Discrete Event Dynamic Systems: Theory and Applications, 12, 265–286, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Efficient Computation and Representation of Large Reachability Sets for Composed Automata

PETER BUCHHOLZ p.buchholz@inf.tu-dresden.de
*Fakultät für Informatik, TU Dresden, D-01062 Dresden, Germany*

PETER KEMPER kemper@ls4.cs.uni-dortmund.de
*Informatik IV, Universität Dortmund, D-44221 Dortmund, Germany*

**Abstract.** We propose an approach that integrates and extends known techniques from different areas to handle and analyze a complex and large system described as a network of synchronized components. State spaces and transition graphs are first generated for single components. Then, we reduce the component state spaces by using a reachability-preserving equivalence relation. The reduced descriptions are used afterwards for reachability analysis. Reachability analysis is performed in an incremental way that exploits the component structure which defines the adjacency matrix of the transition graph of the complete system as a Kronecker product of small component adjacency matrices. This representation often achieves a significant reduction of the number of transition interleavings to be considered during reachability analysis. An acyclic graph is used to encode the set of reachable states. This representation is an extension of ordered binary decision diagrams and allows for a compact representation of huge sets of states. Furthermore, the full state space is easily obtained from the reduced set. The reduced or the full state space can be used in model-checking algorithms to derive detailed results about the behavior of the modeled system. The combination of the proposed techniques yields an approach suitable for extremely large state spaces, which are represented in a space-efficient way and generated and analyzed with low effort.

**Keywords:** automata networks, reachability analysis, Kronecker representation, equivalence, ordered natural decision diagrams

## 1. Introduction

Functional analysis of hardware and software systems is often based on finite transition systems and requires the generation of the set of all reachable states. Unfortunately, the size of the state space is often extremely large, so that a complete enumeration of all states for a complex system of unknown structure is impossible even with today's computer equipment. Consequently, the development of methods to handle complex state spaces is the main challenge in areas like automatic system verification, supervisor synthesis, or state-based quantitative analysis. A large number of different approaches have been proposed and often successfully used in practice (Clarke et al., 1996). Without being exhaustive, we mention a few approaches. Ordered binary decision diagrams (OBDDs) (Bryant, 1986) are very popular for hardware verification, and allow the representation of extremely large transition systems. However, OBDDs are mainly suitable for systems that can be efficiently described by Boolean functions, which is often the case for hardware, but usually not for software systems. Other approaches analyze models in a compositional way by interleaving composition of components with reduction of component state spaces

due to some form of state aggregation, which is usually based on behavior-preserving equivalence (Cleaveland et al., 1993 and Graf et al., 1996). Interleaving aggregation and composition requires a compositional description of a system. A corresponding notion of structure in complex systems typically results from a modular or hierarchical way of modeling such systems. However, although compositional or hierarchical specification approaches have been established, analysis is often done at the level of the flat system that describes all possible interleavings that can occur. There are very few examples exploiting the compositional or hierachical structure of a model for analysis purposes; notable exceptions are Graf et al. (1996) and Behrmann et al. (1999). If a distributed system is described as a set of interacting (stochastic) automata, a further analysis approach to handle complexity comes from performance analysis. In Plateau (1985), it is shown that the generator matrix of the Markov chain described by a stochastic automata network can be represented in a very compact form as the Kronecker product of much smaller component matrices. This approach can also be used to represent adjacency matrices of transition graphs for untimed models in a compact form (Buchholz and Kemper, 1999).

In this paper, we present an approach that combines and extends the above-mentioned techniques in order to manage the state-space explosion of realistic models. The proposed approach allows a space- and time-efficient generation and presentation of huge transition systems. Starting from a description of a complex system as a set of interacting automata, we first generate matrix representations for the isolated automata. The component matrices can be combined via Kronecker products to describe the adjacency matrix of the complete system in a compact form. Based on this representation, reachability analysis is performed to determine the subset of reachable states in the potential state space that results from the cross product of component state spaces. We present a new and efficient algorithm to perform reachability analysis of compositionally described systems. Components can be reduced locally before composition to reduce the size of the potential state space. However, to perform an a priori aggregation, it is necessary that the reduced components still allow us to compute the reachability set of the original system. We present a suitable equivalence relation that can be efficiently computed, preserves reachability of states, and allows the aggregation of equivalent states to reduce the size of components. Even after reduction of component state spaces and representation of the transition system in a compact form, it is often the case that the resulting aggregated state space is still too large to be completely enumerated. Thus, there is a need for a compact representation of the set of states. For the compact representation, we propose acyclic graphs as an extension of ordered binary decision diagrams. In contrast to binary decision diagrams, in which a node has at most two successors, we allow that a node at level $i$ has one successor for each state in the aggregated state space of the $i$-th component. This representation corresponds to the structure of the state space and is usually very compact, even for huge state spaces. The presented data structure for state representation is a slight extension of other BDD-like structures that have been proposed recently (Ciardo and Miner, 1997). The combination of the outlined steps allows us to generate extremely large state spaces and represent state spaces and transition systems in a very compact form. The representation is afterwards usable in a wide variety of analysis algorithms, including model checking (Clarke et al., 1986). If the description is enhanced by stochastic timing, then quantitative analysis that exploits the compact representation can be realized and allows the efficient analysis of

fairly large models, much larger than it is possible with conventional means (Plateau, 1985; Buchholz, 1999b; Kemper, 1996).

The outline of the paper is as follows. In the next section, we introduce synchronized components, and show how the potential state space and the transition system of composed models can be represented by means of component state spaces and adjacency matrices. Section 3 presents the equivalence relation that preserves reachability, and shows how to compute reduced and equivalent representations for components. In Section 4, reachability analysis of composed models is introduced. Section 5 introduces the graph-based compact representation of the set of reachable states. Section 6 presents an algorithm for incremental reachability analysis that integrates all concepts presented in Sections 2–5. In Section 7, we briefly outline how model-checking algorithms can be integrated with the compact representation. Section 8 describes an example of a concurrent pusher taken from the literature to illustrate the approach. The paper ends with conclusions and an overview of our ongoing work.

## 2. Networks of Synchronized Components

We consider networks of finite state components that communicate via synchronized transitions. This scenario can be specified in various manners, e.g., by process algebras like CCS (Milner, 1989) or CSP (Hoare, 1985), by Petri nets with superposed transitions (Best et al., 1998), or by networks of finite automata with synchronization (Plateau, 1985). Here we consider automata networks, because we focus on analysis aspects rather than specification issues.

DEFINITION 2.1 *An automaton is a 4 tuple* $A = (S, \delta, s_0, L)$ *in which* $S = \{0, 1, \ldots, n-1\}$[1] *is the set of states with cardinality n and initial state* $s_0 \in S$. $\delta \subseteq S \times S \times L$ *is the state transition relation for a finite set of labels* $L = L_c \cup \tau$, $\tau \notin L_c$[2], *such that a transition from a state* $s_x$ *to a state* $s_y$ *carries a label* $l \in L$ *and* $(s_x, s_y, l) \in \delta$.

$\tau$ denotes a special label for internal transitions, as it does in CCS. We consider non-deterministic automata, so $\delta$ is a relation and not necessarily a function. An automaton can be characterized by a set of Boolean adjacency matrices, one for each label. Define $Q_l$ as an $n \times n$ matrix with $Q_l(x, y) = 1$ if $(s_x, s_y, l) \in \delta$ and 0 otherwise. In the following, we use addition and multiplication of Boolean matrices, where addition is defined as Boolean *or* and multiplication as the Boolean *and*. $Q = \sum Q_l$ describes the 1-step reachability relation of $A$. Note that matrices $Q_l$ distinguish labels of events, but $Q$ does not. Furthermore, we define an automaton $A/H$ for $H \subseteq L_c$ that results from $A$ by hiding all transitions with labels from set $H$ using $\tau$, i.e., $A/H$ substitutes all transitions $(s_x, s_y, h) \in \delta$ with $h \in H$ by $(s_x, s_y, \tau)$. At the matrix level the new automaton is described by matrices $Q'_l$ for $l \in L \setminus H$ with $Q'_\tau = Q_\tau + \sum_{h \in H} Q_h$ and $Q'_l = Q_l$ for all $l \in L_c \setminus H$.

