Space Effective Model Checking for Component-Interaction Automata

Nikola Beneš*, Milan Křivánek**, and Filip Štefaňák**

Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
{xbenes3,xkrivan8,xstefan5}@fi.muni.cz

Abstract. The techniques of component-based development are becoming a common practice in the area of software engineering. One of the crucial issues in the correctness of such systems is the correct interaction among the components. The formalism of component-interaction automata was devised to model various aspects of such interaction, as well as to allow automated verification in the form of model checking with properties expressed in the component-interaction LTL, a variant of the known linear temporal logic. As the state space of a component-based system can grow exponentially with the number of components, it is desirable to employ reduction techniques to make the verification task more feasible. In our work, we describe the implementation of the ample set partial order reduction method within the component-interaction automata verification framework. Due to the state and action-based nature of both the modelling and specification formalisms, the implementation differs from traditional state-based approaches. After describing the implementation, we present some of the results obtained by employing the enhanced verification framework on a case study.

1 Introduction

The demand to shorten the time necessary to develop complex software and to lower its costs encourages employment of new software development techniques. One of such techniques is the component-based development, which builds software systems out of prefabricated autonomous components that are often developed without any knowledge of their deployment context. Therefore a great deal of attention must be paid to their interaction, since correct interaction of the components plays an important role in the correctness of the system as a whole.

Component-interaction automata [1] represent a formalism designed for specification of component-based systems. Such models can be used to verify desirable properties of the system expressed as formulae of a suitable logic. CoIn verification environment [2] is based on DiVinE verification framework [3] and allows model checking of these specifications. The properties are formalized using a variant of state/event LTL [4] that is better suited for component-based

* The author has been supported by Grant Agency project no. GD102/09/H042.
** The authors have been supported by Academy of Sciences grant no. 1ET400300504.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2354

systems than pure state-based logics, as we are interested in both the state of the components and their communication.

The verification tool has to cope with exponential growth of the state space that is commonly caused by interleaving of independent actions. Ample set partial-order reduction [5] is one of the state space reduction techniques, which tries to identify redundant states and omit their generation while preserving verification properties of the model.

Traditionally, partial order reduction has been used in connection with state-based models. In [6], we have shown how partial order reduction can be performed on state/event-based models. The goal of this work is to implement this reduction method into the CoIn verification environment, which is an example of such formalism. In order to do that, we have to find effective heuristics to check the conditions for ample sets different from those used in traditional state-based approaches.

2 Foundations

Modelling and Specification Formalisms We start with describing the component-interaction automata formalism [1]. Each component is modelled as a finite labelled transition system equipped with an additional structure on labels and a hierarchy of names representing the architectural structure.

A *component-interaction automaton* (CI automaton for short) is a 5-tuple $\mathcal{C} = (Q, A, \delta, Q_0, H)$ where Q is a finite set of *states*, A is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times A \times (S_H \cup \{-\})) \setminus (\{-\} \times A \times \{-\})$ is a set of *labels*, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*, $Q_0 \subseteq Q$ is a nonempty set of *initial states*, and H is a hierarchy of component names where the set of component names is denoted by S_H.

The semantics of the labels is input, output, or internal, based on their structure. In the triple, the middle element represents an action name, the first element represents the name of the component that outputs the action, and the third element represents the name of the component that inputs the action.

The CI automata can be composed together using a parametrized composition operator $\otimes^{\mathcal{F}}$. Given a set of *feasible labels* $\mathcal{F}$ and a set of CI automata, the result of the operation is a product automaton with only labels from $\mathcal{F}$ allowed. In the product, the components cooperate either by interleaving of their original transitions, or by simultaneous execution of two complementary transitions (with labels $(n_1, a, -)$, $(-, a, n_2)$) which is represented by a new internal transition (with label (n_1, a, n_2)).

As for the property specification logic, we use a variant of the state/event LTL [4, 6], which is an extension of LTL for reasoning about both properties of states and actions. Currently, the only state atomic propositions we consider are the *enabledness properties*, $Ap = \{\mathcal{E}(l) \mid l \in \Sigma\}$. We say that a state satisfies the property $\mathcal{E}(l)$ if an outgoing transition with label l is enabled in that state. We define a function $\mathcal{L} : Q \to 2^{Ap}$ as $\mathcal{L}(q) = \{\mathcal{E}(l) \mid q \text{ satisfies } \mathcal{E}(l)\}$.

