100}$ as a system of its own. In more complex situations (Figure 2(b)), $S_{100}$ may possibly be activated in several ways, including via transitions into some of its substates. Also, the transitions within $S_{100}$ may refer to states outside $S_{100}$ (indicated by the guard $g^*$). In such cases—in analogy with our previous compositional technique [13]—we compute the set of states which *regardless* of behaviour outside $S_{100}$ may reach $u$. If this set contains all potential initial states of $S_{100}$ (in Fig. 2(b) the states $x, y, u$) we may infer from the known reachability of $S_{100}$ that also $u$ is reachable. Otherwise, we will simply extend the collection of superstates considered depending on the guards within $S_{100}$ and the transitions to $S_{100}$.

In the obvious way, transitions between (super)states and their guards determine the pattern of dependencies between states in a hierarchical system. We believe that in good hierarchical designs, dependencies are more likely to exist between states close to each other in the hierarchy rather than states hierarchically far from each other. Thus, the simple scenario depicted in Fig. 2(a) should in many cases be encountered with only small extensions of the considered superstates.

## 3 The Hierarchical State/Event Model

A hierarchical state/event machine (HSEM) is a hierarchical automaton consisting of a number of nested primitive, serial, and parallel states. Transitions can be performed between any two states regardless of their type and level, and are labeled with an event, a guard, and a multiset of outputs. Formally an HSEM is a 7-tuple

$$M = \langle S, E, O, T, Sub, type, def \rangle \tag{1}$$

of states $S$, events $E$, outputs $O$, transitions $T$, a function $Sub : S \to \mathcal{P}(S)$ associating states with their substates, a function $type : S \to \{\mathbf{pr}, \mathbf{se}, \mathbf{pa}\}$ mapping

states to their type (indicating whether a state is **pr**imitive, **se**rial, or **pa**rallel), and a partial function $def : S \hookrightarrow S$ mapping serial states to their default substate. The set of serial states in $S$ is referred to as $R$.

The set of transitions $T \subseteq S \times E \times G \times \mathcal{M}(O) \times S$ where $\mathcal{M}(O)$ is the set of all multisets of outputs, and $G$ is the set of guards derived from the grammar $g ::= g_1 \wedge g_2 \,|\, \neg g_1 \,|\, \mathsf{tt} \,|\, \mathsf{s}$. The atomic predicate $\mathsf{s}$ is a state synchronisation on the state $s$, having the intuitive interpretation that $\mathsf{s}$ is true whenever $s$ is active (we will return to the formal semantics in a moment). We use $t = (s_t, e_t, g_t, o_t, s'_t)$ to range over syntactic transitions (with source, event, guard, outputs and target respectively).

For notational convenience we write $s \searrow s'$ whenever $s' \in Sub(s)$. Furthermore we define $\searrow^+$ to be the transitive closure, and $\searrow^*$ to be the transitive and reflexive closure of $\searrow$. If $s \searrow^+ s'$ we say that $s$ is above $s'$, and $s'$ is below $s$. The graph $(S, \searrow)$ is required to be a tree, where the leaves and only the leaves are primitive states, i.e., $\forall s : type(s) = \mathbf{pr} \Leftrightarrow Sub(s) = \emptyset$.

For a set of states $I$, $\mathrm{lca}(I)$ denotes the least common ancestor of $I$ with respect to $\searrow$. For a state $s$, $\mathrm{lsa}(s)$ denotes the *least serial ancestor* of $s$. The *scope* of a transition $t$ is denoted $\chi(t)$ and represents the least common serial ancestor of the states $s_t$ and $s'_t$. For those transitions in which such a state does not exist, we say that $\chi(t) = \$$, where $\$$ is a dummy state above all other states, i.e., $\forall s \in S : \$ \searrow^+ s$.

