Automatic Synthesis of Assumptions for Compositional Model Checking*

Bernd Finkbeiner[1], Sven Schewe[1], and Matthias Brill[2]

[1] Universität des Saarlandes
66123 Saarbrücken, Germany
{finkbeiner, schewe}@cs.uni-sb.de
[2] Carl von Ossietzky Universität
26121 Oldenburg, Germany
matthias.brill@informatik.uni-oldenburg.de

Abstract. We present a new technique for automatically synthesizing the assumptions needed in compositional model checking. The compositional approach reduces the proof that a property is satisfied by the parallel composition of two processes to the simpler argument that the property is guaranteed by one process provided that the other process satisfies an assumption A. Finding A manually is a difficult task that requires detailed insight into how the processes cooperate to satisfy the property. Previous methods to construct A automatically were based on the learning algorithm L^*, which represents A as a deterministic automaton and therefore has exponential worst-case complexity. Our new technique instead represents A as an equivalence relation on the states, which allows for a quasi-linear construction. The model checker can therefore apply compositional reasoning without risking an exponential penalty for computing A.

1 Introduction

Compositional model checking is a divide-and-conquer approach to verification that splits the correctness proof of a concurrent system into arguments over its individual processes. Compositional reasoning [12,4,11,15,20,23] is always advisable when one tries to analyze a complex program; for model checking, which automatically verifies a system by traversing its state space, compositionality is particularly helpful, because the number of states grows exponentially with the number of processes.

In order to check that a property P holds for the parallel composition $M \| N$ of two processes M and N, the compositional approach introduces an assumption A such that P holds for $M \| N$ if and only if P holds for $M \| A$. Because the assumption A is an abstraction of the implementation N, neglecting details not relevant for the property P, A can be much simpler than N. Recently,

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 143–158, 2006.

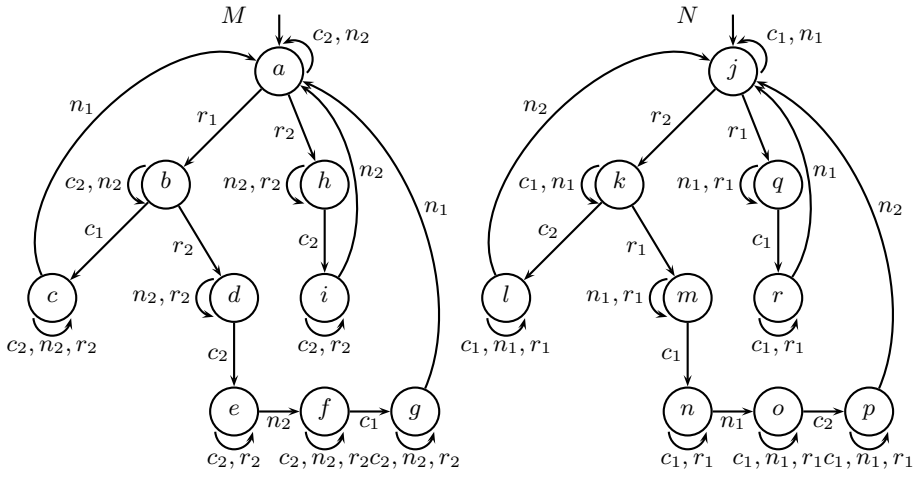


Fig. 1. Mutual exclusion protocol with two processes. Each process can request access to a critical resource (r_1, r_2), obtain the resource (c_1, c_2), and release it (n_1, n_2).

there has been a lot of interest in finding A automatically. There are at least three application scenarios for such a synthesis procedure. The first and most obvious scenario is to use A as program documentation, to be used during system optimization and maintenance: a modification to process N is safe as long as A is still valid. In a second scenario, the model checker provides A as a *certificate* (cf. [18]) for the validity of P: once A is known, *revalidating* the proof, possibly using a different model checker, is simple. The third and most ambitious scenario is to compute and use A during the *same* model checking run, accelerating the verification by compositional reasoning.

An interesting candidate for A is the *weakest* environment assumption under which process M guarantees P [8]. The weakest assumption is independent of N and therefore only needs to be computed once if M is used in different environments. However, because the weakest assumption must account for all possible environment behaviors, it usually has a large state space.