Partial Order Reduction Technique Our approach follows the ample set partial order reduction technique as presented in [7]. The basic idea is to view the verified system as a state transition system in which some of the transition are invisible, and to reduce the system such that all original behaviour is preserved with respect to the ordering of visible transitions.

A *state transition system* is a triple (S, T, S_0) where S is a set of states, S_0 is a nonempty set of initial states and T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$. Furthermore, for each $\alpha \in T$ and for each state $s \in S$ there is at most one $s' \in S$ such that $(s, s') \in \alpha$. We also write $\alpha(s) = s'$.

In the traditional state-based approach, the *invisible* transitions are those that do not change the state atomic propositions. In the state/event-based approach [6], each transition is further equipped with an action. The property to be verified is then supplied with a set of *interesting actions* Act', and the invisible transitions are those with non-interesting actions that do not change the state atomic propositions.

Two transitions α and β are said to be *independent* if whenever α and β are enabled in s, then also α is enabled in $\beta(s)$, and $\alpha(\beta(s)) = \beta(\alpha(s))$ for all s.

While exploring the state space of the system, the ample set method works by selecting only a subset of outgoing transitions from each state. The original set of outgoing transitions from state s is denoted by $enabled(s)$, the selected subset is denoted by $ample(s)$. To ensure that the reduction is correct, the following four conditions must hold.

C0 - nonemptiness $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.
C1 - dependency Along every path in the full state graph that starts at s, a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition from $ample(s)$ occurring first.
C2 - invisibility If $enabled(s) \neq ample(s)$ then every $\alpha \in ample(s)$ is invisible.
C3 - cycle A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

POR and CI automata In our setting, the system consists of a finite set of *simple* CI automata (numbered $1, \ldots, n$) whose state space is described explicitly, composed in a hierarchical way. The hierarchical composition is represented by a number of *composite* CI automata. The hierarchy can be thus represented with a tree, the leaves being the simple automata and the root being the CI automaton representing the whole system.

The (implicit) translation into a state transition systems then works as follows. The states are n-tuples $(s_1, \ldots, s_n)$ of the states of the simple automata. The transitions are then of two kinds: those that represent the progression of only one of the simple automata (*simple transitions*), and those that represent a synchronization of two simple automata (*sync transitions*). The simple transitions can be identified with tuples of the form $\langle i, s_i, s'_i, l \rangle$ and the sync transitions with tuples of the form $\langle i, j, s_i, s'_i, s_j, s'_j, l \rangle$, where i, j are automata numbers, s_i, s'_i, s_j, s'_j their respective states and l is the transition label.

3 Heuristics and Implementation

Overapproximations of the ample set conditions Some of the original ample set conditions are difficult to check, especially given that we want to check them in each state as we build the composite transition system. Therefore we use an overapproximation of these properties, using modified heuristics from [7]. We keep the condition for C3, but use modified versions for C1 and C2.

Firstly, we have to address the issue of selecting the candidates for ample sets, as the result helps us to create more elegant overapproximations. Usually, transitions of a single simple automaton are dependent on each other. Therefore we use the obvious solution of considering ample sets, which for each automaton consist of either all its transitions or none. This approach may not be feasible, though, because the number of subsets of all automata is exponential.

Exploiting the hierarchical structure To further reduce the number of possible candidates, we take advantage of the tree structure of the composition and for ample sets consider only the transitions of a single simple or composite automaton. The selected automaton is denoted by Aut and the set of all simple automata of which it consists by I. The candidate selection starts with the leaves (i.e. simple automata) and progresses towards the root of the hierarchy tree.

Dependency predicate The major problem with original heuristics for checking ample sets [7] is that the overapproximation of the dependency condition (C1) is too restrictive for a system with a lot of synchronization. In fact, only a very small category of systems of CI automata could ever have ample sets that are smaller than the full enabled sets, since it effectively says that an automaton in I that has an enabled action can only ever synchronize with an automaton in I, no matter whether the synchronization is enabled. We relax this condition by allowing synchronizations, that could not be performed by automata in I without them changing their state first.