For synchronization among a set of $N$ automata $A_1, A_2, \ldots, A_N$ we use the index to characterize the different automata and define the synchronized automaton

$$A = A_1|A_2|\dots|A_N = (\times_{i=1}^{N} S^i, \delta, (s_0^1, s_0^2, \dots, s_0^N), \cup_{i=1}^{N} L^i)$$

with $L_c = \bigcup_{i=1}^{J} L^i$ and the set of synchronization labels $LS \subseteq L_c$. The number of states in $S^i$ is denoted as $n^i$, which implies that $\times_{i=1}^{N} S^i$ includes $\prod_{i=1}^{N} n^i$ states. Synchronization is of the rendezvous type and refers to equal labels in $A_i$ and $A_j$, i.e., $A_i$ and $A_j$ cannot perform $l \in LS$ independently of each other after synchronization. Transitions labeled with $l \notin LS$ can be performed independently in different automata. This kind of synchronization is similar to synchronized transitions in CSP (Hoare, 1985) or in the finite state processes presented in Magee and Kramer (1999). Thus, we do not distinguish input and output events as is done, for example, in I/O-automata (Lynch, 1996), in which input events are always enabled. In our model, any automaton that participates in a synchronization can disable the synchronized transition. The dynamics of the composed automaton are described by an interleaving semantics of transitions. However, a great deal of parallelism exists due to transitions occurring independently in different automata.

Define $PS = \{0, \dots, \prod_{i=1}^{N} n^i - 1\}$ as the set of potentially reachable states. A state from $PS$ can be characterized by an $N$-dimensional vector $(x^1, \dots, x^N)$ for $0 \leq x^i < n^i$ or by a single integer $x = \sum_{i=1}^{N} x^i \prod_{j=1}^{i-1} n^i$ if we define $1 = \prod_{j=1}^{0} n^i$. The latter is a mixed radix number representation. Both representations can be used interchangeably. In a similar way, the transition matrices of the composed automaton on state space $PS$ can be generated from the matrices of the automata $A_1$ through $A_N$ using Kronecker algebra (Davio, 1981 and Plateau, 1985).

DEFINITION 2.2 *Let $Q^1, \dots, Q^N$ be square matrices of dimension $(n^i \times n^i)$; their Kronecker product $Q = \bigotimes_{i=1}^{N} Q^i$ is then defined by $Q(x, y) = \prod_{i=1}^{N} Q^i(x^i, y^i)$ with $x = \sum_{i=1}^{N} x^i g^i$, $y = \sum_{i=1}^{N} y^i g^i$, and weights $g^1 = 1, g^i = n^{i-1} g^{i-1}$ (for $i > 1$).*

*The Kronecker sum $B = \bigoplus_{i=1}^{N} Q^i$ is then given by $\bigoplus_{i=1}^{N} Q^i = \sum_{i=1}^{N} I_{l^i} \bigotimes Q^i \bigotimes I_{r^i}$ in which $I_{l^i}, I_{r^i}$ are identity matrices of dimension $l^i \times l^i$, respectively $r^i \times r^i$ and $r^i = \prod_{j=1}^{i-1} n^j$, $l^i = \prod_{j=i+1}^{N} n^j$. Furthermore, $I(a, b) = 1$ iff $a = b$ and $0$ otherwise.*

Let $Q_l$ be the matrix of the composed automaton for label $l$, and define $Q_l^i = I_{n^i}$ for $l \notin L^i$. Matrix $Q_l$ can be represented as

$$Q_l = \begin{cases} \bigotimes_{i=1}^{N} Q_l^i & \text{for } l \in LS \\ \bigoplus_{i=1}^{N} Q_l^i & \text{otherwise} \end{cases}$$

Synchronization is described by Kronecker products and parallelism by Kronecker sums. Observe that $Q_l$ is an $n \times n$ matrix that is completely described by at most $N$ matrices of size $n^i \times n^i$. The composed automaton is defined by a set of matrices, each represented as the Kronecker product or sum of smaller matrices. Since Kronecker sums and products are associative, these matrices can be used in further compositions; hence, the approach is completely compositional.

With the proposed approach, the characterization of $PS$ and $Q_l$ of composed systems is straightforward. The advantages are a space-efficient representation by a set of matrices and a well-defined mathematical composition operation that establishes the $\delta$ relation such

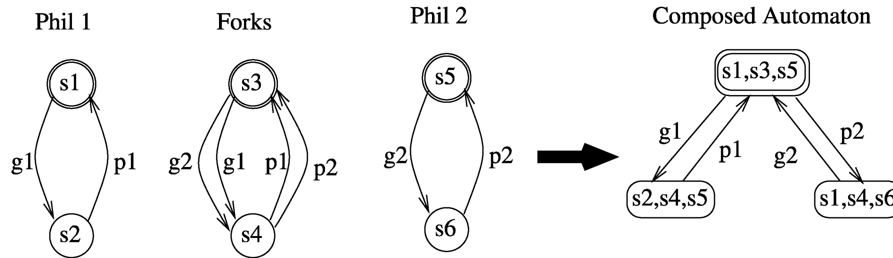


*Figure 1.* Automata description of two dining philosophers.

that state-based algorithms can perform. Such algorithms include state-space exploration using bitstate hashing (Holzmann, 1998), model checking (Clarke et al., 1986), or Markov chain analysis (Stewart, 1994) (the latter presupposes a stochastic interpretation of labels); for an adaptation to Kronecker representations see (Kemper, 1996b; Kemper and Lübeck, 1998; Kemper, 1996a; Buchholz, 1999c). A disadvantage is that only $PS \supseteq RS$ is known. $RS$ is the set of reachable states of the composed automaton (i.e., the reflexive and transitive closure of $\delta$ starting from $s_0$) and $RS$ is often significantly smaller than $PS$.

A small example showing $RS \subset PS$ can be seen in Figure 1, which describes the well-known dining philosophers problem in a configuration of two philosophers and two forks. Each philosopher thinks and eats repeatedly. We let any of the two philosophers pick up simultaneously both forks before eating. In the compositional description, each philosopher is specified by a single automaton with two states, one for thinking and the other for eating. Label $g_i$ indicates that philosopher $i$ gets both forks, whereas label $p_i$ means that philosopher $i$ puts both forks back on the table. The third automaton specifies the two forks that may be available or occupied by one of the philosophers. Initial states are characterized by double circles, and identically named transitions are synchronized. $PS$ contains eight states, but only three of them are reachable, as shown by the composed automaton in Figure 1. Of course, for this small example, a compositional description has no benefits. However, if the number of philosophers is increased and single automata are used to describe some philosophers and the forks lying between these philosophers, then $PS$ is still much larger than $RS$, but the compositional description is much more compact than the global automaton (see Buchholz, 1999b; Buchholz and Kemper, 1999).

Usually $RS$ is more relevant than $PS$ to analyze properties of a model. Consequently, generation of $RS$ from $PS$ and $Q_I$ will be considered (in Section 4). It is well-known that for realistic systems, $RS$ tends to become extremely large, so that it is very difficult to generate and represent the set in a straightforward way even with contemporary computer equipment. This problem is known as the state-space explosion problem, which implies that sophisticated algorithms are required for generating and representing $RS$. In the rest of this paper, we present several steps to alleviate state-space explosion for the networks of automata introduced in this section. In the first step, introduced in the next section, we minimize the size of components in a preprocessing step, to reduce the size of $PS$ and therefore also the effort for the exploration of $RS$.

## 3. Reduction of Components

The approach presented so far is completely compositional: it is possible to exchange components by exchanging corresponding matrices. If a component is replaced by an ''equivalent'' but smaller component, the size of *PS* is obviously reduced. In the following we derive an equivalence relation of the bisimulation type that preserves reachability after composition and is employed to minimize components and thus to minimize the effort for *RS* exploration. Assume that automaton $A_i = (S^i, \delta^i, s_0^i, L^i)$ is replaced with $\tilde{A}_i = (\tilde{S}^i, \tilde{\delta}^i, \tilde{s}_0^i, L^i)$. The latter is called ''aggregated'' in the following. Observe that alphabet $L^i$ is used in both cases to assure that the two automata have identical interfaces. Let $\tilde{Q}_l^i$ be the matrices describing $\tilde{\delta}^i$ and let $\tilde{n}^i$ be the number of states in $\tilde{S}^i$. Replacing $A_i$ with $\tilde{A}_i$ in some composed model means that matrices $Q_l^i$ are replaced with $\tilde{Q}_l^i$ in the Kronecker products and sums. The size of *PS* is reduced by a factor of $\tilde{n}^i/n^i$ assuming $\tilde{n}^i < n^i$. The relation $\tilde{n}^i \leq n^i$ holds in our approach, because $\tilde{A}_i$ is constructed from $A_i$ through the replacement of equivalence classes of states in $A_i$ with single states in $\tilde{A}_i$. The worst case is that each equivalence class will contain a single state, so that $A_i$ and $\tilde{A}_i$ are isomorphic and $\tilde{n}^i = n^i$.

