Stepwise CTL Model Checking of State/Event Systems

Jørn Lind-Nielsen and Henrik Reif Andersen

Department of Information Technology, Technical University of Denmark
e-mail: {jl,hra}@it.dtu.dk

Abstract. In this paper we present an efficient technique for symbolic model checking of any CTL formula with respect to a state/event system. Such a system is a concurrent version of a Mealy machine and is used to describe embedded reactive systems. The technique uses compositionality to find increasingly better upper and lower bounds of the solution to a CTL formula until an exact answer is found. Experiments show this approach to succeed on examples larger than the standard backwards traversal can handle, and even in many cases where both methods succeed it is shown to be faster.

1 Introduction

The range of systems that can be formally verified has improved drastically since the introduction of symbolic model checking [7,8] with the use of reduced and ordered binary decision diagrams (ROBDD) in the eighties [3,2]. Since then many people have improved on the basic algorithms by introducing more efficient techniques, more compact representations and new methods for simplifying the models.

One way to do simplifications on the model is by abstraction, where subcomponents of the system considered are removed to yield a smaller and simpler model on which the verification can be done. The technique described here is based on one such incremental abstraction of the system, where first an initially small subset of the system is used as an abstraction. If this set is not enough to prove or disprove the requirements then the set is incrementally extended until a point where it is possible to give an exact positive or negative answer.

The experimental results are promising and show that the iterative approach can be used with success on larger systems than the usual backwards traversal can handle, and it is still faster even when the usual method is feasible.

This work is a direct extension of the work presented at TACAS'98 [11]. Now the technique covers full CTL model checking and calculates simultaneously both an upper and a lower approximation to the solution of the CTL formula.

We apply this technique to the *state/event model* used in the commercial tool visualSTATE™ [13]. This model is a concurrent version of Mealy machines, that is, it consists of a fixed number of concurrent finite state machines that have pairs of input events and output actions associated with the transitions of

N. Halbwachs and D. Peled (Eds.): CAV'99, LNCS 1633, pp. 316–327, 1999.

the machines. The model is synchronous: each input event is reacted upon by all machines in lock-step; the total output is the multi-set union of the output actions of the individual machines. Further synchronization between the machines is achieved by associating a guard with the transitions. Guards are Boolean combinations of conditions on the local states of the other machines.

Both the state space and the transition relation of the state/event system is represented using ROBDDs with a partitioned transition relation, exactly as described in [11].

1.1 Related Work

Another similar technique for exploiting the structure of the system to be verified is described in [9]. This technique also computes increasingly better approximations to the exact solution of a CTL formula, but it differs from our approach in several ways. Instead of reusing the previously found states as shown in section 6, this technique has to recalculate the approximation from scratch each time a new component is added to the system. It may also have to include all components in the system, whereas we restrict our inclusion to only the dependency closure (cone of influence) of the formula used. Finally the technique is restricted to ACTL(ECTL) formulae, whereas our technique can be used with any CTL formula.

Pardo and Hachtel describes an abstraction technique in [14] where the approximations are done using ROBDD reduction techniques. This technique is based on the μ-calculus (and so includes CTL). It utilizes the structure of a given formula to find appropriate abstractions, whereas our technique depends on the structure of the model.

The technique for showing language containment of L-processes described in [1], also maintains a subset of processes used in the verification and analyzes error traces from the verifier to find new processes in order to extend this set. Although the overall goal of finding the result through an abstract, simplified model is the same as our, the properties verified are different and the L-processes have properties quite different from ours.

Abstraction as in [5] is similar to ours when the full dependency closure is included from the beginning, and thus it discards the idea of an iterative approach.

The idea of abstraction and compositionality is explored in more detail in David Longs' thesis [12].

2 State/Event Systems

A state/event system $\mathcal{S}$ consists of n machines $M_1, \ldots, M_n$, an alphabet E of input events and an alphabet O of outputs. Each machine M_i is a triple (S_i, s_i^0, T_i) of local states S_i, an initial state $s_i^0 \in S_i$, and a set of transitions T_i. The set of transitions is a relation

$$T_i \subseteq S_i \times E \times G_i \times \mathcal{M}(O) \times S_i,$$

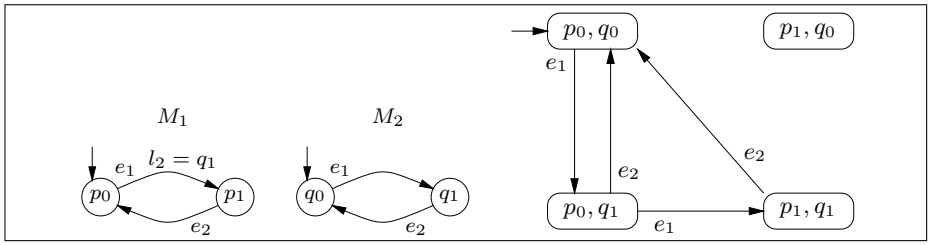

Fig. 1. Two state/event machines and the corresponding parallel combination. The small arrows indicate the initial states.