Several researchers have therefore investigated a different construction based on the L^* algorithm, a *learning* technique for deterministic automata [6,2,1]. In this setting, a candidate assumption A', represented as a deterministic automaton, is evaluated against both N and P by model checking. As long as either A' rejects some computation of N or $M \| A'$ accepts a computation that violates P, A' is refined to eliminate the particular counter example. The advantage of this approach is that it takes M into account and therefore produces assumptions that are much smaller than the weakest assumption. However, it is a less general technique: it will only yield an assumption if $M \| N$ actually satisfies P (and is therefore a compositional *proof* technique rather than a compositional *verification* technique). Furthermore, the structure of the deterministic automaton does not correspond to the structure of the (possibly nondeterministic) process N and is therefore usually not a good form of documentation. Also, learning (and

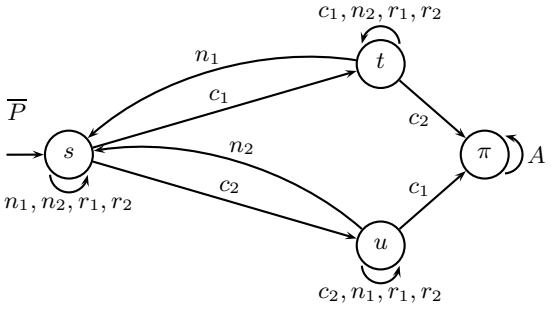


Fig. 2. Error LTS for the mutual exclusion property. Mutual exclusion forbids a second access to the critical resource by c_2 after a first access by c_1 has occurred that has not yet been released by n_1, and, symmetrically, an access by c_1 after an access by c_2 before the next n_2. In these cases, the property reaches the error state π.

storing) a deterministic automaton is expensive. Experience with the LTSA tool [6] suggests that the cost of computing the assumption in this way is several orders of magnitude higher than verifying the property by simple non-compositional model checking. In the worst case, the size of A (and therefore also the cost of the learning process) is exponential in the size of N.

In this paper, we argue that the synthesis of the assumption should not be significantly more expensive than solving the verification problem itself. We present a new approach to finding A, where, rather than synthesizing a deterministic automaton, we compute in linear time an *equivalence relation* $\sim$ on the states of N. The assumption A is the quotient of N with respect to $\sim$.

This reduction technique resembles the methods for process minimization used in compositional reachability analysis [21,22,10,3], which reduce a partially composed system to an observationally equivalent process. However, our equivalence relation is different: rather than preserving the entire observational behavior of a process, we only preserve the reachability of an error. Since this is a much coarser equivalence, the resulting quotient is much smaller.

Consider the mutual exclusion protocol in Figure 1. Each of the two processes can request access to a critical resource with the action r_1 (for process M) or r_2 (for process N), then obtain the resource with c_1 or c_2, and finally release the resource with n_1 or n_2. The protocol satisfies the mutual exclusion property, which forbids the c_2 action to occur after c_1 has happened and before the next n_1 has happened, and, symmetrically, the c_1 action to occur after a c_2 and before the next n_2. Mutual exclusion can be proven by *model checking*, i.e., by composing $M \| N$ with the error system for the mutual exclusion property, shown in Figure 2, and showing that the error state π is unreachable.

Compositional model checking considers the composition $M \| A$ instead of the full system $M \| N$. In our approach, the assumption A is the quotient of N with respect to an equivalence relation on the states of N that merges two states into a single equivalence class if they either both lead to an error in $M \| N$ or both

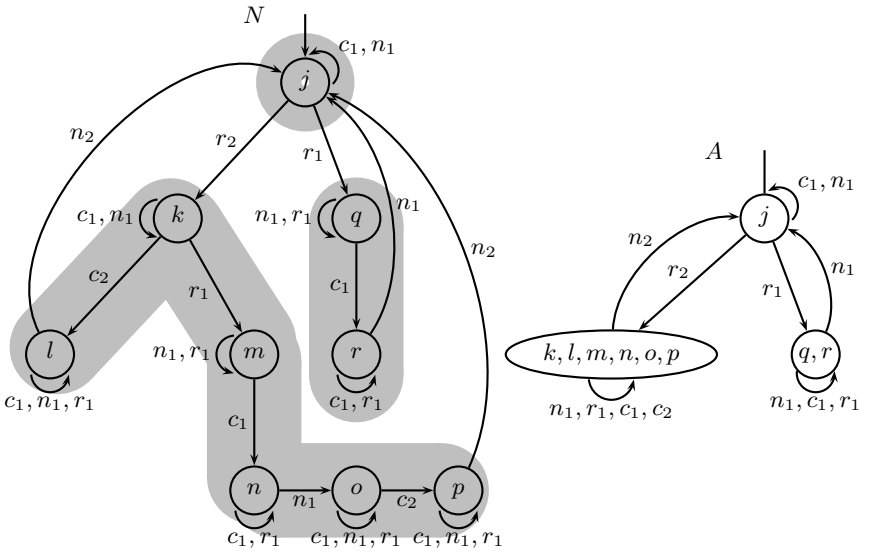


Fig. 3. Assumption $A = N/\sim$ for the compositional proof of mutual exclusion, defined by the equivalence relation $\sim$ on the states of LTS N. The equivalence classes of $\sim$ are shown in grey.