Replacement of $A_i$ with $\tilde{A}_i$ is only useful if both automata behave, in some sense, ''equivalently'', i.e., the properties we want to verify are identical in both cases. Equivalence relations of the bisimulation type are among the most popular of the many existing property-preserving equivalence relations (Milner, 1989). Bisimulation equivalences can be computed efficiently (Cleaveland et al., 1993) and preserve many interesting properties. For the next step, namely the computation of *RS* for the composed model, we need an equivalence that allows the generation of *RS* for the original model from the model in which automata are aggregated. This is not possible with most forms of bisimulation equivalence. However, as shown in Buchholz (1999a,b) in the context of stochastic systems, it is possible to define an extension of bisimulation equivalence that can be applied to deduce the required result. Before we introduce this specific equivalence relation and its computation, we introduce aggregation of automata according to an arbitrary equivalence relation. We first define some notations. For an equivalence relation $\mathscr{R} \subseteq S \times S$ on some finite set $S$, we use $S_{\mathscr{R}}$ to denote the set of equivalence classes and $rep(\tilde{s}_x)$ for $\tilde{s}_x \in S_{\mathscr{R}}$ as the set of elements from $S$ that are collected in equivalence class $\tilde{s}_x$. Consequently, $(s_x, s_y) \in \mathscr{R} \Longleftrightarrow s_x, s_y \in rep(\tilde{s}_x)$ for some $\tilde{s}_x \in S_{\mathscr{R}}$.

DEFINITION 3.1 *Let $A = (S, \delta, s_0, L)$ be an automaton and $\mathscr{R}$ be an equivalence relation on state space $S$. The aggregated automaton according to $\mathscr{R}$ is defined as $A_{\mathscr{R}} = (\tilde{S}, \tilde{\delta}, \tilde{s}_0, \tilde{L})$, in which $\tilde{S} = S_{\mathscr{R}}$, $\tilde{s}_0$ is the unique equivalence class with $s_0 \in rep(\tilde{s}_0)$, $\tilde{L} = L$, and $\tilde{\delta}$ is defined as follows: $(\tilde{s}_x, \tilde{s}_y, l) \in \tilde{\delta} \Longleftrightarrow s_x \in rep(\tilde{s}_x)$ and $s_y \in rep(\tilde{s}_y)$ with $(s_x, s_y, l) \in \delta$ exist.*

Automaton $A_{\mathscr{R}}$ is described by matrices $\tilde{Q}_l$ that are computed elementwise as $\tilde{Q}_l(\tilde{x}, \tilde{y}) = \sum_{s_x \in rep(\tilde{s}_x)} \sum_{s_y \in rep(\tilde{s}_y)} Q_l(x, y)$. For this general concept of aggregation, we have to define appropriate equivalence relations.

We start with observational graphs (Cleaveland et al., 1993). The underlying concept is that the interleaving semantics make it possible to perform non-synchronized transitions independently in the components. In the following, we consider only transitions labeled

with $\tau$ as non-synchronized or internal, because all other transitions can potentially be used for synchronization. In a concrete environment (i.e., composition of automata), all transitions from $L^i \setminus LS$ are internal for $A_i$. By hiding labels from $L^i \setminus LS$, we can explicitly make all labels not used for synchronization internal. Now define $Q_{\tau*} = \sum_{k=0}^{\infty}(Q_\tau)^k$ and $Q_{l*} = Q_{\tau*}Q_lQ_{\tau*}$. Matrix $Q_{\tau*}$ describes the reflexive and transitive closure relation according to internal transitions, and can be computed with an effort cubic in the number of automata states (Cleaveland et al., 1993).

DEFINITION 3.2 *Let $A = (S, \delta, s_0, L)$ be an automaton and $\mathcal{R}$ be an equivalence relation on state space S. $\mathcal{R}$ is a weak inverse bisimulation $\Longleftrightarrow$1) $(s_0, s_x) \in \mathcal{R}$ implies $Q_{\tau*}(0,x)$ and 2) if $(s_x, s_y) \in \mathcal{R}$, then $Q_{l*}(z,x) = 1$ implies $Q_{l*}(z', y) = 1$ for some $s_{z'}$ with $(s_z, s_{z'}) \in \mathcal{R}$ and vice versa.*

Unlike most other bisimulation relations, weak inverse bisimulation considers only the past behavior of an automation, not its future behavior. This justifies the name *inverse bisimulation*. The largest weak inverse bisimulation can be computed with a partition refinement algorithm (see Buchholz, 1999b) like those commonly used to generate bisimulations (Cleaveland et al., 1993).

THEOREM 3.1 *If $\tilde{A}_{\mathcal{R}}$ results from automaton A by an aggregation with respect to some weak inverse bisimulation $\mathcal{R}$, then in every embedding environment the following relation holds:*

1. *if state $\tilde{s}_x \in \tilde{S}$ is reachable after $\tilde{A}$ is embedded, then all $s_x \in rep(\tilde{s}_x)$ are reachable after A is embedded in the same environment, and*

2. *if state $\tilde{s}_x \in \tilde{S}$ is not reachable after $\tilde{A}$ is embedded, then all $s_x \in rep(\tilde{s}_x)$ are not reachable after A is embedded in the same environment.*

**Proof:** The proof of the theorem can be found in Buchholz (1999a,b); we give only an outline here.

Item 2 follows trivially from the definition of transitions in the aggregated automaton. Item 1 can be proved by induction over the number of synchronized transitions in a sequence. Initially, without a synchronized event, all states from $rep(\tilde{s}_0)$ are reachable due to item 1 of Definition 3.2. After one synchronized transition $l$ has taken place, all states in $rep(\tilde{s})$ are reachable if $\tilde{Q}_{l*}(\tilde{s}_0, \tilde{s}) = 1$, due to item 2 in Definition 3.2. This step can be repeated in an induction to show that reachability of a state in the aggregated automaton implies reachability of the set of states that the aggregated state represents in the detailed automaton. ■

In the following, we use the notation $\tilde{A}$ for the automaton that results from $A$ by aggregation according to some weak inverse bisimulation relation; usually the largest weak inverse bisimulation is used. Theorem 3.1 shows that the set of reachable states for $A = A_1|A_2|\ldots|A_N$ can be characterized by the reachability set of $\tilde{A} = \tilde{A}_1|\tilde{A}_2|\ldots|\tilde{A}_N$ and the equivalence classes $rep(\tilde{s}^i)$. Reachability of a state $(\tilde{s}^1, \ldots, \tilde{s}^N)$ in $\tilde{A}$ implies reachability of all states $(s^1, \ldots, s^N)$ with $s^i \in rep(\tilde{s}^i)$ in $A$, and unreachability of

$(\tilde{s}^1, \ldots, \tilde{s}^N)$ implies that all states $(s^1, \ldots, s^N)$ with $s^i \in rep(\tilde{s}^i)$ are also unreachable. The potential state space of $\tilde{A}$ contains $\prod_{i=1}^N \tilde{n}^i$ instead of $\prod_{i=1}^N n^i$ states. For many automata, $\tilde{n}^i < n^i$, so we often reduce the potential state space drastically by an *a priori* aggregation step, without losing the ability to compute *RS*.

## 4. Reachability Analysis

In this section we consider reachability analysis of an automaton $A = A_1 | \ldots | A_N$ that resulted from the composition of $N$ automata according to a set of synchronization labels *LS*. Usually, we can assume that the isolated automata $A_i$ are already reduced, as described in the previous section. However, to avoid an overloading of notation, we use $A_i$ instead of $\tilde{A}_i$ here.

A first algorithm for computing *RS* in the context of Petri nets was published in Kemper (1996b). We consider a significantly improved algorithm for composed automata. Our first observation is that the state-numbering scheme that transfers state $(x^1, \ldots, x^N)$ to integer $x$ is a perfect hash function that assigns to each state from *RS* a unique number from $\{0, \ldots, n-1\}$. Thus, a bit vector of length $n = \prod_{i=1}^N n^i$ is the basic data structure that is used to represent *RS*. The main advantage of the bit vector representation is that the effort needed to decide whether a state has already been reached is in $O(1)$, and the same effort is necessary to include a new state. In tree-like data structures, which are commonly used in state-space generation algorithms, the effort for the membership operation is $O(\log m)$, in which $m$ is the number of states that have already been found.