A *configuration* of an HSEM is an $|R|$-tuple of states indexed by the serial states. The *configuration space* $\Sigma$ of an HSEM is the product of the set of substates of each serial state,

$$\Sigma = \prod_{s \in R} Sub(s) \ . \tag{2}$$

The *projection* $\pi_s : \Sigma \to Sub(s)$ of a configuration $\sigma$ onto a serial state $s$ yields the value of $s$ in $\sigma$. The projection of a configuration onto a parallel or primitive state is undefined. A state $s$ is active in $\sigma$ if either $s$ is the root state, the parent of $s$ is an active parallel state, or the parent is an active serial state and $s$ is the projection of $\sigma$ onto the parent. In order to formalise this we define the infix operator $\mathsf{in}$ as

$$s \,\mathsf{in}\, \sigma \Leftrightarrow \forall s' \searrow^+ s : s' \in R \Rightarrow \pi_{s'}(\sigma) \searrow^* s \ . \tag{3}$$

We denote by $\Sigma_s = \{\sigma \,|\, s \,\mathsf{in}\, \sigma\}$ the set of configurations in which $s$ is active.

Let $\sigma \models g$ whenever $\sigma$ satisfies $g$. The interpretation of a guard is defined as: $\sigma \models \mathsf{tt}$ (any configuration satisfies the true guard), $\sigma \models s$ iff $s \,\mathsf{in}\, \sigma$, $\sigma \models g_1 \wedge g_2$ iff $\sigma \models g_1$ and $\sigma \models g_2$, and $\sigma \models \neg g$ iff $\sigma \not\models g$. A pair $(e, \sigma)$ is said to enable a transition $t$, written $(e, \sigma) \models t$, iff $e = e_t$, $s_t \,\mathsf{in}\, \sigma$, and $\sigma \models g_t$.

Before introducing the formal semantics, we summarise the intuitive idea behind a computation step in HSEM. An HSEM is event driven, i.e., it only reacts when an event is received from the environment. When this happens, a maximal set of non-conflicting and enabled transitions is executed, where non-conflicting means no transitions in the set have nested scope. This conforms

to the idea that the scope defines the area affected by the transition. When a transition is executed, it forces a state change to the target. All implicitly activated serial states enter their default state. In fact, a transition is understood to leave the scope and immediately reactivate it.

Formally, a set $\Delta \subseteq T$ is *enabled* on $(e, \sigma)$ if $\forall t \in \Delta : (e, \sigma) \models t$, $\Delta$ is *compatible* if $\forall t, t' \in \Delta : (t \neq t' \Rightarrow \chi(t) \not\searrow^* \chi(t'))$, and $\Delta$ is *maximal* if $\forall \Delta' \subseteq T : \Delta \subset \Delta' \Rightarrow \Delta'$ is incompatible or disabled on $(e, \sigma)$. The semantics of an HSEM is defined in terms of a transition relation $\rightarrow \subseteq \Sigma \times E \times \mathcal{M}(O) \times \Sigma$. We have $\sigma \xrightarrow{e/o} \sigma'$ if and only if there exists a set $\Delta \in T$, such that:[4]

1. $\Delta$ is compatible, enabled on $(e, \sigma)$, and maximal,
2. $o = \uplus_{t \in \Delta} o_t$,
3. $\forall t \in \Delta : s'_t \,\mathsf{in}\, \sigma'$,
4. $\forall t \in \Delta, s \in S : s \,\mathsf{in}\, \sigma' \wedge \mathit{type}(s) = \mathbf{se} \wedge \chi(t) \searrow^+ s \not\searrow^+ s'_t \Rightarrow \pi_s(\sigma') = \mathit{def}(s)$, and
5. $\forall s \in R : (\forall t \in \Delta : \chi(t) \not\searrow^* s) \Rightarrow \pi_s(\sigma) = \pi_s(\sigma')$.

The second constraint defines the output of the transition, the third that all targets are active after the transition, the fourth that all implicitly activated serial states (those not on the path between the scope and the target of any transition) are recursively set to their default state, and the last that all states not under the scope of any transition remain unchanged.

## 4 Reusable Reachability Checking

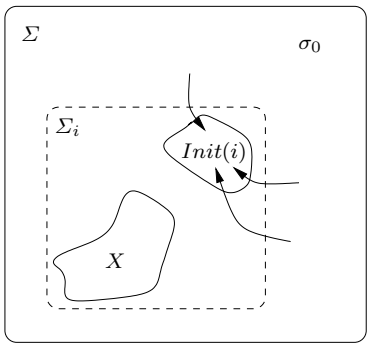
The consistency checker of visualSTATE™ performs seven predefined types of checks, each of which can be reduced to verifying one of two types of properties. The first property type is *reachability*. For instance, visualSTATE™ checks for absence of dead code in the sense that all transitions must be possibly enabled and all states must be possibly entered. E.g., checking whether a transition $t$ will ever become executable is equivalent to checking whether its guard is satisfiable, i.e., whether we can reach a configuration $\sigma$ such that $\exists e : (e, \sigma) \models t$. Similarly, checking whether a state $s$ may be entered amounts to checking whether the system can reach a configuration within $\Sigma_s$.