avoid the error in $M \| N$. Figure 3 shows the equivalence relation for the example. There are three equivalence classes: state j, states q, r, and states k, l, m, n, o, p. The quotient $A = N/\sim$ (where each equivalence class is a state, and there is an edge from one equivalence class to another if there is an edge from one of the members of the first class to a member of the second class) thus has only three states.

Like the weakest assumption, the quotient A can be used as an assumption both in proving *and* in disproving the property P. The full system $M \| N$ satisfies P if and only if the composition $M \| A$ satisfies P. Our algorithm for constructing the equivalence relation takes $O(|M| \cdot |N| \cdot \log |N| \cdot |P|)$ time, exceeding the cost of standard model checking only by a logarithmic factor. The generated assumption A is related to the process N by a simple homomorphism. Our construction is therefore a good solution for the first application scenario (documentation) as well as for the second scenario (certification).

Can we furthermore use the construction of A in the third scenario, to accelerate the model checking process by compositional reasoning? For this purpose, the complexity of the basic construction is too expensive. We give a modified construction that runs in $O(|\mathcal{M}| \cdot |N| \cdot \log |N| \cdot |P|)$ time, where $\mathcal{M}$ is an *abstraction* of M. The abstraction is computed in an automatic *abstraction refinement loop* that, starting with the trivial abstraction, incrementally increases the size of the abstraction. The loop can be interrupted after any number of iterations, yielding a sound (but not necessarily minimal) assumption. The algorithm terminates when the assumption cannot be reduced any further.

2 Labeled Transition Systems

We use *labeled transition systems* (LTS) to describe the behavior of processes. A labeled transition system $M = \langle V, E, v_0, A\rangle$ is given as a set V of states with a designated initial state $v_0 \in V$, a finite alphabet A of actions, and a set $E \subseteq V \times A \times V$ of edges.

A sequence $\overrightarrow{a} = a_1 a_2 a_3 \dots a_n \in A^*$ of actions in the alphabet of an LTS M is called a *run* of M if there is a sequence $\overrightarrow{v} = v_0 v_1 v_2 \dots v_n \in V^+$, starting in the initial state v_0, such that $(v_{i-1}, a_i, v_i) \in E$ is an edge of M for all $i = \{1, \dots, n\}$. $\overrightarrow{v}$ is called a *state trace* of $\overrightarrow{a}$. The set of runs of an LTS is called its *language*.

A system generally consists of multiple processes. The LTS of each process restricts the possible behavior of the system: a sequence $\overrightarrow{a}$ of actions is a run of the system iff it is a run of all processes.

Composition. The *composition* $M \| N$ of two LTS $M = \langle V_1, E_1, v_0^1, A\rangle$ and $N = \langle V_2, E_2, v_0^2, A\rangle$ is the LTS $\langle V, E, v_0, A\rangle$ with

- $V' = V_1 \times V_2$ and $v_0 = (v_0^1, v_0^2)$,
- $((v_1, v_2), a, (v_1', v_2')) \in E'$
  $\Leftrightarrow (v_1, a, v_1') \in E_1 \wedge (v_2, a, v_2') \in E_2$,
- $V \subseteq V'$ is the set of *reachable* states of V', and
- $E = E' \cap V \times A \times V$ is the set of reachable transitions.

Specification. An LTS $M = \langle V, E, v_0, A\rangle$ is called *deterministic* if, for all states $v \in V$ of M and all actions $a \in A$ of the alphabet of M at most one edge with label a exits ($|E \cap \{v\} \times \{a\} \times V| \leq 1$). A deterministic LTS P is called a *property*.

An LTS S *satisfies* P, denoted by $S \models P$, iff the language of S is contained in the language of P. For a (deterministic) property $P = \langle V, E, v_0, A\rangle$, the LTS $\overline{P} = \langle V \cup \{\pi\}, E_\pi, v_0, A\rangle$ with $E_\pi = E \cup \{\pi\} \times A \times \{\pi\} \cup \{(v, a, \pi) \mid v \in V, a \in A$ and $\{v\} \times \{a\} \times V \cap E = \emptyset\}$ is called the *error LTS* of P.