Definition 1 (dependency condition C1′). *Let $current_i(s)$ be the set of all transitions that could be performed by simple automaton i from state s_i, where $s = (s_1, \ldots, s_n)$. We define C1′ as $\forall i \in I\, \forall \alpha \in current_i(s)\, (\text{automata of } \alpha \subseteq I)$.*

Lemma 1. *Let $ample(s)$ be the set of all enabled transitions that belong to simple automata in I. Then C1′ implies C1.*

Proof. Suppose the opposite: C1′ holds, but C1 does not. Then there is a path from s on which a transition β dependent on $\alpha \in ample(s)$ appears before all transitions from $ample(s)$. That β is dependent on α means that β and α share at least one automaton. As C1′ holds, the automaton or automata of α are in I. Therefore at least one automaton of β, say i, is in I. Clearly, β cannot be enabled in s, as then it would have to be in $ample(s)$.

Thus, if one automaton of β is not in I, the current state of i needs to change before β becomes enabled; otherwise β would violate C1′. If all automata of β are in I, at least one of them has to change its state. However, to change any

state of an automaton in I, a transition in $ample(s)$ has to be performed first, since all automata in I can currently synchronize only among themselves, which leads to a contradiction. □

Visibility predicate In order to check C2, we need a visibility predicate. As mentioned in the previous section, all transitions with an interesting action and all transitions that change the state atomic propositions have to be considered visible. As usual in the partial order reduction method, we are only interested in the change of those state atomic propositions that appear in the formula that is to be verified. However, as we deal with enabledness propositions, which are properties of the whole state space, we need to use an overapproximation.

Definition 2 (closure). *Let the set of all state labels in the formula be denoted by Ap'. Then an $\mathcal{E}$-closure c of Ap' is defined as:*

$$c(Ap') = Ap' \cup \{\mathcal{E}(m, a, -), \mathcal{E}(-, a, n) \mid \mathcal{E}(m, a, n) \in Ap'\}$$

The *visible* predicate is then defined as follows.

Definition 3 (*visible*). *Let the set of all state labels in the formula be denoted by Ap' and the set of interesting action labels by Act'. Then, if $\alpha = \langle i, s_i, s_i', l\rangle$ is a simple transition:*

$$visible(\alpha) \iff l \in Act' \vee (\mathcal{L}(s_i) \cap c(Ap') \neq \mathcal{L}(s_i') \cap c(Ap'))$$

and if $\alpha = \langle i, j, s_i, s_i', s_j, s_j', l\rangle$ is a sync transition:

$$visible(\alpha) \iff l \in Act' \vee \exists k \in \{i, j\} : (\mathcal{L}(s_k) \cap c(Ap') \neq \mathcal{L}(s_k') \cap c(Ap'))$$

Lemma 2 (*visible* is correct). *If a transition $(s, (m, a, n), s')$ in the state space is visible, the predicate $visible(\alpha)$ holds, where α is the corresponding transition of the state transition system.*

Proof. If a transition $(s, (m, a, n), s')$ is visible, then either $(m, a, n) \in Act'$ or $\mathcal{L}(s) \cap Ap' \neq \mathcal{L}(s') \cap Ap'$. In the first case, $visible(\alpha)$ clearly holds. For the second case, suppose that there is some $\mathcal{E}(k, b, l) \in \mathcal{L}(s) \cap Ap' \setminus \mathcal{L}(s') \cap Ap'$. That means that there is a transition $s \xrightarrow{(k,b,l)} t$, but there is no (k, b, l) transition enabled in s'. This transition can be either simple or sync. If it is simple, then there is some i such that $s_i \xrightarrow{(k,b,l)} t_i$ and clearly, $s_i' \neq s_i$, otherwise transition (k, b, l) would also be enabled in state s'. Thus $\mathcal{E}(k, b, l) \in \mathcal{L}(s_i) \cap c(Ap')$, but $\mathcal{E}(k, b, l) \notin \mathcal{L}(s_i') \cap c(Ap')$ and since $s_i' \neq s_i$, the transition α must be local for automaton i and $visible(\alpha)$ holds.