where $\mathcal{M}(O)$ is a multi-set of outputs, and G_i is the set of guards not containing references to machine i. These guards are generated from the following simple grammar for Boolean expressions:

$$g \quad ::= \quad l_j = p \mid \neg g \mid g \wedge g \mid tt\,.$$

The atomic predicate $l_j = p$ is read as "machine j is at local state p" and tt denotes a true guard. The global state set S of the state/event system is the product of the local state sets: $S = S_1 \times S_2 \times \cdots \times S_n$. The guards are interpreted straightforwardly over S as given by a satisfaction relation $s \models g$. The expression $l_j = p$ holds for any $s \in S$ exactly when the j'th component of s is p, i.e., $s_j = p$. The interpretation of the other cases is as usual. The transition relation is total, by assuming that the system stays in its current state when no transitions are enabled for a given input event.

Considering a global state s, all guards in the transition relation can be evaluated. We define a version of the transition relation in which the guards have been evaluated. This relation is denoted $s \xrightarrow{e\ o}_i s'_i$ expressing that machine i when receiving input event e makes a transition from s_i to s'_i and generates output o (here s_i is the i'th component of s). Formally,

$$s \xrightarrow{e\ o}_i s'_i \ \Leftrightarrow_{def}\ \exists g.\ (s_i, e, g, o, s'_i) \in T_i \text{ and } s \models g\,.$$

The transition relation of the whole system is defined as:

$$s \xrightarrow{e\ o} s' \Leftrightarrow_{def} \forall i.\ s \xrightarrow{e\ o_i}_i s'_i \quad \text{where} \quad s' = (s'_1, \ldots, s'_n) \text{ and } o = o_1 \uplus \ldots \uplus o_n$$

Where $\uplus$ denotes multi set union. The example in figure 1 shows a system with two state/event machines and the corresponding parallel combination. Machine M_2 starts in state q_0 and goes to state q_1 on the receipt of event e_1. Machine M_1 can not move on e_1 because of the guard. After this M_2 may go back on event e_2 or M_1 may enter state p_1 on event e_1. At last both M_1 and M_2 can return to their initial states on event e_2.

3 CTL Specifications

CTL [6] is a temporal logic used to specify formal requirements to finite state machines like the state/event systems presented here. Such specifications consist

of the Boolean constant true tt, conjunction $\wedge$, negation $\neg$, state predicates and temporal operators. We use a state predicate variant where the location of a machine is stated: $l_i = s$ meaning that machine i should be in state s, similar to the guards.

The temporal operators are the *next* operator $\mathbf{X}(\phi)$, the *future* operator $\mathbf{F}(\phi)$, the *globally* operator $\mathbf{G}(\phi)$ and the *until* operator $(\phi_1 \mathbf{U}\, \phi_2)$. Each of these operators must be directly preceded with a path quantifier to state whether the formula should hold for all possible execution paths of the system ($\mathbf{A}$) or only for at least one execution path ($\mathbf{E}$).

The solution to a CTL formula ϕ is the set of states $[\![\phi]\!]$ that satisfy the formula ϕ. A state/event system $\mathcal{S}$ is said to satisfy the property ϕ, $\mathcal{S} \models \phi$ if the initial state s^0 is in the solution to ϕ, $s^0 \in [\![\phi]\!]$.

To describe the exact semantics of the operators we use an additional function $[\![\mathbf{EX}]\!]$ that operates on a set of states P. This function returns all the states that may reach at least one of the states in P in exactly one step and is defined as $[\![\mathbf{EX}]\!]\ P = \{s \in S \mid \exists e, o, s'.\ s \xrightarrow{e\ o} s' \wedge s' \in P\}$. The operators are then defined, for a State/Event system with the transition relation $s \xrightarrow{e\ o} s'$, as:

$$
\begin{array}{ll}
[\![tt]\!] & = S \\
[\![\neg\phi]\!] & = S \backslash [\![\phi]\!] \\
[\![\mathbf{EX}\ \phi]\!] & = [\![\mathbf{EX}]\!][\![\phi]\!]
\end{array}
\qquad
\begin{array}{ll}
[\![l_i = s]\!] & = \{s' \in S \mid s'_i = s\} \\
[\![\phi_1 \wedge \phi_2]\!] & = [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![\mathbf{EG}\ \phi]\!] & = \nu U \subseteq S.\ [\![\phi]\!] \cap [\![\mathbf{EX}]\!] U
\end{array}
$$

$$
[\![\mathbf{E}(\phi_1\ \mathbf{U}\ \phi_2)]\!] = \mu U \subseteq S.\ [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap [\![\mathbf{EX}]\!] U)
$$

Here we use $\nu x.f(x)$ and $\mu x.f(x)$ as the maximal and minimal fixed points of a monotone function f on a complete lattice, as given by Tarski's fixed point theorem [16]. The rest of the operators can be defined using the above operators [8].