The error state π is treated specially in the composition $S \| \overline{P}$ of a process S and an error LTS. For $S = \langle V_1, E_1, v_0^1, A\rangle$ and $P = \langle V_2, E_2, v_0^2, A\rangle$, $S \| \overline{P}$ is the LTS $\langle V, E, v_0, A\rangle$ with

- $V' = (V_1 \times V_2) \cup \{\pi\}$ and $v_0 = (v_0^1, v_0^2)$,
- $((v_1, v_2), a, (v_1', v_2')) \in E'$
  $\Leftrightarrow (v_1, a, v_1') \in E_1 \wedge (v_2, a, v_2') \in E_2$,
- $(\pi, a, v) \in E' \Leftrightarrow v = \pi$,
- $((v_1, v_2), a, \pi) \in E' \Leftrightarrow \{v_1\} \times \{a\} \times V_1 \cap E_1 \neq \emptyset$ and
  $\{v_2\} \times \{a\} \times V_2 \cap E_2 = \emptyset$,
- $V \subseteq V'$ is the set of *reachable* states of V', and
- $E = E' \cap V \times A \times V$ is the set or reachable transitions.

Model checking. The verification problem is to decide for a given system S and a property P if $S \models P$. The verification problem can be solved by *model checking*, which checks if the error state π is reachable in the composition $S \| \overline{P}$. If $S = M \| N$ consists of two processes, the cost of model checking is in time and space $O(|M| \cdot |N| \cdot |P|)$.

Abstraction. Abstraction is a general verification technique, in which the behavior of a given process is approximated over a smaller state space. In this paper, we consider *homomorphic abstractions*, as introduced by Clarke, Grumberg, and Long [5]. An LTS $A = \langle V', E', v_0', A\rangle$ is a (homomorphic) *abstraction* of an LTS $N = \langle V, E, v_0, A\rangle$ if there exists a total and surjective function $h : A \to A'$, such that $h(v_0) = v_0'$, and for all edges (v, a, v') in E there is an edge $(h(v), a, h(v'))$ in E'.

In the following, we identify the homomorphism h with the induced equivalence $v \approx v' \equiv h(v) = h(v')$ on the states. The canonic abstraction defined by an equivalence relation $\approx$ is the quotient LTS with respect to $\approx$. We denote the equivalence class of a state n with respect to $\approx$ by $[n]_\approx$, or, if $\approx$ is clear from the context, by $[n]$. Let $V/\approx = \{[v] \mid v \in V\}$ denote the set of equivalence classes of a set V of states. The *quotient* of the LTS $N = \langle V, E, v_0, A\rangle$ with respect to $\approx$ is the LTS $N/\approx = \langle V/\approx, E', [v_0], A\rangle$, where $([v], a, [v']) \in E'$ iff there are two states $w \in [v]$ and $w' \in [v']$ such that $([v], a, [v']) \in E$.

Compositional verification. Our approach is based on the following compositional verification rule [19,2]:

$$\frac{\begin{array}{ll}(1) & M \| A \models P \\ (2) & N \models A\end{array}}{M \| N \models P}$$

To prove that a two-process system $M \| N$ satisfies a property P, the rule replaces, in premise (1), the process N by the *assumption* A, which, according to premise (2), must be chosen such that its language contains the language of N. In our setting, $A = N/\approx$ is the quotient of N with respect to an equivalence relation $\approx$ on the states of N. Since the language of an LTS is always contained in the language of its quotient, we obtain the following simplified rule:

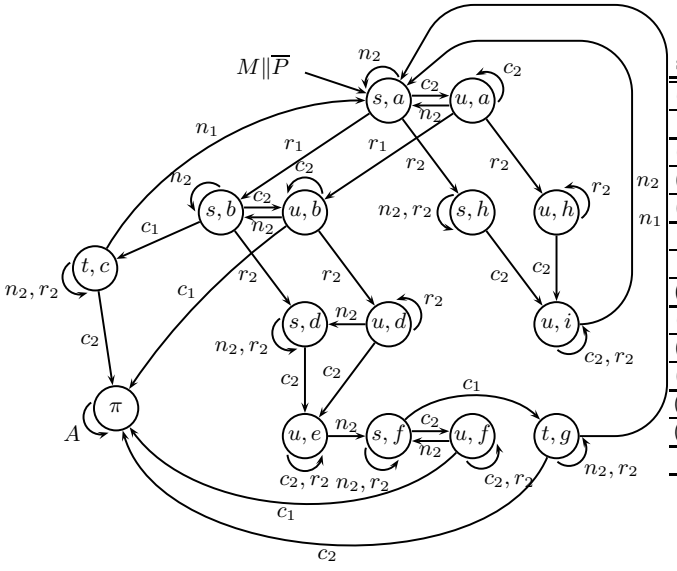
$$\frac{M \| N/\approx \models P}{M \| N \models P}$$

For an arbitrary equivalence relation $\approx$, the rule is sound but not necessarily invertible: the language of $M \| N$ may be a proper subset of the language of $M \| N/\approx$. In order to use the assumption both for proving and for disproving properties, we are interested in equivalences $\sim$ such that $M \| N/\sim \models P$ iff $M \| N \models P$. In the following sections, we present methods to construct such equivalences.