The usual method for generating the reachability set of an automaton (or a network of automata) is to compute the transitive closure of the reachability relation starting from the initial state. The effort of this approach is proportional to the number of arcs in the reachability graph, and the storage requirements are proportional to the number of states plus an additional queue or stack needed to hold the indices of states for which successor states have not been explored. Thus, an algorithm for reachability analysis would be more efficient if it reduces the number of interleavings of transitions that are considered during reachability analysis. In a distributed setting, like the automata network approach we consider here, it is rather natural to reduce the number of interleavings by separating the successor states that result from independent transitions in different automata. A first approach in this direction was proposed in Buchholz and Kemper (2000); that approach considers all transitions in an automaton that result from a synchronized transition followed by an arbitrary number of local transitions. As shown in Buchholz and Kemper (2000), this step reduces the number of interleavings during reachability analysis, but does not reduce the number of states for which successor states are searched. Here we present an improved algorithm that considers transitions that start with an arbitrary number of local transitions followed by one synchronized transition. With this new approach, it is possible to reduce the number of interleavings and, additionally, the number of states for which successor states are generated. Both effects will be described in more detail below. For the basic step of our algorithm define

$$succ_l^i(x^i) = \begin{cases} \{z^i | Q^i_{\tau*}(x^i, z^i) = 1\} & \text{if } l = \tau, \\ \{z^i | \exists y^i : Q^i_{\tau*}(x^i, y^i) Q^i_l(y^i, z^i) = 1\} & \text{if } l \in L^i_c, \\ \{x^i\} & \text{otherwise} \end{cases}$$

This describes the set of all states reachable in $A^i$ via internal transitions and, if $l \neq \tau$, one $l$-labeled transition. For automata that do not include label $l$, the state is not modified. For a global state $(x^1, \ldots, x^N)$, define the successor function according to label $l$ as

$$SUCC_l(x^1, \ldots, x^N) = \begin{cases} \emptyset & \text{if } l \neq \tau \text{ and } \exists i \text{ with } succ_l^i(x^i) = \emptyset \\ \times_{i=1}^N succ_l^i(x^i) & \text{otherwise} \end{cases}$$

The successor function for the state $(x^1, \ldots, x^N)$ that covers all states reachable with an arbitrary number of local transitions followed by at most one synchronized transition is given by

$$SUCC(x^1, \ldots, x^N) = \cup_{l \in L_c} SUCC_l(x^1, \ldots, x^N) \setminus \{(x^1, \ldots, x^N)\}$$

The transitive closure of the successor relation $SUCC(x^1, \ldots, x^N)$ starting with the initial state equals the set $RS$. To improve reachability analysis, the number of states for which successor states are computed has to be minimized. This can be done using the results of the following theorem.

THEOREM 4.1 *If* $(y^1, \ldots, y^N) \in SUCC_\tau(x^1, \ldots, x^N)$, *then*

$$SUCC(y^1, \ldots, y^N) \subseteq SUCC(x^1, \ldots, x^N)$$

**Proof:** Since $(y^1, \ldots, y^N) \in SUCC_\tau(x^1, \ldots, x^N) Q^i_{\tau*}(x^i, y^i) = 1$ for all $i = 1, \ldots, J$. Consequently, $y^i$ is reachable from $x^i$ by a sequence of internal transitions.

This implies $succ_l^i(y^i) \subseteq succ_l^i(x^i)$, because

$$\begin{aligned} z^i \in succ_l^i(y^i) &\Rightarrow \exists v^i : Q^i_{\tau*}(y^i, v^i) Q^i_l(v^i, z^i) = 1 \\ &\Rightarrow Q^i_{\tau*}(x^i, y^i) Q^i_{\tau*}(y^i, v^i) Q^i_l(v^i, z^i) = 1 \Rightarrow z^i \in succ_l^i(x^i) \end{aligned}$$

Thus we have

$$\begin{aligned} \forall i, \forall l : succ_l^i(y^i) \subseteq succ_l^i(x^i) &\Rightarrow \forall l : SUCC_l(y^1, \ldots, y^N) \subseteq SUCC_l(x^1, \ldots, x^N) \\ &\Rightarrow SUCC(y^1, \ldots, y^N) \subseteq SUCC(x^1, \ldots, x^N) \end{aligned}$$

A direct consequence of the theorem is that the sets $SUCC(x^1, \ldots, x^N)$ for the initial state and for the states that are reached immediately after the occurrence of a synchronized transition describe the complete reachability set. On the basis of these concepts, the algorithm shown in Figure 2 can be used to generate $RS$.

$Reachability_analysis$

```
1.  U = {0} ;
2.  for all y ∈ PS do
3.    if y ∈ SUCC_τ(x_0^1, ..., x_0^N) then
4.      r[y] = 1 ;
5.    else
6.      r[y] = 0 ;
7.  while U ≠ ∅ do
8.    remove (x^1, ..., x^N) from U ;
9.    compute SUCC_τ(x^1, ..., x^N) ;
10.   for all y ∈ SUCC_τ(x^1, ..., x^N) with r[y] = 0 do
11.     r[y] = 1 ;
12.   compute SUCC(x^1, ..., x^N) \ SUCC_τ(x^1, ..., x^N) ;
13.   for all y ∈ SUCC(x^1, ..., x^N) \ SUCC_τ(x^1, ..., x^N) with r[y] = 0 do
14.     r[y] = 1 ;
15.     U = U ∪ {y} :
16.   od
17. od
```

*Figure 2.* Computation of *RS* for composed automata.

We now outline the proof of the correctness of the algorithm, give some implementation hints, and argue why the algorithm is more efficient than conventional reachability analysis for most models.

**Correctness of the algorithm:** Since the set of reachable states equals the transitive closure of the relation $SUCC(..)$ starting with the initial state, we have to show that if a state $(y^1, \ldots, y^N)$ is not put into $U$, then there exists some state $(x^1, \ldots, x^N)$ that has been put in $U$ and $SUCC(y^1, \ldots, y^N) \subseteq SUCC(x^1, \ldots, x^N)$. A state $(y^1, \ldots, y^N)$ is not put into $U$ in line 12 of the algorithm, if $(y^1, \ldots, y^N) \in SUCC_\tau(x^1, \ldots, x^N)$ for some $(x^1, \ldots, x^N)$ that has been removed from $U$. However, by theorem 4.1 this implies $SUCC(y^1, \ldots, y^N) \subseteq SUCC(x^1, \ldots, x^N)$.

**Implementation hints:** In the algorithm, $U$ is a set for storing non-explored states; usually, these states are stored as integers rather than vectors, but other data structures are possible as well. **r** is the hash table, which is bit vector of length $n$. To minimize the number of states in $U$, we first compute successor states that are reachable only by local transitions (steps 9–11), because these states do not need to be put into $U$. Afterwards, synchronized transitions are considered (steps 12–14), and only the states that have not been reached before must be put into $U$.

**Complexity of the algorithm:** The worst-case complexity of the algorithm improves conventional reachability analysis by a factor of $\log n$, because of the use of the perfect

hash function for states. This holds as long as vector **r** fits in main memory. However, in almost all examples, we obtained much larger improvements than $\log n$. There are three reasons for the improvements:

1. Pre-aggregation reduces the size of the state spaces in which reachability analysis is performed.

2. The set of states for which successor states are computed is restricted to those states that are reached immediately after a synchronized transition occurred and that have not been reached before via internal transitions. In automata networks with some internal behavior of the automata, this significantly reduces the number of states that are put in $U$.

3. Due to the computation of sets of successor states, only very few interleavings are considered. A simple example is a system with three automata, each of which has two local transitions. Each local transition ends in a different state. Each set $succ_l^i(x^i)$ contains three states that are generated from two transitions. Thus, 9 states have to be stored, and 6 transitions are necessary to generate the sets. The reachability graph resulting from the composition of the three automata contains 27 states that result from 54 transitions; if reachability analysis is performed for the global system, all transitions are considered during computation of the reachability set.

The proposed algorithm is efficient as long as vector **r** fits in main memory. Since **r** stores each state of *PS* as a single bit, relatively large state spaces can be handled with contemporary workstations. However, for realistic systems, *PS* is often too large even after the size of components has been reduced. In that case, it is possible to perform reachability analysis in an incremental way. We will discuss the corresponding algorithm in Section 6 after presenting the compact representation of *RS* in the next section.