4 Bounded CTL Solutions

In this section we introduce the *bounded* CTL solution. A bounded CTL solution consists of two sets of states, namely $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$ which are lower- and upper approximations to the solution of the formula. The idea is to test for inclusion of the initial state in $\mathcal{L}[\![\phi]\!]_I$ and exclusion from $\mathcal{U}[\![\phi]\!]_I$. In the first case we know that the formula holds and in the second that it does not.

To describe bounded CTL solutions we need to formalize the concept of *dependency* between machines in a state/event system. We choose the notion that one machine M_i depends on another machine M_j if there exists at least one guard in M_i that has a syntactic reference to a state in M_j. These dependencies form a directed graph, which we call the *dependency graph*. In this graph each vertex represent a machine and an edge from a vertex i to a vertex j represents a dependency in machine M_i on a state in machine M_j. Note that we can ignore any dependencies introduced by the global synchronization of the input events.

A formula ϕ depends on a machine M_i if ϕ contains a sub-formula of the form $l_i = s$, and the *sort* of ϕ is all the machines ϕ depends on. The *dependency closure* of a machine M_i is all the machines that are reachable from M_i in the

dependency graph, including M_i. This is also sometimes refered to as the *cone of influence*. The dependency closure of a formula ϕ is the union of all the dependency closures of the machines in the sort of ϕ.

Assume (for the time being) that we have an efficient way to calculate a bounded solution to a CTL formula ϕ using only the machines in an index set I. The result should be two sets of states $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$ with the following properties:

$$\mathcal{L}[\![\phi]\!]_I \subseteq [\![\phi]\!] \subseteq \mathcal{U}[\![\phi]\!]_I. \quad (1)$$
$$\mathcal{L}[\![\phi]\!]_{I_1} \subseteq \mathcal{L}[\![\phi]\!]_{I_2} \text{ if } I_1 \subseteq I_2. \quad (2)$$
$$\mathcal{U}[\![\phi]\!]_{I_1} \supseteq \mathcal{U}[\![\phi]\!]_{I_2} \text{ if } I_1 \subseteq I_2. \quad (3)$$
$$\mathcal{L}[\![\phi]\!]_I = [\![\phi]\!] = \mathcal{U}[\![\phi]\!]_I \text{ if } I \text{ is dependency closed.} \quad (4)$$

Both $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$ are only defined for sets I that include the sort of ϕ. Property (1) means that $\mathcal{L}[\![\phi]\!]_I$ is a lower approximation of $[\![\phi]\!]$ and $\mathcal{U}[\![\phi]\!]_I$ is an upper approximation of $[\![\phi]\!]$. Property (2) and (3) mean that the approximations converge monotonically towards the correct solution of ϕ and property (4) states that we get the correct solution to ϕ when I contains all the machines found in the dependency closure of ϕ.

In section 5 we will show an algorithm that efficiently computes the bounded solution to any CTL formulae. With this, it is possible to make a serious improvement to the usual algorithm for finding CTL solutions. The algorithm utilizes the fact that we may be able to prove or disprove the property ϕ using only a (hopefully) small subset of all the machines in the system.

Our algorithm for verifying a CTL property is as follows

Algorithm *CTL verifier*
Input: A CTL formula ϕ and a state/event system $\mathcal{S}$ and it's dependency graph G
Output: **true** if $s^0 \in [\![\phi]\!]$ and **false** otherwise
1. I = sort(ϕ); result = **unknown**
2. **repeat**
3. calculate $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$
4. **if** $s^0 \in \mathcal{L}[\![\phi]\!]_I$ **then** result = **true**
5. **if** $s^0 \notin \mathcal{U}[\![\phi]\!]_I$ **then** result = **false**
6. $I = I \cup extend(I, G)$
7. **until** result $\neq$ **unknown**

First we set I to be the sort of ϕ and use this to calculate $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$. If $s^0 \in \mathcal{L}[\![\phi]\!]_I$, then we know, from property (2), that ϕ holds for the system, and if $s^0 \notin \mathcal{U}[\![\phi]\!]_I$ then we know, from property (3), that ϕ does not hold. If neither is the case then we add more machines to I and try again. This continues until ϕ is either proved or disproved. The algorithm is guaranteed to stop with either a false or a true result when I is the dependency closure of ϕ, in which case we have $\mathcal{L}[\![\phi]\!]_I = \mathcal{U}[\![\phi]\!]_I$, from property (4), and thus either $s^0 \in \mathcal{L}[\![\phi]\!]_I$ or $s^0 \notin \mathcal{U}[\![\phi]\!]_I$.