3 Forward Equivalence

We call two states n_1 and n_2 of N *forward-equivalent* if merging them does not make additional states in $M \| N \| \overline{P}$ reachable. For example, in Figure 3, the states m, n, o, and p are forward equivalent.

Let m_0, n_0, and p_0 be the initial states of M, N, and P, respectively. The *forward equivalence relation* $\sim_F$ is defined as follows.



state	V_F	V_B
(s,a)	$\{j\}$	$\{k,l,R\}$
(s,b)	$\{q\}$	$\{j,k,l,R\}$
(s,d)	$\emptyset$	$\{j,k,l,R\}$
(s,f)	$\emptyset$	$\{j,k,l,R\}$
(s,h)	$\{k\}$	$\emptyset$
(t,c)	$\{r\}$	$\{j,k,l,R\}$
(t,g)	$\emptyset$	$\{j,k,l,R\}$
(u,a)	$\emptyset$	$\{j,k,l,q,r,R\}$
(u,b)	$\emptyset$	$\{j,k,l,q,r,R\}$
(u,d)	$\emptyset$	$\{j,k,l,R\}$
(u,e)	$\emptyset$	$\{j,k,l,R\}$
(u,f)	$\emptyset$	$\{j,k,l,q,r,R\}$
(u,h)	$\emptyset$	$\emptyset$
(u,i)	$\{l\}$	$\emptyset$
π	$\emptyset$	$\{j,k,l,q,r,R\}$

Fig. 4. Labeling of $M\|\overline{P}$ in the computation of the forward and backward equivalences in the mutual exclusion example. The labeling with V_F, obtained during the forward traversal and shown in the second column, indicate that states m, n, o, and p of N are forward equivalent and can be merged into a single equivalence class R (cf. Figure 5). The labeling with V_B, obtained during the backward traversal and shown in the third column, indicates that states q and r, and states k, l and R of N_F are backward equivalent. Merging these states yields the assumption shown in Figure 3.

> Two states n_1 and n_2 of N are forward equivalent, $n_1 \sim_F n_2$, iff, for all states m of M and all states p of P, there is a path from $v_0 = (m_0, n_0, p_0)$ to the (m, n_1, p) if and only if there is a path from v_0 to (m, n_2, p).

The forward equivalence relation yields an invertible verification rule: $M\|N/\sim_F \models P$ iff $M\|N \models P$.

We compute $\sim_F$ in two steps. In the first step, we decorate the states of $Q = M\|\overline{P}$ with sets V_F of states of N such that the label of a state q of Q contains a state n of N iff there is a path from v_0 to (n, q) in $N\|Q$. In the second step, we extract the equivalence relation from the labels: for two states n_1 and n_2 of N, $n_1 \sim_F n_2$ iff for every label V_F on some state of Q, n_1 is in V_F if and only if n_2 is in V_F.

The labeling process is carried out as a fixed point computation, beginning with $\{n_0\}$ as the label on (m_0, p_0) and the empty set on all other states. If there is an edge with action a from a state (m, p) labeled with set V_F to a state (m', p'), then every state n of N that has an incoming edge with action a from some state in V_F is added to the label of (m', p'). By traversing the graph forward in a breadth-first manner, it suffices to consider each edge in $M\|\overline{P}$ at most once. The fixed point is therefore reached after at most $|M| \cdot |N| \cdot |P|$ steps. Let N_F be the quotient of N with respect to $\sim_F$.

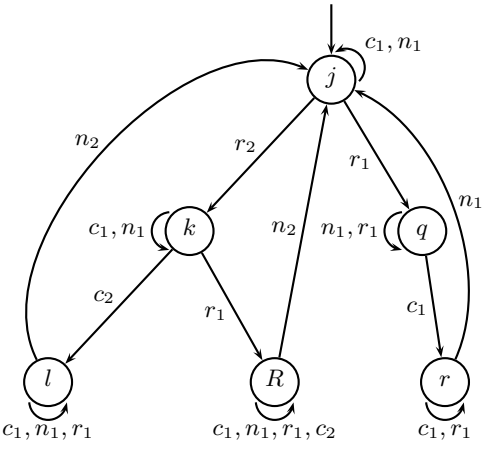


Fig. 5. The quotient N_F of process N for the compositional proof of mutual exclusion. States m, n, o, and p have been merged into the equivalence class R.