The remaining two types of consistency checks reduce to a check for absence of *local deadlocks*. A local deadlock occurs if the system can reach a configuration in which one of the superstates will never change value nor be deactivated no matter what sequence of events is offered.

In the following two sections we present our novel technique exploiting reusability and compositionality through its application to reachability analysis only. In the full version of this paper [7] and in [6] the applicability of the technique to local deadlock detection is given in detail.

In general, a reachability question involves a set of goal configurations $X \subseteq \Sigma$. The question posed is whether $X$ is reachable in the sense that there exists a

[4] The symbol $\uplus$ denotes multiset union




(a) Reusing reachability of $i$.

```
X := {σ | σ is a goal conf.}
while Init(i) ⊈ X and σ0 ∉ X do
begin
    X' := Bi(X) ∪ X
    if X ≠ X' then
        X := X'
    else if Init(i) ∩ X ≠ ∅ then
        i := lsa(i)
    else
        return false
end
return true
```

(b) Algorithm 1.

**Fig. 3.** Reusable reachability check.

sequence of events such that the system starting at the initial configuration $\sigma_0$ enters a configuration in $X$. To explain the idea of *reusability*, let $i$ be a state such that $X \subset \Sigma_i$, i.e., reachability of any configuration within $X$ implies reachability of the state $i$ (see Fig. 3(a)). Notice that such a state always exists, e.g., the root will satisfy this condition for any $X \neq \Sigma$. Also, if $X = \Sigma_s$ any superstate of $s$ will suffice. The question we ask is how existing information about reachability of $i$ may be reused to simplify reachability-checking of $X$. The simple case is clearly when $i$ is not reachable. In this case there is no way that $X$ can be reachable either, since $X$ only contains configurations where $i$ is active. Since we expect (or hope) most of the reachability questions issued by visualSTATE™ to be true this only superficially reduces the number of computations. However, although more challenging, we can also make use of the information that $i$ is reachable, as explained below.

Knowing $i$ is reachable, still leaves open which of the configurations in $\Sigma_i$ are in fact reachable (and in particular if any configuration in $X$ is). However, any reachable configuration $\sigma$ in $\Sigma_i$ must necessarily be reachable through a sequence of the following form:[5]

$$\sigma_0 \rightsquigarrow \sigma_1 \rightsquigarrow \cdots \underbrace{\sigma_n}_{\notin \Sigma_i} \rightsquigarrow \underbrace{\sigma_{n+1} \rightsquigarrow \sigma_{n+2} \rightsquigarrow \cdots \sigma_{n+k} \rightsquigarrow \sigma}_{\in \Sigma_i} \ . \tag{4}$$

Let the initial configurations for $i$, $Init(i)$, be the configurations for which $i$ is active and which are reachable in one step from a configuration in which $i$ is inactive (e.g., the configuration $\sigma_{n+1}$ in (4); see also Fig. 3(a)). Algorithmically, (an over approximation of) $Init(i)$ may be obtained effectively in a straightforward manner directly from the syntactic transitions. Consider then the following

[5] Here $\sigma \rightsquigarrow \sigma'$ abbreviates $\exists e, o : \sigma \xrightarrow{e/o} \sigma'$.

backwards step computation:

$$B_i(Y) = \{\sigma \in \Sigma_i \mid \exists\, \sigma' : \sigma \rightsquigarrow \sigma' \wedge \sigma' \in Y\} \tag{5}$$

that is, $B_i(Y)$ is the set of configurations with $i$ active, which in one step may reach $Y$. To settle reachability of $X$, we iteratively apply $B_i$ according to Algorithm 1 in Fig. 3(b). Reachability of $X$ may now be confirmed if either the initial configuration of the system is encountered ($\sigma_0 \in X$) or the backwards iteration reaches a stage with *all* initial states for $i$ included ($Init(i) \subseteq X$). Dually, if the backwards iteration reaches a fixed point ($X^* = B_i(X^*)$), reachability of $X$ can be rejected if *no* initial configuration for $i$ has been encountered (i.e., $X^* \cap Init(i) = \emptyset$.). If some but not all of the initial configurations for $i$ have been encountered, the analysis does not allow us to conclude on the reachability of $X$ based on reachability of $i$. Instead, the backwards iteration is continued with $i$ substituted with its directly enclosing, serial superstate.