The function *extend* selects a new set of machines to be included in I. We have chosen to include new machines in a breadth-first manner, so that *extend* returns all machines reachable in G from I in one step.

5 Bounded CTL Calculation

In section 4 we showed how to verify a CTL formula ϕ, by using only a minimal set of machines I from a state/event system and using an efficient algorithm for the calculation of lower and upper approximations of $[\![\phi]\!]$. In this section we will show one such algorithm, and for this we need some more definitions. Relating to an index set I of machines, two states $s, s' \in S$ are *I-equivalent*, written as $s =_I s'$, if for all $i \in I$, $s_i = s'_i$. A set of states P is *I-sorted* if the following predicate is true

$$I\text{-sorted}(P) \Leftrightarrow_{def} \forall s, s' \in S.\ (s \in P \wedge s =_I s') \Rightarrow s' \in P.$$

This means that if a state s is in P then all other states s', which are I-equivalent to s, must also be in the set P. This is equivalent to say that a set P is I-sorted if it is independent of the machines in the complement $\overline{I} = \{1, \ldots, n\} \setminus I$.

Consider as an example two machines with the sets of states $S_0 = \{p_0, p_1\}$, $S_1 = \{q_0, q_1, q_2\}$ and a sort $I = \{0\}$. Now the two pairs of states (p_0, q_1) and (p_0, q_2) are I-equivalent because their first states match. The set $P = \{(p_0, q_0), (p_0, q_1), (p_0, q_2)\}$ is I-sorted because it is independent of the states in S_1.

The bounded calculation of the constants tt and $l_i = s$, negation and conjunction is straight forward as shown in figure 2. The results are clearly I-sorted and satisfies the properties in section 4, if the sub-expressions ϕ, ϕ_1 and ϕ_2 does so.

Next State Operator: To show how to find $\mathcal{L}[\![\mathbf{EX}\ \phi]\!]_I$ and $\mathcal{U}[\![\mathbf{EX}\ \phi]\!]_I$ we introduce two new operators: $[\![\mathbf{E}_\forall\mathbf{X}]\!]_I$ and $[\![\mathbf{E}_\exists\mathbf{X}]\!]_I$. The lower approximation $[\![\mathbf{E}_\forall\mathbf{X}]\!]_I\ P$ is a conservative approximation to $[\![\mathbf{EX}]\!]$ that only includes states that are guaranteed to reach P in one step, regardless of the states of the machines in $\overline{I}$. The upper approximation $[\![\mathbf{E}_\exists\mathbf{X}]\!]_I\ P$ is an optimistic approximation that includes all states that just might reach P. These two operators are defined as:

$$[\![\mathbf{E}_\forall\mathbf{X}]\!]_I\ P = \{s \in S \mid \forall s' \in S.\ s =_I s' \Rightarrow\ s' \in [\![\mathbf{EX}]\!]_I\ P\}$$
$$[\![\mathbf{E}_\exists\mathbf{X}]\!]_I\ P = \{s \in S \mid \exists s' \in S.\ s =_I s' \wedge\ s' \in [\![\mathbf{EX}]\!]_I\ P\}$$

where the results of both operators are I-sorted when $[\![\mathbf{EX}]\!]_I\ P$ is I-sorted, as a result of the extra quantifiers. The calculation of $[\![\mathbf{EX}]\!]_I\ P$ can be done efficiently *when P is I-sorted*, using a partitioned transition relation [4]. The definition of $[\![\mathbf{EX}]\!]_I$ is

$$[\![\mathbf{EX}]\!]_I\ P = \{s \in S \mid \exists e, o, s'.\ s \xrightarrow{e\ o} s' \wedge s' \in P\}.$$

This seems to depend on all n machines in $\mathcal{S}$, but as a result of P being I-sorted, it can be reduced to

$$[\![\mathbf{EX}]\!]_I\ P = \{s \in S \mid \exists e, o.\ \exists s'_{\overline{I}}.\ \exists s'_I.\ s \xrightarrow{e\ o}_I s'_I \wedge (s'_1, \ldots, s'_N) \in P\}$$

where $\exists s'_I.\ s \xrightarrow{e\ o}_I s'_I$ means $\bigwedge_{i \in I} \exists s'_i.\ s \xrightarrow{e\ o}_i s'_i$. This clearly depends on only the transition relations for the machines in I.