Figure 4 illustrates the computation of the forward equivalence on the states of process N from the mutual exclusion example. The second column shows the result V_F of the forward labeling process. States m, n, o and p are forward-equivalent, since they are not contained in the label of any state of $M\|\overline{P}$. Figure 5 shows the resulting quotient N_F.

A careful analysis shows that N_F can be constructed in $O(\log |N_F| \cdot |M| \cdot |N| \cdot |P|)$ time. We fix an arbitrary order $<_{M\|\overline{P}}$ on the states $V_{M\|\overline{P}}$ of $M\|\overline{P}$, and defer a linear pre-order $\preceq_N$ on the states V_N of N, such that two states $v, v' \in V_N$ are identified iff they are forward equivalent ($\simeq_N \equiv \sim_F$). Let $dec : V_{M\|\overline{P}} \to 2^{V_N}$ be the function that maps each state of $M\|\overline{P}$ to the set of states it is decorated with. We define $\prec_N = \{(v, v') \in {V_N}^2 \mid \exists w \in V_{M\|\overline{P}}.\, v \notin dec(w) \ni v' \wedge \forall w' <_{M\|\overline{P}} w.\, v \in dec(w) \leftrightarrow v' \in dec(w)\}$ and $\simeq_N = \{(v, v') \in {V_N}^2 \mid \forall w \in V_{M\|\overline{P}}.\, v \in dec(w) \leftrightarrow v' \in dec(w)\}$.

We can therefore construct the quotients by sorting the states of V_N with respect to $\preceq_N$, using AVL-trees. Concurrently to the sorting, we immediately merge equivalent states. The nodes of the AVL-tree is therefore bound by the number $|N_F|$ of quotients. Since comparing two elements of V_N can be performed in time $O(M)$, N_F can be constructed in time $O(\log |N_F| \cdot |M| \cdot |N| \cdot |P|)$.

4 Backward Equivalence

We call two states n_1 and n_2 of N_F (cf. Figure 5) *backward-equivalent* if merging them neither introduces nor removes an error path in $M\|N$. In the example of Figure 3, the states k, l, R and the states q, r are backward equivalent.

Two states n_1 and n_2 of N_F are *backward equivalent*, $n_1 \sim_B n_2$, iff, for all states m of M and all states p of P, there is a path from (m, n_1, p) to the error state π if and only if there is a path from (m, n_2, p) to π.

Combining the backward equivalence relation with the forward equivalence relation, we again obtain an invertible verification rule:

$$M \| N_F / \sim_B \models P \text{ iff } M \| N_F \models P \text{ iff } M \| N \models P.$$

The construction of $\sim_B$ is based on a labeling of the state graph of $Q = M \| \overline{P}$ with sets V_B of states of N_F such that the label of a state q of Q contains a state n of N_F iff there is a path from (n, q) to π in $N_F \| Q$. We extract the equivalence relation from the labels as follows: for two states n_1 and n_2 of N_F, $n_1 \sim n_2$ iff for every label V on some state of Q, n_1 is in V if and only if n_2 is in V.

The labeling process is carried out as a fixed point computation beginning with the entire state set of N as the label on the error state π and the empty set on all other states. If there is an edge with action a between a state (m, p) and a state (m', p') labeled with set V, then every state n of N that has an edge with action a to some state in V is added to the label of (m, p). By following the edges backwards from the error states in a breadth-first manner, it suffices to consider each edge in $M \| \overline{P}$ at most once. The fixed point is therefore again reached after at most $|M| \cdot |N| \cdot |P|$ steps. The assumption A is defined by the composition $\sim := \sim_B \circ \sim_F$ of the two equivalence relations: for two states n_1, n_2 of N, $n_1 \sim n_2$ iff $[n_1]_{\sim_F} \sim_B [n_2]_{\sim_F}$.

For the mutual exclusion example, the result V_B of the backward labeling is shown in the third column of the table in Figure 4. States k, l and R, and states q and r occur in the label of the same states of $M \| \overline{P}$. Consequently, they are backward-equivalent, and $\sim_B$ reduces the forward quotient N_F to the assumption LTS depicted in Figure 3.