## 5. Compact Representations of Reachability Sets

For the representation of *RS*, we have already introduced three different structures, namely the bit vector of length $n$, the vector representation $(x^1, \ldots, x^N)$, and an integer representation. All representations require a great deal of space if *PS* and *RS* are large (i.e., in the range of several billion states). To represent *RS* in a compact form, we borrow ideas from ordered binary decision diagrams (OBDDs) (Bryant, 1986) and multi-valued decision diagrams (MDDs) (Wegener, 2000) to represent *RS* as an acyclic graph with $N$ levels, for a system of $N$ composed automata (Buchholz and Kemper, 1999). The difference to OBDDs and MDDs is that we can allow nodes to have a variable number of outgoing arcs, and that nodes of the lowest level do not refer to Boolean values. The idea is that a node at level $i$ for $1 \leq i \leq N$ represents the reachable subset of states in component $i$ that are subject to the condition of states of components at levels $j < i$.

To introduce the concept, we first note that *PS* can be represented by a tree with $N$ levels, where each node at level $i$ has $n^i$ sons. A path in the tree corresponds to a state $(x^1, \ldots, x^N)$.

If we eliminate from this structure all paths to unreachable states, we obtain a representation of $RS$. Nodes at level $i$ describe reachable states from $S^i$. Let $RS(x^1, \ldots, x^i) = \{y \in RS | x^1 = y^1 \wedge \ldots \wedge x^i = y^i\}$, the subset of states from $RS$ with fixed states for the automata 1 through $i$. $RS(x^1, \ldots, x^i)$ refers to a subtree with root node at level $i+1$; the root note is denoted by $RS^{i+1}(x^1, \ldots, x^i)$. $RS^{i+1}(x^1, \ldots, x^i)$ is the subset of states from $S^{i+1}$ that are reachable if the state of the automata 1 through $i$ equals $(x^1, \ldots, x^i)$. To complete the notation, $RS() = RS$ and $RS^1()$ is the root node of the tree.

Two subtrees $RS(x^1, \ldots, x^i)$ and $RS(y^1, \ldots, y^i)$ are equal, if and only if $RS^i(x^1, \ldots, x^{i-1}) = RS^i(y^1, \ldots, y^{i-1})$ and all pairs of subtrees $RS(x^1, \ldots, x^i, x^{i+1})$, $R(y^1, \ldots, y^i, y^{i+1})$ with $x^{i+1} = y^{i+1}$ are equal. By applying a folding operation in a bottom-up manner, like that of OBDDs, we represent equal subtrees only once, and we obtain a unique acyclic graph (DAG) for a given ordering of components (whose set of paths is equal to the set of paths in the tree and represents the states from $RS$). By introducing appropriate arc inscriptions, it is possible to derive for each state from $RS$ a number $\{0, \ldots |RS| - 1\}$ that corresponds to its relative position in $PS$; i.e., for $x < y$ and $x, y \in PS \cap RS$, $x$ receives a smaller number than $y$; see Buchholz and Kemper (1999) for further details.

We consider a small example to clarify the representation of $RS$ by means of acyclic graphs. The automata model and the corresponding representation of $RS()$ are shown in Figure 3. In the model, 12 out of 18 states are reachable. The graph representation of $RS$ is very compact, because it allows the folding of several subtrees. For large systems, the graph representation often remains compact, even when the size of $PS$ or $RS$ explodes. This can also be seen in the example presented in Section 8. As in other BDD-like structures, the ordering of components determines the size of the resulting data structure; furthermore, as in other graph structures, an optimal ordering cannot be determined efficiently, but heuristics based on the number of component states and the synchronization structure may be found.

In Figure 3, the state space is represented without a priori aggregation of components. However, using the equivalence relation defined above, it is possible to reduce automata

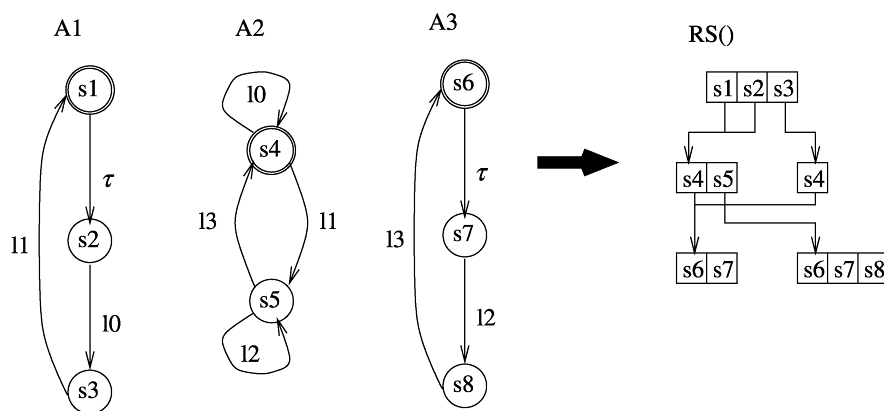


*Figure 3.* Automata model and the corresponding representation of $RS$ as an acyclic graph.

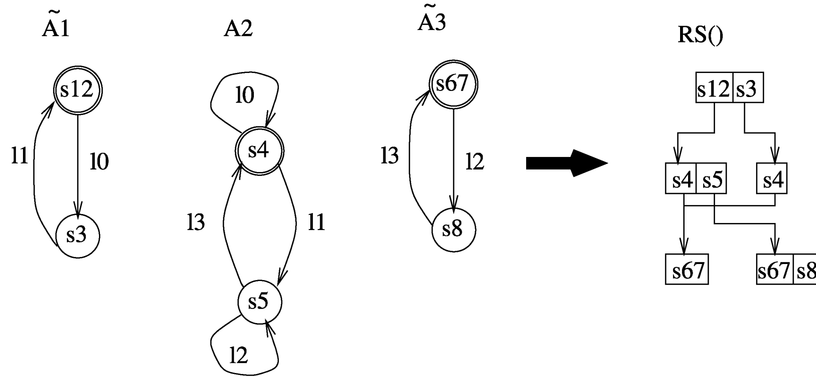


*Figure 4.* Aggregated automata and the corresponding graph representation of *RS*.

$A1$ and $A3$ from three to two states. The aggregated automata and the corresponding representation of *RS* is shown in Figure 4. Only states that represent equivalence classes with more than one member are renamed. Thus, $s12$ represents $s1$ and $s2$ and $s67$ represents $s6$ and $s7$. Due to the aggregation condition that preserves reachability of states, the structure of the graph is identical for the aggregated and unaggregated systems, but the nodes of the graph for the aggregated system may contain fewer elements, yielding a more compact representation.

## 6. Incremental Reachability Analysis

If *PS* is extremely large, even a bit vector for hashing can be too costly. Therefore, we formulate an incremental reachability analysis algorithm that is based on the observation that if $PS >> RS$, then $PS^I >> RS^I$ often holds for a subset $I$ of synchronized automata. If subset $I$ is explored beforehand, one can proceed with $RS^I$ instead of $PS^I$ for the overall exploration. Define a partition $\mathcal{N}^1, \ldots, \mathcal{N}^M$ on the set of components such that $\mathcal{N}^J \subseteq \{1, \ldots, N\}$, $\mathcal{N}^I \cap \mathcal{N}^J = \emptyset$ and $\cup_{I=1}^M \mathcal{N}^I = \{1, \ldots, N\}$. Each partition group $\mathcal{N}^I$ describes composition of a subset of components, and the resulting composed model can be analyzed with the methods presented so far. Thus, $A = A_1 | \ldots | A_N$ is described by $A = A^1 | \ldots | A^M$, and $A^I = A_{i_1} | \ldots | A_{i_l}$ with $i_j \in \mathcal{N}^I$. The set of synchronized labels *LS* is in all cases identical. $PS^I = \times_{i \in \mathcal{N}^I} S^i$ is the potential state space for the subset of components in $\mathcal{N}^I$. $PS^I$ contains $\prod_{i \in \mathcal{N}^I} n^i$ states. We consider a partition to be feasible if for each $PS^I$ a bit vector fits in primary memory. We search for partitions that are feasible and include a small number of partition groups. It is usually not necessary to find the ''best'' partition for the automata, in the sense that the number of partition groups is minimized. Usually, it is more important to collect into a partition group those automata that are strongly synchronized. For example, one adequate heuristic algorithm that has been implemented groups automata that are synchronized until the size of $PS^I$ becomes too large. Of course, it cannot be assured that a feasible partition exists, even if *RS* is small enough to fit in memory. In those cases, one has to use secondary memory that slows down the generation,

or already reached states are stored in the graph structure for *RS*, thus introducing a $\log n$ overhead.