The reusability approach depends on a previous reachability check of the non-primitive states in the system. Since this is itself a series of reachability checks the above approach can be applied immediately if we perform a preorder traversal of the state tree determining reachability of each state as we encounter them, reusing the previous checks. If a state turns out to be unreachable we can immediately conclude that all substates are unreachable.

## 5 Compositional Reachability Checking

The reusable reachability analysis offered by the algorithm of Fig. 3(b) is based on the backward step function $B_i$. An obvious drawback is that computation of $B_i$ requires access to the global transition relation $\rightarrow$. In this section we show how to incorporate the *compositional* technique of [13] by replacing the use of $B_i$ with a backwards step function, $CB_I$, which only requires partial knowledge about the transition relation corresponding to a selected and minimal subsystem. The selection is determined by a *sort* $I$ identifying the set of superstates currently considered. Initially, the sort $I$ only includes superstates directly relevant for the reachability question. Later, also superstates on which the initial sort behaviourally depend will be included.

A subset $I$ of $R$ (the set of serial states) is called a *sort* if it is non-empty, and is convex in the sense that $u \in I$ whenever $\mathrm{lca}(I) \searrow^* u \searrow^* y$ for some $y \in I$.[6] For any nonempty set $A \subseteq R$ the set $Convex(A)$ denotes the minimal superset of $A$ satisfying the properties for a sort. The state $\mathrm{lca}(I)$ of a sort will turn out to be an ideal choice for the state $i$ used in the reusable reachability algorithm in the previous section.

Two configurations $\sigma$ and $\sigma'$ are said to be *$I$-equivalent*, written $\sigma =_I \sigma'$, whenever they agree on all states in $I$. More formally

$$\sigma =_I \sigma' \iff \forall s \in I : \pi_s(\sigma) = \pi_s(\sigma') \ . \tag{6}$$

[6] Only if $\mathrm{lca}(I)$ is a serial state does this imply that $\mathrm{lca}(I) \in I$.

For notational convenience we write $\Sigma_I = \Sigma_{\mathrm{lca}(I)}$. A set $P \subseteq \Sigma_I$ of configurations is said to be $I$-sorted in case

$$\forall \sigma, \sigma' \in \Sigma_I : \sigma =_I \sigma' \Rightarrow (\sigma \in P \Leftrightarrow \sigma' \in P) \quad . \tag{7}$$

Notice that we require that $P \subseteq \Sigma_I$ for $P$ to be $I$-sorted. This follows from the idea that the reusable reachability check restricts the analysis to the subsystem with root $\mathrm{lca}(I)$. $P$ being $I$-sorted intuitively means that it only depends on states within $I$. Using ROBDDs allows for very compact representations of $I$-sorted sets as the parts of the configuration set outside the sort may be ignored.

From an $I$-sorted set $X$ we perform within $\Sigma_I$ a *compositional* backwards computation step by including all configurations with $\mathrm{lca}(I)$ active which, irrespective of the behaviour of the superstates outside $I$, can reach $X$. One backward step is given by the function $CB_I$ defined by:

$$CB_I(X) = \{\sigma \in \Sigma_I \mid \forall \sigma' \in \Sigma_I : \sigma =_I \sigma' \Rightarrow \exists\, \sigma'' \in X : \sigma' \rightsquigarrow \sigma''\} \quad . \tag{8}$$

Observe that $CB_I$ is monotonic in both $X$ and $I$. By iterating the application of $CB_I$, we can compute the set of configurations that are able to reach a configuration within $X$ independently of behaviours outside the considered sort $I$. This is the minimum fixed-point $\mu Y.X \cup CB_I(Y)$ which we refer to as $CB_I^*(X)$. In an ROBDD based implementation, the global transition relation may be partitioned into conjunctive parts with contributions from each superstate. Crucial for our approach is the fact that $CB_I$ may be computed without involving the global transition relation directly, but only the parts of the partitioning relevant for the considered sort $I$. We refer to [13] for a similar observation for flat SEMs.