5 Assumptions from Abstractions

Traversing the state space of $M \| \overline{P}$, as in the constructions of the previous sections, is not feasible if M is large, for example because it is again composed from multiple processes. In this section, we modify the algorithms to work on an *abstraction* of M. We assume that the abstraction is defined by a given equivalence relation $\approx$. This equivalence relation is used to construct a modal transition system, which in turn is used to compute upper and lower bounds for the labels V_F (or V_B) of the states of $M \| \overline{P}$. We present an algorithm for computing $\approx$ in Section 6.

Replacing M with an abstraction $\mathcal{M}$ introduces the possibility that two states of N both lead to an error when composed with $\mathcal{M}$, but only one of them leads to an error when composed with M. The algorithm must therefore distinguish situations that *may* lead to an error (i.e., the error is reached in the composition with $\mathcal{M}$ but not necessarily in M) from situations that *must* lead to an error (both in composition with $\mathcal{M}$ and in composition with M). Merging two states of N is safe in two cases: (1) if they *both must* lead to an error and (2) if *neither*

of them *may* lead to an error. We formalize this idea using modal transition systems. (The concept of modal transition systems has recently been successfully applied in model checking for single processes [7,14,9].)

A *modal transition system* (MTS) [17,16] is a tuple $\mathcal{M}=\langle V, E_{must}, E_{may}, v_0, A\rangle$ such that $\mathcal{M}_{must} = \langle V, E_{must}, v_0, A\rangle$ and $\mathcal{M}_{may} = \langle V, E_{may}, v_0, A\rangle$ are labeled transition systems and $E_{must} \subseteq E_{may}$.

An abstraction, given as an equivalence $\approx$ on the states of a labeled transition system $M = \langle V, E, v_0, A\rangle$, defines a modal transition system $\mathcal{M} = \langle V/\approx, E_{must}, E_{may}, [v_0], A\rangle$, where there is a *may* edge $([v], a, [v']) \in E_{may}$ iff *there is* a state $w \in [v]$ and a state $w' \in [v']$ such that $(w, a, w') \in E$.

An intuitive symmetric definition for the must edges E_{must}, which can be applied both for the computation of forward and backward equivalence classes, would be $E_{must}=\{([v], a, [v']) \in E_{may} \mid \forall w{\in}[v]\, \forall w'{\in}[v'].\, (w, a, w') \in E\}$. Stronger results can be obtained by using different sets of must edges for forward and backward analysis:

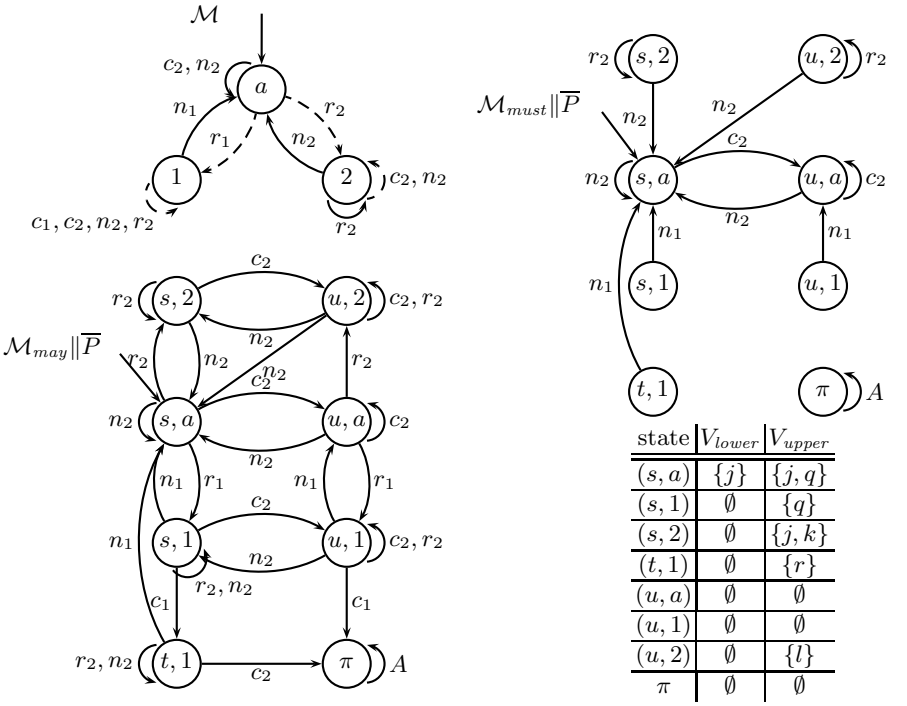
- For the computation of *forward* equivalence classes, an edge $([v], a, [v']) \in E_{must}$ is a must edge iff for all states $w' \in [v']$ there is a state $w \in [v]$ such that $(w, a, w') \in E$.
- For the computation of *backward* equivalence classes, an edge $([v], a, [v']) \in E_{must}$ is a must edge iff for all states $w \in [v]$ there is a state $w' \in [v']$ such that $(w, a, w') \in E$.