Let $RS^I$ be the set of reachable states for the model composed of the components from $\mathcal{N}^I$. Knowing $RS^I$ and $RS^J$ for partition groups $\mathcal{N}^I$ and $\mathcal{N}^J$, we can define $PS^{I+J}$ as $RS^I \times RS^J$, which is smaller than *PS* resulting from $\mathcal{N}^I \cup \mathcal{N}^J$ whenever $RS^I$ and $RS^J$ are smaller than $PS^I$ and $PS^J$ (i.e., whenever the composition of some components produces unreachable states in the potential reachability set). Obviously, it is good to select components with tight synchronization for a partition group.

We can now outline the following algorithm for compositional reachability analysis:

1. Reduce all components with respect to reachability as proposed in Section 3.

2. Find a feasible partition with a small number of partition groups.

3. For each subset of components, perform reachability analysis as described in Section 4, and represent the set of reachable states in compact form as introduced in Section 5. If all automata belong to a single partition group, then the algorithm terminates successfully.

4. If more than one partition group exists, then find a new feasible partition by combining partition groups, and go to step 3, for which only new partition groups have to be analyzed. If no such partition exists, then analysis cannot be performed with the available memory.

If the algorithm terminates successfully, then we obtain a DAG representation of *RS* and a Kronecker representation of the transition system. Note that if step 3 is reached after step 4, computation of *SUCC* requires that one either adapt the Kronecker representation to the explored partition groups, which usually requires more space since matrix dimensions increase, or compute *SUCC* from the original Kronecker representation and translate state indices among the different representations.

Once a DAG of *RS* is available, it can be used in two ways:

1. The bisimulation remains present, so to decide whether state $(x^1, \ldots, x^N)$ is reachable, we first need representation $(\tilde{x}^1, \ldots, \tilde{x}^N)(x^i \in rep(\tilde{x}^i))$. It can be found in a time proportional to the number of components, if we store with each $x^i$ the corresponding $\tilde{x}^i$. Afterwards, state $(\tilde{x}^1, \ldots, \tilde{x}^N)$ has to be found in the DAG; this requires at most $N$ steps with an effort of $O(log\ \tilde{n}_i)$ at level $i$.

2. The bisimulation is resolved, i.e., each $\tilde{x}^i$ is replaced by its elements in each node of the DAG. This transformation is in $O(G \cdot n)$ if the DAG contains $G$ nodes and each component contains at most $n$ states. Although this approach slightly enlarges the space used by the DAG, it results in a DAG representation of *RS* that is simpler to combine with other analysis methods. For example, a different bisimulation can be applied subsequently to obtain a reduced representation for certain properties, or model-checking algorithms (see Section 7) can be applied using other analysis tools.

In conventional algorithms that completely enumerate $RS$ and store it in a tree-like structure, searching for a specific state requires an effort in $O(\log |RS|)$, which is usually much higher than the effort to access a DAG. Additionally, our representation requires only a small fraction of the memory that is needed for storing the complete set $RS$. Based on the DAG structure, state-based analysis can be applied on the DAG of either the aggregated or the original model. The disaggregation step expands the DAG of the aggregated state space to the DAG of the original state space, but it leaves the number of nodes in the DAG unchanged; only the cardinality of single nodes is enlarged. The choice between these alternatives depends on the required results. If the aggregated DAG includes enough information, it is usually preferable to do analysis at that level. Since the set of successor states of a state reachable by specific transitions can be generated efficiently from the Kronecker representation, typical state-based analysis algorithms can be applied.

## 7. Model Checking of Composed Systems

To analyze more complex functional properties of discrete event systems, it is necessary to express the properties to be proved in some formal notation. Usually, temporal logics are used for this purpose. The process of proving a logical formula for a finite transition system is called model checking (Clarke et al., 1986 and Clarke et al., 1996). Very roughly, model checking performs an exhaustive search algorithm on the state space to verify for which states the required property holds and for which states it does not hold. If a system is described in a compositional form, it is sometimes possible to exploit the component structure for a more efficient analysis. Two different types of model checking exist. One of them uses an efficient search procedure to check the required properties, and the other one represents the system and the negated property as automata and checks whether the intersection of the languages of both automata is empty. Both approaches create state-space explosion problems that need to be alleviated. OBDDs have been successfully used for model checking, allowing the analysis of extremely large systems provided that the system can be coded by Boolean formulas (Burch et al., 1992). In a similar way, model-checking algorithms can be combined with the graph-based representation of $RS$ and the Kronecker representation of the composed automaton presented in this paper. In fact, we have implemented model checkers of both types for the compositional structure in the APNN toolbox (Bause et al., 1998 and Kemper and Lübeck et al., 1998).

Here we present briefly a model checker of the first type that uses computational tree logic (CTL) for specifying the desired properties. CTL is a temporal logic that is commonly used, because it can be checked efficiently; however, not all properties can be expressed by CTL. Since CTL model checking is now well-established, we only give a very brief introduction of the logic and the basic functions for checking a formula for a finite transition system; further details can be found in the literature (Clarke et al., 1986 and McMillan, 1993). We present in some more detail the advantages that arise from the use of the compositional representation in model-checking algorithms.

CTL formulas are built from atomic propositions, logical compositions, and temporal

operators. Atomic propositions are defined as Boolean functions that have the state of the system as an argument; i.e., $f((x^1, \ldots, x^N)) \rightarrow \mathbb{B}$ is an atomic proposition. Often $f$ can be defined compositionally such that

$$f((x^1, \ldots, x^N)) = \bigvee_{i=1}^{J} f_i(x^i) \text{ or } f((x^1, \ldots, x^N)) = \bigwedge_{i=1}^{J} f_i(x^i)$$

or any other combination of $\vee$ and $\wedge$ composition. In this case the set of states observing the proposition can be very efficiently generated from the DAG representation of *RS*. Since $\neg(a \vee b) = \neg a \wedge \neg b$ it is sufficient to consider the $f((x^1, \ldots, x^N)) = \wedge_{i=1}^{J} f_i(x^i)$.

To determine the set of states that observe the formula, it is sufficient to run once through the DAG for *RS* to generate a DAG that includes all states that observe atomic proposition. At each level $j$, only those states $x^j$ for which $f_j(x^j) = 1$, and therefore implicitly also $f_i(x^i) = 1 (i < j)$ holds, remain in the DAG. Other states and their successors are deleted.

This approach can also start with the DAG without resolving the bisimulation. In that case a state $\tilde{x}^j$ is kept in the DAG if $f_j(x^j) = 1$ for some $x^j \in rep(\tilde{x}^j)$. After generation, it is easy to generate the DAG including all states that observe the proposition by expanding $(\tilde{x}^1, \ldots, \tilde{x}^J)$ only for those states $(x^1, \ldots, x^N)$ for which $x^i \in rep(\tilde{x}^i)$ and $f_i(x^i) = 1$ for all $i = 1, \ldots, J$.

Each atomic proposition is a CTL formula. CTL formulas can be composed via logical operations. Thus, if $f$ and $g$ are CTL formulas, then so are

$$\neg f, f \wedge g, f \vee g, f \Rightarrow g, f \Leftrightarrow g$$

with the usual meaning. A DAG representation of the set of states that observe a composed formula can be generated from the DAG representations of the subformulas. The corresponding operations are similar to the same operations for other BDD data structures (Wegener, 2000).

Additionally, CTL contains temporal operators to express conditions over paths. The temporal operators are *nexttime* (X), *globally* (G), *sometimes* (or *finally*) (F), and *until* (U). These operators have to be combined with the all-path quantifier (A) or the exists-path quantifier (E). The set of states that observe a temporal formula is computed by a search algorithm on the reachability graph (Clarke et al., 1986); the search algorithm can be combined with the Kronecker representation of composed automata networks (Kemper and Lübeck, 1998).