If computing $CB_I^*(X)$ does not resolve the reachability question, we extend the sort $I$ with the states $Dep(I)$ (see Fig. 4) that the behaviour of $I$ depends on. Now, extending $Dep$ to sets in the obvious pointwise manner, we say that a sort $I$ is *dependency closed* provided $Dep(I) \subseteq I$. The basic properties of $CB_I^*$ are captured by the following lemma:

**Lemma 1.** *Let $X$ be an $I$-sorted subset of $\Sigma$. For all sorts $I, J$ with $I \subseteq J$ the following holds:*

1. $CB_I^*(X) \subseteq CB_J^*(X)$,
2. $CB_J^*(X) = CB_J^*(CB_I^*(X))$,
3. *$I$ dependency closed* $\wedge\ Init(I) \cap CB_I^*(X) = \emptyset \Rightarrow CB_I^*(X) = CB_J^*(X)$.

The first property guarantees that we may conclude $X$ reachable as soon as all initial configurations of some known reachable state is encountered (say the global initial state). The second property allows us to reuse backwards computations performed with one sort as the starting point for a larger sort. The last property allows reachability of $X$ to be rejected in case $I$ is dependency closed and no initial configuration of $I$ has been encountered (as no new configurations will be encountered by extending the sort).

Algorithm 2 in Fig. 5 is the result of using the compositional backward step $CB_I$ instead of $B_i$, with $Minsort(X)$ offering a minimal sort for the set of

configurations $X$. When the algorithm returns false, none of the configurations in $X$ are reachable. If true is returned, it means that at least one goal configuration is reachable *under the assumption* that $\mathrm{lca}(I)$ is known to be reachable.

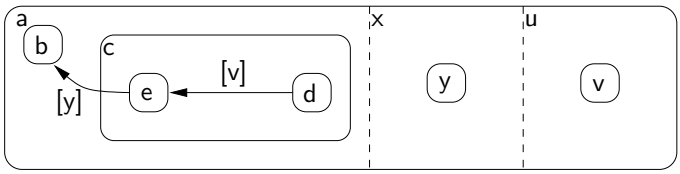



**Fig. 4.** State c depends on u, due to the transition from e to u and since u is the parent of v upon which the transition is guarded. Likewise does the transition from e to b create dependencies from state a (the scope of the transition) to state c (the parent of the source) and x (the parent of the state upon which the transition is guarded).

$I := Minsort(X)$
**while** $Init(I) \not\subseteq X$ **and** $\sigma_0 \notin X$ **do**
**begin**
  $X' := CB_I(X) \cup X$
  **if** $X \neq X'$ **then**
    $X := X'$
  **else if** $Dep(I) \not\subseteq I$ **then**
    $I := Convex(I \cup Dep(I))$
  **else if** $Init(I) \cap X \neq \emptyset$ **then**
    $I := I \cup \{\mathrm{lsa}(\mathrm{lca}(I))\}$
  **else**
    **return** false
**end**
**return** true

**Fig. 5.** Algorithm 2, reusable and compositional reachability.

## 6 Experimental Results

To evaluate our approach, the runtime and memory usage of an experimental implementation using our method is compared to an implementation for flat systems. We will refer to the first as the *hierarchical checker* and the second as the *flat checker*. Both checkers utilise the compositional backwards analysis and use ROBDDs to represent sets of states and transition relations, but only the

hierarchical checker uses the reusable reachability check. Only satisfiability of transitions is verified, i.e., whether the system for each transition can reach a configuration such that the transition is enabled. The hierarchical checker additionally checks whether non-primitive states are reachable since this is necessary in order to apply the reusable reachability check.

The two implementations where first compared on flat test cases previously used in [13]. Without going into details, adding the reusable reachability checking did not degrade performance.

The lack of adequate examples has forced us to develop a method to generate scalable hierarchical systems. It is possible to scale both the maximum nesting depth, the number of substates of parallel and serial states, and the total number of serial states (which is equivalent to the number of automata in the flat system). Serial and parallel states alternate on the path from the root to the leaves starting with a parallel state. The number of states are adjusted by pruning the state tree, i.e., just because a system has a nesting depth of 12 does not mean, that all leaves are placed at level 12 (the size of such a system would be extreme). If the generated system is not deep enough to accommodate the number of wanted states with the chosen width of parallel and serial states, the width is expanded. E.g., a system with 100 serial states and depth 1 will have a parallel root with 100 substates.