If the (k, b, l) transition is sync, then there have to be i, j such that $s_i \xrightarrow{(k,b,-)} t_i$ and $s_j \xrightarrow{(-,b,l)} t_j$. As the (k, b, l) transition is not enabled in s', that means that either α changes the state of i and s_i' has no $(k, b, -)$ transition, or α changes the state of j and s_j' has no $(-, b, l)$ transition. Suppose w.l.o.g. that it is the case with i. Clearly $\mathcal{E}(k, b, -) \in c(Ap')$ by the definition of the closure. But then $\mathcal{E}(k, b, -) \in \mathcal{L}(s_i) \cap c(Ap')$ and $\mathcal{E}(k, b, -) \notin \mathcal{L}(s_i') \cap c(Ap')$, thus $visible(\alpha)$ holds. The other cases and directions are similar. □

Checking of C0, C2 and C3 We also slightly change the original approach to checking the conditions C0, C2 and C3. The first one is trivial to check, since we only have to determine whether I has any enabled transitions. The other two are always invalidated by a counterexample transition, which means that it is sufficient to check them for all enabled transitions once and propagate the invalidation to all composite automata to which it belongs. These conditions are checked for all automata before proceeding with the checking of C1, which is more expensive and therefore only attempted on automata that have passed the first test.

begin
  $C_0 \leftarrow \emptyset$;
  $C_{23} \leftarrow$ Set of all automata;
  foreach $\alpha \in$ `enabled(`s`)` **do**
    $C_0 \leftarrow C_0 \cup$ automata of α;
    if `visible(`α`)` $\vee$ `inStack(`s'`)` **then** $C_{23} \leftarrow C_{23} \setminus$ automata of α;
  end
  foreach $i \in$ Set of all composite automata **do**
    $A \leftarrow$ automata which compose i;
    if $A \cap C_0 \neq \emptyset$ **then** $C_0 \leftarrow C_0 \cup \{i\}$;
    if not $A \subseteq C_{23}$ **then** $C_{23} \leftarrow C_{23} \setminus \{i\}$;
  end
  Candidate set $\leftarrow C_0 \cap C_{23}$;
end

Algorithm 1: Checking of C0, C2 and C3

Checking of C1 We want to determine whether there exists any action in $\bigcup_{i \in I} current_i(s)$ which is a synchronization with an automaton not in I. To do so, we take names of all input (resp. output) actions from each $s_i, i \in I$ and then compare them with names of output (resp. input) actions from all states of all simple automata not in I. Any matching couple is a counterexample for the ample set.

When we find a set that satisfies all four conditions, we accept it as an ample set and use it for the verification instead of the set of all enabled actions.

begin
  $C_{\text{in}}, C_{\text{out}}, O_{\text{in}}, O_{\text{out}} \leftarrow \emptyset$;
  foreach $i \in$ simple automata in I **do**
    $C_{\text{in}} \leftarrow C_{\text{in}} \cup$ names of input actions of i from s_i;
    $C_{\text{out}} \leftarrow C_{\text{out}} \cup$ names of output actions of i from s_i;
  end
  foreach $i \in$ simple automata **not** in I **do**
    $O_{\text{in}} \leftarrow O_{\text{in}} \cup$ names of all input actions of i;
    $O_{\text{out}} \leftarrow O_{\text{out}} \cup$ names of all output actions of i;
  end
  return $(C_{\text{in}} \cap O_{\text{out}} = \emptyset) \wedge (C_{\text{out}} \cap O_{\text{in}} = \emptyset)$;
end

Algorithm 2: Checking of C1

4 Case Study

To provide some evidence for the effectiveness of the partial order method, we have implemented the method within the CoIn verification environment and applied it on a case study. Our previous experience with verification of this model, which has uncovered the need of a partial order reduction method for state/event systems, is reported upon in [8].