Now we can define $\mathcal{L}[\![\mathbf{EX}\ \phi]\!]_I$ and $\mathcal{U}[\![\mathbf{EX}\ \phi]\!]_I$ as

$$\begin{aligned}\mathcal{L}[\![\mathbf{EX}\ \phi]\!]_I &= [\![\mathbf{E}_\forall \mathbf{X}]\!]_I\ \mathcal{L}[\![\phi]\!]_I \\ \mathcal{U}[\![\mathbf{EX}\ \phi]\!]_I &= [\![\mathbf{E}_\exists \mathbf{X}]\!]_I\ \mathcal{U}[\![\phi]\!]_I.\end{aligned}$$

Both $\mathcal{L}[\![\mathbf{EX}\ \phi]\!]_I$ and $\mathcal{U}[\![\mathbf{EX}\ \phi]\!]_I$ are clearly I-sorted because both $[\![\mathbf{E}_\forall \mathbf{X}]\!]_I$ and $[\![\mathbf{E}_\exists \mathbf{X}]\!]_I$ are so, and if ϕ satisfies the properties in section 4 then so does $\mathcal{L}[\![\mathbf{EX}\ \phi]\!]_I$ and $\mathcal{U}[\![\mathbf{EX}\ \phi]\!]_I$.

Globally and Until operators: The semantics for $\mathbf{EG}\ \phi$ is defined in the same manner as $\mathbf{EX}\ \phi$, with an added fixed point calculation:

$$\begin{aligned}\mathcal{L}[\![\mathbf{EG}\ \phi]\!]_I &= \nu U.\ \mathcal{L}[\![\phi]\!]_I \cap [\![\mathbf{E}_\forall \mathbf{X}]\!]_I U \\ \mathcal{U}[\![\mathbf{EG}\ \phi]\!]_I &= \nu U.\ \mathcal{U}[\![\phi]\!]_I \cap [\![\mathbf{E}_\exists \mathbf{X}]\!]_I U.\end{aligned}$$

The result is also I-sorted and satisfies the properties in section 4 if ϕ does. For $\mathbf{E}(\phi_1\ \mathbf{U}\ \phi_2)$ we take

$$\begin{aligned}\mathcal{L}[\![\mathbf{E}(\phi_1\ \mathbf{U}\ \phi_2)]\!]_I &= \mu U.\ \mathcal{L}[\![\phi_2]\!]_I \cup (\mathcal{L}[\![\phi_1]\!]_I \cap [\![\mathbf{E}_\forall \mathbf{X}]\!]_I\ U) \\ \mathcal{U}[\![\mathbf{E}(\phi_1\ \mathbf{U}\ \phi_2)]\!]_I &= \mu U.\ \mathcal{U}[\![\phi_2]\!]_I \cup (\mathcal{U}[\![\phi_1]\!]_I \cap [\![\mathbf{E}_\exists \mathbf{X}]\!]_I\ U).\end{aligned}$$

6 Reusing Bounded CTL Calculations

One problem with the operators $\mathcal{L}[\![\phi]\!]_I$ and $\mathcal{U}[\![\phi]\!]_I$ from section 5, when used in the bounded CTL verifier, is that all previously found states have to be rediscovered whenever a new set of machines I is introduced. In this section we will show how to avoid this problem for the **EG** operator and sketch how to do it for the $\mathbf{E}(\phi_1\mathbf{U}\ \phi_2)$ operator. The final algorithm is shown in figure 2.

First we show how the calculation of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2}$ can be improved by reusing the previous calculation of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_1}$ when $I_1 \subseteq I_2$. From the definition of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_I$ we get the following:

$$\begin{aligned}&\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2} \subseteq \mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_1} \subseteq \mathcal{U}[\![\phi]\!]_{I_1} \textit{ and} \\ &\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2} \subseteq \mathcal{U}[\![\phi]\!]_{I_2} \qquad \subseteq \mathcal{U}[\![\phi]\!]_{I_1}\end{aligned}$$

and from this we know that

$$\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2} \subseteq \mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_1} \cap \mathcal{U}[\![\phi]\!]_{I_2}.$$

We also know, from Tarski's fixed point theorem, that $\nu f \subseteq x \Rightarrow \nu f = \bigcap_i f^i(x)$, which means the maximum fixed point calculation of f can be started from any