As stated in the introduction, we believe that in good designs, dependencies are more likely to be local. The generated test cases reflect this by only including transitions between nearby states. The guards are created at random, but the probability that a guard synchronises with a given state is inverse exponential to the distance between the scope of the transition and the state. The number of transitions is proportional to the number of serial states. Transitions are arranged so that any state is potentially reachable, i.e., if the transitions were unguarded all states would be reachable. Events are distributed such that the system is guaranteed to be deterministic.

Figure 6 shows the runtime of both the hierarchical and the flat checker for a fixed number of substates in parallel and serial states (4 in parallel and 3 in serial), but with varying depth and number of serial states (which corresponds to the number of automata in the equivalent flat system). It is interesting to notice that the runtime of the hierarchical checker is much more consistent than that of the flat checker, i.e., the runtime of the flat checker does vary greatly for different systems generated with the same parameters, as the depth is increased. Although each grid point of the figures shows the mean time of 20 measurements,[7] it is still hard to achieve a smooth mesh for the flat checker.

While the flat checker suffers under the introduction of a hierarchy, the hierarchical checker actually benefits from it. How can it be that the addition of a hierarchy decreases the runtime of the hierarchical checker? As stated earlier, we

[7] It took about two days to run the 1920 cases providing the basis of the 96 depicted grid points. The test was performed on a Sun UltraSparc 2 with two 300 MHz processors and 1 GB of RAM (although the enforced limit of $10^6$ nodes assured a maximal memory consumption below 20 MB).

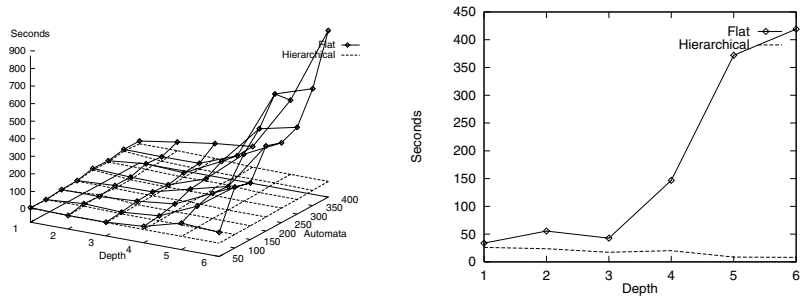



**Fig. 6.** Comparison of the runtime of the flat and hierarchical checker. (a) The runtime of both checkers is plotted as a function of the nesting depth and number of automata/serial states. (b) A slice of the mesh where the number of automata is 300. As can be seen, the runtime of the flat checker explodes as the depth increases, whereas the runtime of the hierarchical checker decreases slightly.

believe that a good hierarchical design is modular in its nature. If a particular system cannot be easily described using a hierarchy, this is probably due to too many interdependencies in the system. Our test cases incorporate this idea: In a system with depth one, the distance between any two states in two different superstates will be constant. Hence the probability with which a guard refers to a state in another superstate is constant, i.e., it is likely that many superstates depend on each other.

It is worth noticing, that our method allows us to drop reachability questions which result in an unreachable initial *lca* state (in this case the answer will be *no*). The number of questions dropped because of this is proportional to the number of unreachable states in the test case. This number varies, but is most of the time below 5-10% of the total number of checked states (primitive states are not checked), although 50% unreachable states have been observed. Testing whether the non-primitive states are reachable is very fast compared to the time it takes to check the transitions. It is noteworthy that some test cases, even without any unreachable states, showed a difference in runtime with a factor of over 180 in favor of the hierarchical checker compared to the flat one.

Table 1 provides further information on the performance of the hierarchical checker on a single case with depth 12, 399 serial states, 3 substates in each parallel state, and 4 substates in each serial state.[8] This results in a total of 1596 transitions, although optimisations did allow the checker to verify 331 transitions without performing a reachability analysis, leaving 1265 checks (not counting reachability checking of non-primitive states). The table shows the number of questions distributed over the initial and final depth of the *lca* state of the questions. For instance we can see that 59 of the questions starting at depth

[8] This corresponds to a state space of $10^{240}$ configurations

5 are verified without including additional states toward the root, but that 2 questions needed to expand the sort such that the final answer was found at depth 3. It is apparent that a large number of questions is verified in terms of a small subsystem. This illustrates why our method does scale as well as it does. This particular system is verified within 26 seconds using the hierarchical checker, whereas the flat checker uses 497 seconds.