The modelled system, the *Trading System*, serves to handle sales in a chain of supermarkets. Its functionality includes the interaction with the cashier at the cash desk, as well as accounting the sale at the inventory. The system is open, designed to interact with external components representing users of the system (cashiers and managers) and a bank application. The model of the system consists of 140 simple CI automata, composed hierarchically into 34 composite automata up to 6 levels of depth. The behaviour of the model features a high degree of independent interleaving of actions, it can be thus expected to achieve a fair amount of state space reduction using the partial order reduction method.

The results obtained by using the method are summarized in Table 1. The various models the verification was performed on were created by complementing the Trading Systems with various components depicting the users of the system.

Table 1. Experimental results (the reduction ratio relates the number of states of the model to the number of states obtained after applying partial order reduction)

Model		without POR			with POR			reduction ratio
		# states	RAM (MB)	time (s)	# states	RAM (MB)	time (s)	
C	2	749 340	139	498	30 618	11	40	24 : 1
C	5	1 498 679	274	1 010	61 771	17	66	24 : 1
C	9	750 684	139	499	30 774	11	40	24 : 1
SC	2	29 341	9	19	2 959	5	4	10 : 1
SC	5	58 681	15	39	6 013	5	6	10 : 1
SC	9	29 629	9	20	2 995	5	4	10 : 1
SCM	2	22 745 391	4 045	21 656	2 016 210	494	2 619	11 : 1
SCM	5	—	—	—	4 084 764	987	4 367	—
SCM	9	22 915 023	4 076	21 864	2 037 002	499	2 640	11 : 1
SCR	2	2 994 016	570	2 119	28 633	11	39	105 : 1
SCR	5	5 988 032	1 128	4 631	58 078	17	67	103 : 1
SCR	9	3 034 336	578	2 150	29 006	10	40	105 : 1
SCSM	2	6 369 598	1 135	4 692	542 794	139	688	12 : 1
SCSM	5	12 739 195	2 263	10 434	1 098 699	273	1 144	12 : 1
SCSM	9	6 413 518	1 143	4 725	548 094	140	694	12 : 1
TSC	2	1 356 277	245	934	37 398	13	48	36 : 1
TSC	5	2 712 553	484	1 936	76 219	22	83	36 : 1
TSC	9	1 373 653	248	948	37 888	13	48	36 : 1

Here, C, SC, SCM, SCR, SCSM and TSC stand for the different user components composed with the system. For each of this variants, three properties were verified, those correspond with properties 2, 5 and 9 as described in [8].

The table shows the number of states of each model combined with each property and the memory and time that was needed to generate the state space, both with and without employing the partial order reduction. A dash (—) in the table means that the information is not available, as the process of state space generation exceeded the maximum of 4 GB of memory. Our experience with applying the partial order reduction on this case study is very positive. In all cases, the state space has been reduced to at least one tenth of the original size and there have been cases where the reduction ratio surpassed one hundred.

5 Conclusion

In our work we present an implementation of the partial-order reduction for state/event LTL in the framework of CI automata. We explain the necessity of modification of the original heuristics for computing ample sets as well as our method of choosing candidate sets, which takes advantage of the hierarchical structure of CI automata. The case study shows how much time and space can be saved using POR in particular cases.

References

1. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction automata as a verification-oriented component-based system specification. In: Proceedings of SAVCBS'05, ACM (2005) 31–38
2. Beneš, N., Brim, L., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: The CoIn Tool: Modelling and Verification of Interactions in Component-Based Systems. In: Proceedings of FACS'08. (2008) 221–225
3. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – a tool for distributed verification. In: Proceedings of CAV'06. Volume 4144 of LNCS., Springer-Verlag (2006) 278–281
4. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Proceedings of IFM'04. Volume 2999 of LNCS., Springer-Verlag (2004) 128–147
5. Peled, D.: All from one, one from all: on model checking using representatives. In: Proceedings of CAV'93. Volume 697 of LNCS., Springer-Verlag (1993) 409–423
6. Beneš, N., Brim, L., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: Partial order reduction for state/event LTL. In: Proceedings of iFM'09. Volume 5423 of LNCS., Springer (2009) 307–321
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. Cambridge, London, MIT Press (1999)
8. Beneš, N., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: A case study in parallel verification of component-based systems. In: Proceedings of PDMC'08. (2008) 35–51