## 8. Example

As a larger example, we consider the modeling of the control software of a system of concurrent pushers. Pushers are devices used in manufacturing systems to move an item from one position to another. A chain of pushers can be built to move an item from A to B, from B to C, and so forth. A small chain of pushers is shown in Figure 5. Pushers are driven

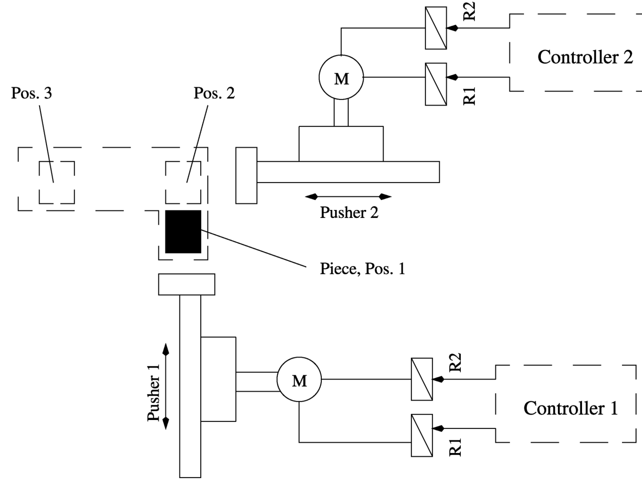


*Figure 5.* Chain of two pushers.

by electric motors and need to have controllers in order for their movements to be useful within a larger setting. Heiner[3] (1997) provides a Petri net model of $N$ pushers and their controllers, which is a safe net that is scalable by parameter $N$. The model naturally partitions into $N+2$ components, including one per pusher plus a consumer and a producer component that consume and produce items to be pushed around. We consider the time and space requirements for the DAG generation of the reachability set, which includes:

1. Generation of a set of synchronized automata for the Kronecker representation.

2. Aggregation of these automata by weak inverse bisimulation.

3. Kronecker-based state-space exploration of the aggregated system, yielding a DAG.

4. Disaggregation of the DAG from the aggregated system into the DAG of the original system.

Table 1 gives the results observed for increasing numbers of $N$ pushers (first column) yielding a system with $C$ components (second column). The number of places and

*Table 1.* Analysis results for an increasing number of pushers.

| N | C | P/T | $\|R\|$ | $\|\hat{R}\|$ | DAG Nodes | DAG KB | Time $\hat{R}$ t. cl. | Time $R$ t. cl. | Time $R$ Basic |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 10 | 150/140 | $2.644e+6$ | 49,266 | 17 | 1.2/2.6 | 5 | 250 | 1,333 |
| 10 | 12 | 186/174 | $1.073e+9$ | 682,686 | 37 | 2.6/5.9 | 18 | — | — |
| 12 | 14 | 222/208 | $4.204e+10$ | $9.508e+6$ | 53 | 3.7/8.5 | 260 | — | — |
| 14 | 16 | 258/242 | $1.640e+12$ | $1.324e+8$ | 69 | 4.9/11.1 | 4,435 | — | — |

transitions are given in the third column to illustrate the size of the Petri net model. Columns 4 and 5 give the dimensions of the state space of the original system $|R|$ and the aggregated system $|\hat{R}|$. The number of nodes is the same for the DAGs of both systems (column 6). The DAGs differ only in the number of outgoing arcs; this causes the DAGs to have different sizes (column 7, given in kilobytes). For example, for $N = 8$, the DAG of $\hat{R}$ uses 1.2 KB, while the one for $R$ uses 2.6 KB. The last three columns give the computation times in seconds for ''wall clock time''. The value time $\hat{R}$ (column 8) describes the time it takes for aggregation to first reduce the components, and for reachability analysis of the aggregated system, using the algorithm presented in this paper (i.e., the transitive closure of local transitions is exploited). The time includes the disaggregation step that generates the DAG representing $R$ from $\hat{R}$. The next to last column shows the time it takes to generate the DAG for $R$ without local aggregation of the automata, but with the reachability analysis algorithm presented here. The last column shows the time it takes to generate $R$ with a conventional reachability analysis algorithm that does not precompute the transitive closure of local transitions, but uses the Kronecker representation for the automata matrices and the DAG for state-space representation. Since the effort is much higher without aggregation than with aggregation, we stopped the computation without aggregation for $N > 8$. The same software is used for state-space exploration with and without aggregation, but for the aggregated system, an incremental generation is only necessary for $N \geq 8$, while for the original system, it is used for all $N \geq 5$. Comparing the results in the last two columns, one can see that the new reachability analysis algorithm reduces the generation time by a factor of 6; this is, of course, model-dependent. Furthermore, the new algorithm requires less memory, because the number of states to be stored in $U$ is reduced. In the example, memory requirements are reduced by a factor of 3, e.g., for $N = 8$ the conventional approach requires 87 MB for state-space generation, and the new approach only 27 MB. All results are obtained on a PC with a 550 Mhz processor and 1 GB primary memory. Programs are written in C and compiled with the gcc and compiler option $-O4$.

The user's ability to select partitions is a degree of freedom in our approach that is useful for influencing performance of the algorithms but also for checking consistency. The table below considers models for $N$ up to 16 with partitions that result from grouping adjacent pushers into components, i.e., for $N = 16$ we build $C = 8$ components by (producer, pusher1, pusher2), (pusher3, pusher4), ..., (pusher15, pusher16, consumer). For this partition, it is possible to reduce the size of the components significantly by applying the aggregation. Thus, the resulting state space of the aggregated system is moderate and can be generated with great efficiency. It takes only 17 seconds to generate the overall state space of the model with $N = 16$ that contains $6.39e+13$ states. Obviously, that is not the limit of the method on the current machine. For the system with $N = 16$ pushers described by $C = 8$ components, the potential state space resulting from the cross product of component state spaces contains $4.94e+19$ states, of which $6.39e+13$ are reachable.

However, we should be honest and point out, that the example shows a behavior that is very convenient for our method, because components communicate only with their neighbors. If such a case, a grouping of adjacent components makes synchronizations internal and usually allows a significant reduction of the component state spaces by aggregation. In models with a more complex synchronization structure, we cannot expect

such extreme effects. Nevertheless, as shown by the results for $N = 8$ without pre-aggregation, reachability analysis can be performed on our current machine with the proposed algorithm for systems with up to $10^8$ states. Since in almost all cases, pre-aggregation of components yields some reduction of the component state spaces, even models with a complex synchronization structure can be analyzed for large state-space sizes. Since the effort of computing the aggregates is cubic in the number of component states, it is not possible to enlarge components arbitrarily; there is always a trade-off between the effort for aggregation and the effort for reachability analysis. Few large component state spaces increase the effort for aggregation and usually decrease the effort for reachability analysis, many small component state spaces have the opposite effect.

Heiner exercises the model for different analysis tools, including INA, PEP, and PROD, which implement invariant analysis, conventional state-space exploration, stubborn set methods, and the prefix algorithm. Our results are consistent with respect to $|R|$ for $N = 5, 6$, but they differ for $N = 7$ from the results in Heiner (1997); for $N \geq 8$ no values are given in Heiner (1997). Our exploration approach outperforms the other tools evaluated there. Of course, the effort of the approach must also be compared with that of OBDD techniques. First experiences show that analyzable state-space sizes and the performance of the algorithm are comparable to established OBDD-based tools, but that our approach handles a different class of models, as OBDD techniques rely on a Boolean representation of states; that can be inefficient for general models. Furthermore, OBDD-based techniques do not exploit the compositional structure of a model. Our results show that the presented state-based analysis is a useful addition to the variety of approaches available for the functional analysis of discrete event systems.

One purpose of reachability analysis is the detection of functional properties by model checking. In Heiner (1997), the pusher model is analyzed with respect to many properties. In this paper, we focus on some safety properties that are requirements of the real system and that can be computed with great efficiency; i.e., we consider properties P2, P5, and P6 of Heiner (1997):

**P2** requires, that at any time, a pusher can be driven in one direction only: $\mathbf{AG}(\neg(Pi_R1_on \wedge Pi_R2_on))$ where $Pi_R1_on$ and $Pi_R2_on$ indicate controler states of pusher $Pi$ that demand the pusher to drive in and out.

**P5** requires, that no pusher should extend too far: $\mathbf{AG}(\neg(Pi_px \wedge Pi_py))$ where $Pi_px$ and $Pi_py$ are specific places in the Petri net model of pusher $Pi$. If both places are marked in a state, then the pusher can extend too far.