$$
\begin{array}{lll}
\mathcal{B}[\![tt]\!]_{I_k} & = & (S\ ,\ S) \\
\mathcal{B}[\![l_i = s]\!]_{I_k} & = \text{let } L = \{s' \in S \mid s'_i = s\} & \\
 & \quad\ U = \{s' \in S \mid s'_i = s\} & \\
 & \text{in } (L, U) & \\
\mathcal{B}[\![\neg\phi]\!]_{I_k} & = \text{let } (L, U) = \mathcal{B}[\![\phi]\!]_{I_k} & \\
 & \text{in } (S \setminus U\ ,\ S \setminus L) & \\
\mathcal{B}[\![\phi_1 \wedge \phi_2]\!]_{I_k} & = \text{let } (L_1, U_1) = \mathcal{B}[\![\phi_1]\!]_{I_k} & \\
 & \quad\ (L_2, U_2) = \mathcal{B}[\![\phi_2]\!]_{I_k} & \\
 & \text{in } (L_1 \cap L_2\ ,\ U_1 \cap U_2) & \\
\mathcal{B}[\![\mathbf{EX}\ \phi]\!]_{I_k} & = \text{let } (L, U) = \mathcal{B}[\![\phi]\!]_{I_k} & \\
 & \text{in } ([\![\mathbf{E}_\forall\mathbf{X}]\!]_{I_k}\ L\ ,\ [\![\mathbf{E}_\exists\mathbf{X}]\!]_{I_k}\ U) & \\
\mathcal{B}[\![\mathbf{EG}\ \phi]\!]_{I_k} & = \text{let } (L_1, U_1) = \mathcal{B}[\![\phi]\!]_{I_k} & \\
 & \quad\ (\_\_, U_2) = \mathcal{B}[\![\mathbf{EG}\ \phi]\!]_{I_{k-1}} & \\
 & \quad\ U = \nu V.\ (U_1 \cap U_2) \cap [\![\mathbf{E}_\exists\mathbf{X}]\!]_{I_k} V & \text{(a)} \\
 & \quad\ L = \nu V.\ (L_1 \cap U) \cap [\![\mathbf{E}_\forall\mathbf{X}]\!]_{I_k} V & \text{(b)} \\
 & \text{in } (L\ ,\ U) & \\
\mathcal{B}[\![\mathbf{E}\ (\phi_1\mathbf{U}\ \phi_2)]\!]_{I_k} & = \text{let } (L_1, U_1) = \mathcal{B}[\![\phi_1]\!]_{I_k} & \\
 & \quad\ (L_2, U_2) = \mathcal{B}[\![\phi_2]\!]_{I_k} & \\
 & \quad\ (L_3, \_\_) = \mathcal{B}[\![\mathbf{E}\ (\phi_1\mathbf{U}\ \phi_2)]\!]_{I_{k-1}} & \\
 & \quad\ L = \mu V.\ (L_2 \cup L_3) \cup (L_1 \cap [\![\mathbf{E}_\exists\mathbf{X}]\!]_{I_k} V) & \text{(c)} \\
 & \quad\ U = \mu V.\ (U_2 \cup L) \cup (U_1 \cap [\![\mathbf{E}_\exists\mathbf{X}]\!]_{I_k} V) & \text{(d)} \\
 & \text{in } (L\ ,\ U) &
\end{array}
$$

Fig. 2. Full description of how the lower and upper approximations $(\mathcal{L}[\![\phi]\!]_I, \mathcal{U}[\![\phi]\!]_I) = \mathcal{B}[\![\phi]\!]_I$ are calculated for a state/event system $\mathcal{S}$. The sorts are I_k for the current sort and I_{k-1} for the previous sort. Initially we have $\mathcal{B}[\![\phi]\!]_{I_0} = (\emptyset, S)$ and I_0 is the sort of the expression. We use L for a lower approximation and U for an upper approximation. The lines (a)–(d) show where we reuse previously found states.

x as long as x includes the maximal fixed point of f. Here we use $f^i(x)$ as the i'th application of f on itself. From this it is clear that the fixed point calculation of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2}$ can be started from the intersection of the two sets $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_1}$ and $\mathcal{U}[\![\phi]\!]_{I_2}$. Normally this fixed point calculation would have been started from $\mathcal{U}[\![\phi]\!]_{I_2}$, but in this way we reuse the calculation of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_1}$ to speed up the calculation of $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2}$.

The same idea can be used for the lower approximation $\mathcal{L}[\![\mathbf{EG}\ \phi]\!]_{I_2}$, where the fixed point iteration can be started from the intersection of $\mathcal{L}[\![\phi]\!]_{I_2}$ and $\mathcal{U}[\![\mathbf{EG}\ \phi]\!]_{I_2}$, so that we reuse the calculation of the upper approximation. The algorithm in figure 2 utilizes this in line (a) for the upper approximation and in line (b) for the lower approximation.

Exactly the same can be done for $\mathcal{L}[\![\mathbf{E}(\phi_1\mathbf{U}\ \phi_2)]\!]_I$ and $\mathcal{U}[\![\mathbf{E}(\phi_1\mathbf{U}\ \phi_2)]\!]_I$, except that the previous *lower* approximations should be used to restart the calculation, as shown in line c and d in figure 2.