**Table 1.** Distribution of reachability questions. The vertical axis shows the initial distance between the root and the subsystem analysed, and the horizontal axis shows the final distance. From the diagonal it can be seen that most questions are answered without including additional states toward the root.

| | | **Final distance** | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Sum |
| **Initial distance** | 1 | 114 | | | | | | | | | | | | 114 |
| | 2 | 30 | 65 | | | | | | | | | | | 95 |
| | 3 | 20 | 5 | 83 | | | | | | | | | | 108 |
| | 4 | 25 | 5 | 10 | 70 | | | | | | | | | 110 |
| | 5 | 12 | 0 | 2 | 8 | 59 | | | | | | | | 81 |
| | 6 | 0 | 0 | 6 | 8 | 10 | 77 | | | | | | | 101 |
| | 7 | 0 | 0 | 6 | 7 | 12 | 16 | 70 | | | | | | 111 |
| | 8 | 8 | 0 | 0 | 7 | 1 | 1 | 12 | 66 | | | | | 95 |
| | 9 | 0 | 0 | 0 | 0 | 0 | 11 | 10 | 3 | 89 | | | | 113 |
| | 10 | 0 | 0 | 0 | 10 | 0 | 1 | 5 | 7 | 5 | 75 | | | 103 |
| | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 8 | 9 | 91 | | 111 |
| | 12 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 2 | 2 | 9 | 14 | 90 | 123 |
| | Sum | 209 | 75 | 107 | 116 | 82 | 106 | 99 | 79 | 104 | 93 | 105 | 90 | 1265 |

## 7 Conclusion

In this paper we have presented a verification technique for hierarchical systems. The technique combines a new idea of reusability of reachability checks with a previously demonstrated successful compositional verification technique. The experimental results are encouraging: in contrast to a straightforward flattening approach the new technique proves not only insensitive to the hierarchical depth, but even leads to improved performance as the depth increases (given a fixed number of serial states). A topic for further research is how to extend the techniques to model-checking of more general temporal properties and how to combine it with utilisation of multiple instantiations of abstract superstates.

## Acknowledgment

The authors would like to thank Steffen Braa Andersen, Claus Krogholm Pedersen and Peter Smed Vestergaard for their valuable contributions to the work of this paper.

## References

1. Baan VisualState A/S. http://www.visualstate.com.
2. I-Logix Inc. http://www.ilogix.com.
3. ObjecTime Limited. http://www.objectime.on.ca.
4. Rational Software Corporation. http://www.rational.com.
5. Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. *Proceedings of the 6th ACM Symposium on Foundations*, 1998.
6. Steffen Braa Andersen, Gerd Behrmann, Claus Krogholm Pedersen, and Peter Smed Vestergaard. Reuseability and Compositionality applied to Verification of Hierarchical Systems. Master's thesis, Aalborg University, June 1998.
7. Gerd Behrmann, Kim G. Larsen, Henrik R. Andersen, Henrik Hulgaard, and Jørn Lind-Nielsen. Verification of Hierarchical State/Event Systems. To appear as a BRICS report (http://www.brics.dk), 1999.
8. G. Booch, I. Jacobsen, and J. Rumbaugh. *Unified Modelling Language User Guide*. Addison Wesley, 1997.
9. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.
10. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
11. F. Jahanian and A.K. Mok. A graphtheoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.
12. N.G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requiremets specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9):694–707, September 1994.
13. Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre Kristoffersen, and Kim G. Larsen. Verification of Large State/Event Systems using Compositionality and Dependency Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 201–216. Springer, March/April 1998.
14. David Y.W. Park, Jens U. Skakkebæk, and David L. Dill. Static Analysis to Identify Invariants in RSML Specifications. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 133–142. Springer, September 1998.
15. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991.
16. B. Selic, G. Gullekson, and P. T. Ward. *Real-time object oriented modeling and design*. J. Wiley, 1994.


10.1007/b107031130012