**P6** requires, that while a pusher moves, no new work piece must arrive at its input position: $\mathbf{AG}(Pi_position_full \Rightarrow Pi_basic)$ where $Pi_position_full$ refers to the marking of

*Table 2*. Results for partitions enlarged by the grouping of pushers.

| N | C | P/T | $\|R\|$ | $\|\hat{R}\|$ | DAG Nodes | DAG KB | Time $\hat{R}$ t. cl. |
|---|---|---|---|---|---|---|---|
| 14 | 7 | 258/242 | 1.640e + 12 | 122,617 | 27 | 2.3/26.1 | 7 |
| 15 | 8 | 276/259 | 1.024e + 13 | 595,541 | 35 | 2.9/20.3 | 12 |
| 16 | 8 | 294/276 | 6390e + 13 | 755,600 | 35 | 3.0/31.3 | 17 |

an input place and *Pi_basic* describes that the pusher is in a state where it is not moving but awaiting an input.

Clearly, these properties shall hold for all pushers $Pi, i = 1, 2, \ldots, N$. We can evaluate these formulas by searching the DAG of $R$ for appropriate states. Note, that each formula considers only a single pusher $Pi$, such that its atomic proposition can be computed on behalf of $S^i$. If the evaluation yields the same value for all states $s \in S^i$, for example, *true*, then the atomic proposition has this value for all states in $R$. However, if the evaluation yields different values for different states $s, s \in S^i$, as it is the case for formulas $P2$, $P5$, and $P6$ above, then we need to check states in the nodes at a level of the DAG of $R$ that corresponds to the component with pusher $Pi$. Since the DAG contains only few nodes per level, this evaluation is extremely fast, i.e., measuring cpu time and user time gives results as 0.0 seconds, which means that the system library functions are simply not precise enough to measure the computation, so the real value is significantly less than a single second. The effort does not vary among the selected pushers $i$ for any $i \in \{1, \ldots, 16\}$. Note that this class of properties is the one that is analyzed most efficiently, for other properties like liveness properties the computational burden is significant and requires other techniques for model checking. Nevertheless, the class of properties we selected is of interest, and by doing so we could restate some results of Heiner (1997) for increased values of $N$; i.e., properties $P2$ and $P6$ are fulfilled, but $P5$ is not fulfilled. For the sake of completeness, we recall from Heiner (1997) that property $P5$ requires additional constraints on timing to be fulfilled by the model as well as by the real system.

## 9. Conclusions

In this paper, we present an approach to compute and represent the set of reachable states for a network of automata in a compact way. Analysis is compositional and combines steps that include reduction of components due to equivalence, creation of Kronecker representations of large graphs, and creation of compact representations of large sets by acyclic graphs. The basic model class, that of automata composed via synchronized transitions, is very general and is used as a low-level view for several other specification techniques, like process algebras and specific classes of Petri nets. The approach allows the handling of much larger state spaces than is possible with conventional means. Since our approach supports Depth-First-Search strategies as well as Breadth-First-Search strategies, it should be useful to combine our approach with the method of Stern and Dill (1998) that uses second-level memory. Unlike OBDD techniques, our approach cannot go beyond the state spaces described in Burch et al. (1992), but the kind of models that can profit from our approach is larger, since we do not restrict ourselves to Boolean function representations. An obvious future improvement to our approach would be to adopt the efficient methods of OBDDs to compute sets of successor states from sets of predecessor states (Burch et al., 1992). The approach can be easily adopted for our DAG data structure, but has not yet been implemented in our tool environment. Thus, we did not introduce it here.

The presented method has been integrated in a toolbox for the quantitative and qualitative analysis of discrete event dynamic systems described as Petri nets with

transition fusion or at the automata level (Bause et al., 1998 and Buchholz and Kemper, 1999). Thus, the approach is completely automated and requires no user interaction once the system has been defined as a set of interacting components. It is applied in combination with an available model checker for computational tree logic and various quantitative analysis tools.

Future directions in the area of compositional analysis include further integration of OBDD techniques, parallelization of analysis algorithms, the extension of equivalence relations, and further exploitation of the model structure in CTL model-checking algorithms.

## Notes

1. The state space of an automaton is isomorphic to a finite set of integers. It depends on the context whether we use the notation $x$ or $s_x$ for the $x$-th state in the set.
2. $L_c$ denotes the set of labels that can be used to observe the dynamics of the automaton, and will also be used to compose automata via synchronized transitions.
3. We thank M. Heiner for web access to the original model at http://www-dssz.Informatik. TU-Cottbus.DE.

## References

Bause, F., Buchholz, P., and Kemper, P. 1998. A toolbox for functional and quantitative analysis of DEDS. In R. Pujanger, N. N. Savino, and B. Serra, (eds), *Quantitative Evaluation of Computing and Communication Systems* 356–359, Springer LNCS 1469.

Behrmann, G., Larsen, K., Andersen, H. R., Hulgaard, H., and Lind-Nielsen, J. 1999. Verification of hierarchical state/event systems using reusability and compositionality. In W. R. Cleaveland (ed.) *Tools and Algorithms for the Construction and Analysis of Systems* 163–177, Springer LNCS 1579.

Best, E., Fraczak, W., Hopkins, R. P., Klaudel, H., and Pelz, E. 1998. M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica* 35: 813–857.

Bryant, R. E. 1986. Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computer* 35(8): 677–691.

Buchholz, P. 1999a. Exact performance equivalence—an equivalence relation for stochastic automata. *Theoretical Computer Science* 215(1/2): 263–287.

Buchholz, P. 1999b. Hierarchical structuring of superposed GSPNs. *IEEE Transactions on Software Engineering* 25(2): 166–181.

Buchholz, P. 1999c. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics* 31(4): 375–404.

Buchholz, P., and Kemper, P. 1999. Modular state level analysis of distributed systems—techniques and tool support. In R. Cleaveland (ed.), *Tools and Algorithms for the Construction and Analysis of Systems* 420–434, Springer LNCS 1579.

Buchholz, P., and Kemper, P. 2000. Efficient computation and representation of large reachability sets for composed automata. In R. Boel and G. Stremersch (eds), *Discrete Event Systems Analysis and Control* 49–56, Kluwer Academic.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. 1992. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation* 98(2): 142–170.

Ciardo, G., and Miner, A. S. 1997. Storage alternatives for large structured state spaces. In R. Marie and B. Plateau (eds), *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation* 44–57, Springer LNCS 1245.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions and Programming Languages and Systems* 8(2): 244–263.

Clarke, E. M., and Wing, J. M. et al. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys* 28(4): 626–643.

Cleaveland, R., Parrow, J., and Steffen, B. 1993. The concurrency workbench: a semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems* 15(1): 36–72.

Davio, M. 1981. Kronecker products and shuffle algebra. *IEEE Transactions on Computers* 30: 116–125.

Graf, S., Steffen, B., and Lüttgen, G. 1996. Compositional minimization of finite state systems. *Formal Aspects of Computing* 8(5): 607–616.

Heiner, M. 1997. Verification and optimization of control programs by Petri nets without state explosion. In *Proc. 2nd Workshop on Manufacturing and Petri Nets* 69–84, available via http://www-dssz.Informatik.TU-Cottbus.DE/ wwwdssz/

Hoare, C. 1985. *Communicating Sequential Processes*. Prentice Hall.

Holzmann, G. J. 1998. An analysis of bitstate hashing. *Formal Methods in System Design* 13(3): 301–314.

Kemper, P. 1996. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering* 22(9): 615–628.

Kemper, P. 1996. Reachability analysis based on structured representations. In J. Billington and W. Reisig (eds), *Application and Theory of Petri Nets 1996* 269–288, Springer LNCS 1091.

Kemper, P., and Lübeck, R. 1998. Model checking based on Kronecker algebra. Forschungsbericht 669, Fachbereich Informatik, Universität Dortmund.

Lynch, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Magee, J., and Kramer, J. 1999. *Concurrency—State models & Java programs*. Wiley.

McMillan, K. L. 1993. *Symbolic model checking: an approach to the state space explosion problem*. Kluwer.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall.

Plateau, B. 1985. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. *Performance Evaluation Review* 13: 142–154.

Stern, U., and Dill, D. L. 1998. Using magnetic disk instead of main memory in the Mur$\varphi$ verifier. In A. J. Hu and M. Y. Vardi (eds), *CAV'98* 172–183, Springer LNCS 1427.

Stewart, W. J. 1994. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press.

Wegener, I. 2000. *Branching Programs and Binary Decision Diagrams*. Monographs on Discrete Mathematics and Applications. SIAM.