System	Machines	Local states	Declared	Reachable
INTERVM	6	182	10^6	15144
VCR	7	46	10^5	1279
DKVM	9	55	10^6	377568
HI-FI	9	59	10^7	1416384
FLOW	10	232	10^5	17040
MOTOR	12	41	10^6	34560
AVS	12	66	10^7	1438416
VIDEO	13	74	10^8	1219440
ATC	14	194	10^{10}	6399552
OIL	24	96	10^{13}	237230192
TRAIN1	373	931	10^{136}	–
TRAIN2	1421	3204	10^{476}	–

Table 1. The state/event systems used in the experiments. The last two columns show the size of the declared and reachable state space. The size of the declared state space is the product of the number of local states of each machine. The reachable state space is only known for those systems where a forward iteration of the state space can complete.

7 Examples

The technique presented here has been tested on ten industrial state/event systems and two systems constructed by students in a course on embedded systems. The examples are all constructed using visualSTATE™ [13] and cover a large range of different applications. The examples are HI-FI, AVS, ATC, FLOW, MOTOR, INTERVM, DKVM, OIL, TRAIN1 and TRAIN2 which are industrial examples and VCR and VIDEO which are constructed by students. In table 1 we have listed some characteristics for these examples.

The experiments were carried out on a pentium 166MHz PC with 32Mb of memory, running Linux. For the ROBDD encoding we used BuDDy [10], a locally produced ROBDD package which is comparable in efficiency to other state of the art ROBDD packages, such as CUDD [15].

For each example we tested three different sets of CTL formulae. One set of formulae for detecting non-determinism in the system, one set for detecting local deadlocks and one for finding homestates.

Non-determinism occurs when two transitions leading out of a state depends on the same event and has guards that are enabled at the same time in a reachable global state. That is, the intersection of the two guards $g = g_1 \wedge g_2$ should be non-empty and reachable. Every combination of guards were then checked for reachability using the formula **EF** g.

Locally deadlocked states are local states from which there can never be enabled any outgoing transition, no matter how the rest of the system behaves. So for each local state s in the system we check for absence of local deadlocks using the formula **AG** $(s \Rightarrow$ **EF** $\neg s)$

Homestates are states that can be reached from any point in the reachable state space. So for each local state s of the system we get the formula **AG** (**EF** s).

We have unfortunately only access to the examples in an anonymous form, so we have no way of generating more specialized properties.

In table 2 we have listed the time it takes to complete checking a whole set of CTL formulae using the standard backwards traversal with either *all* machines in the system or only the machines in the dependency closure, and the time used with stepwise traversal. For the largest system it is only the stepwise traversal that succeeds and with the exception of one system (ATC) the stepwise traversal is also faster or comparable in speed to the standard backwards traversal.

We have also shown the number of tests that can be verified using fewer machines than in the dependency closure, how much of the dependency closure there was needed to do it, how many tests that had to include the full dependency closure and the average size of that dependency closure. From this we can see that in the TRAIN2 example we can verify most of the formulae using only a small fraction ($3 - 15\%$) of the dependency closure and when the full dependency closure has to be included then the average size of it is only as little as 1 (although we know that some tests includes more than 200 machines in the dependency closure). This indicates that TRAIN2 is a *loosely coupled* system i.e. a system with few dependencies among the state machines.

We also see that ATC and OIL are more strongly coupled, as the average dependency closure is larger than for the other examples. This property is also mirrored in the time needed to verify the two examples.

8 Conclusion

We have extended the successful model checking technique presented in [11] with the ability to do full CTL model checking and not only reachability and deadlock detection. We have also added the calculation of both upper *and* lower approximations to the result and in this way making it possible to stop earlier in the verification process with either a negative or a positive answer.

Test examples have shown the stepwise traversal of the state space to be more efficient, than the normal backwards traversal, in terms of both time and space for a range of industrial examples. We have also shown that the stepwise technique may succeed in cases where the standard techniques fails.

The examples also indicates that the stepwise traversal works best on loosely coupled systems, that is; systems with few dependencies among the involved state machines.

Acknowledgement

Thanks to the colleagues of the VVS project for numerous fruitful discussions on the verification of state/event models.

Test Data			Run times				Dependencies							
Example	Test	Num	Full	DC.	Step	Red.	Part. DC.				Full DC.			
							Ok		Err		Ok		Err	Size
INTERVM	D	182	6.7	**6.2**	6.8		150	41%	0	0%	32		0	4.5
(6)	H	182	50.3	47.3	**40.6**	+19%	96	57%	0	0%	86		0	4.8
VCR	C	1	0.3	**0.2**	**0.2**		0	0%	0	0%	1		0	6.0
(7)	D	46	0.6	**0.4**	0.8		2	40%	0	0%	44		0	5.4
	H	46	2.2	**1.3**	1.5		2	40%	0	0%	44		0	5.4
DKVM	D	55	0.5	**0.4**	**0.4**		46	20%	0	0%	9		0	1.0
(9)	H	55	8.7	8.7	**6.7**		27	45%	1	11%	27		0	1.7
HI-FI	D	59	0.8	0.7	**0.5**		56	18%	0	0%	3		0	3.0
(9)	H	59	3.5	3.1	**2.3**		52	56%	0	0%	7		0	5.3
FLOW	C	2	0.6	0.6	0.6		2	75%	0	0%	0		0	-
(10)	D	232	1.4	**1.1**	**1.1**		224	49%	0	0%	8		0	1.0
	H	232	3.6	2.4	**2.1**		223	49%	0	0%	9		0	1.2
MOTOR	D	41	0.9	0.9	**0.6**		32	21%	0	0%	9		0	1.0
(12)	H	41	1.2	1.2	**0.7**		32	24%	0	0%	9		0	1.0
AVS	C	5	1.2	1.2	**1.1**		4	27%	0	0%	1		0	3.0
(12)	D	66	2.0	1.7	**1.5**		57	34%	0	0%	8		1	1.3
	H	66	6.0	5.2	**4.0**		42	64%	3	76%	20		1	3.2
VIDEO	D	74	0.9	0.7	**0.6**		70	30%	0	0%	4		0	2.0
(13)	H	74	2.6	**1.3**	**1.3**		57	54%	0	0%	17		0	4.3
ATC	C	122	153.5	140.4	**138.6**	+10%	11	92%	0	0%	111		0	12.0
(14)	D	194	119.0	**110.2**	135.3	-14%	3	8%	63	75%	6		122	11.6
	H	194	495.3	461.2	**443.7**	+10%	3	8%	63	75%	6		122	11.6
OIL	C	114	242.8	177.4	**163.8**	+33%	2	25%	29	25%	83		0	12.0
(24)	D	96	15.0	9.6	**7.3**	+51%	58	19%	6	17%	26		6	6.6
	H	96	35.5	23.8	**15.5**	+56%	22	22%	33	9%	31		10	7.8
TRAIN1	C	99	76.1	3.7	**3.6**	+95%	82	57%	0	0%	17		0	4.5
(373)	D	931	449.7	**5.0**	5.3	+99%	388	41%	9	58%	502		32	1.0
	H	931	478.8	**4.9**	**4.9**	+99%	354	41%	42	50%	500		35	1.0
TRAIN2	C	1245	-	-	**265.4**	+100%	912	8%	30	6%	303		0	1.1
(1421)	D	3204	-	-	**199.0**	+100%	1569	3%	16	8%	1583		36	1.0
	H	3204	-	-	**197.1**	+100%	1521	3%	58	15%	1585		40	1.0

Table 2. Test examples for runtime and dependency analysis. All times are in seconds. The tests are C-Conflicts, D-Deadlock and H-Homestates. The Num column shows the number of tests, the Full column is the time used with all machines included from the beginning (and still using a partitioned transition relation), the DC. column is the time used with only the dependency closure included from the beginning and the Step column is the time used with stepwise traversal. A dash means timeout after one hour or spaceout around 20Mb, all other tests were done with less than 250k ROBDD nodes in memory at one time. The Red column is the reduction in runtime $= (Full - Step)/Full$. Some systems have no conflicts and we have left out the data for these. The Part.DC. column shows the number of tests that could be verified using fewer state machines than in the full dependency closure, whether the result was true (Ok) or false (Err) and how much of the dependency closure was included. The Full DC. column shows the number of tests that needed the full dependency closure to be proven true (Ok) or false (Err) and the average size of the dependency closure (Size).

References

1. F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In C. Courcoubetis, editor, *CAV'93. 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 29–40, Berlin, 1993. Springer-Verlag.
2. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
3. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proc. 1991 Int. Conf. on VLSI*, August 1991.
5. William Chan, Richard J. Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, volume 23,2 of *ACM Software Engineering Notes*, pages 102–112, New York, March2–5 1998. ACM Press.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
7. J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.
8. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.
9. W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 76–81, Washington, November10–14 1996. IEEE Computer Society Press.
10. Jørn Lind-Nielsen. *BuDDy - A Binary Decision Diagram Package*. Technical University of Denmark, 1997. `http://britta.it.dtu.dk/~jl/buddy`.
11. 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 *TACAS'98 Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, 1998.
12. David E. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon, 1993.
13. Beologic® A/S. *visualSTATE™ 3.0 User's Guide*, 1996.
14. Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *Computer Aided Verification, CAV'97*. Springer Verlag, 1997.
15. Fabio Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, 1997.
16. A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math.*, 5:285–309, 1955.