We extend the composition operator to modal transition systems. The *composition* $\mathcal{M}\|N$ of an MTS $\mathcal{M} = \langle V_1, E_1^{must}, E_1^{may}, v_0^1, A\rangle$ and an LTS $N = \langle V_2, E_2, v_0^2, A\rangle$ is constructed such that $(\mathcal{M}\|N)_{must} = \mathcal{M}_{must}\|N$ and $(\mathcal{M}\|N)_{may} = \mathcal{M}_{may}\|N$.

We construct the assumption A for the model checking problem $M\|N \models P$ again as an equivalence $\simeq := \simeq_B \circ \simeq_F$ on the states of N. Let $\mathcal{M}$ be the MTS defined by an abstraction of M, and let m_0, n_0, and p_0 be the initial states of $\mathcal{M}$, N, and P, respectively. The forward equivalence relation $\simeq_F$ is defined as follows: for two states n_1 and n_2 of N,

> $n_1 \simeq_F n_2$ iff for all states m of $\mathcal{M}$ and all states p of P, one of the following two conditions holds: (1) there is a path from (m_0, n_0, p_0) to (m, n_1, p) *and* a path from (m_0, n_0, p_0) to (m, n_2, p) in $\mathcal{M}_{must}\|N\|\overline{P}$, or (2) there is *no* path from (m_0, n_0, p_0) to (m, n_1, p) *and* there is *no* path from (m_0, n_0, p_0) to (m, n_2, p) in $\mathcal{M}_{may}\|N\|\overline{P}$.

To compute $\simeq_F$, we apply the fixed point construction from Section 3 twice: once on the graph $\mathcal{M}_{must}\|\overline{P}$, labeling each state with a subset V_{lower} of the states of N, and once on the graph $\mathcal{M}_{may}\|\overline{P}$, labeling each node with a subset V_{upper} of the states of N. If a state $[s]$ of $\mathcal{M}_{must}\|\overline{P}$ is labeled with (V_{lower}, V_{upper}) then all states $t \in [s]$ of $M\|\overline{P}$ are labeled with a *subset* $V_F \subseteq V_{upper}$ of V_{upper} (using the method suggested in Section 3). Likewise, if $\approx$ does not identify the initial state with any other state ($[(m_0, p_0)] = \{(m_0, p_0)\}$), all states $t \in [s]$ of $M\|\overline{P}$ are labeled with a *superset* $V_F \supseteq V_{lower}$ of V_{lower}. These upper and lower



state	V_{lower}	V_{upper}
(s, a)	$\{j\}$	$\{j, q\}$
$(s, 1)$	$\emptyset$	$\{q\}$
$(s, 2)$	$\emptyset$	$\{j, k\}$
$(t, 1)$	$\emptyset$	$\{r\}$
(u, a)	$\emptyset$	$\emptyset$
$(u, 1)$	$\emptyset$	$\emptyset$
$(u, 2)$	$\emptyset$	$\{l\}$
π	$\emptyset$	$\emptyset$

Fig. 6. Computation of the forward equivalence in the mutual exclusion example, based on an abstraction $\mathcal{M}$ of process M. The MTS $\mathcal{M}$ is the result of merging states b, c, d, e, f and g of M into equivalence class 1 and states h and i into equivalence class 2. States m, n, o, and p of N are forward-equivalent because they occur in none of the V_{upper} labels. Merging these states results in the quotient N_F shown in Figure 5.

bounds on the labeling of the single states of $M \| \overline{P}$ allow for the definition of an equivalence relation $\simeq_F$: For two states n_1 and n_2 of N, $n_1 \simeq_F n_2$ iff for every pair of labels V_{lower} and V_{upper} on some state, either n_1 is in V_{lower} and n_2 is in V_{lower}, or n_1 is in $V \setminus V_{upper}$ and n_2 is in $V \setminus V_{upper}$. Let N_F be the quotient of N with respect to $\simeq_F$.

Figure 6 illustrates the computation of $\simeq_F$ for the mutual exclusion example. The MTS $\mathcal{M}$ is the result of merging states b, c, d, e, f and g of M into equivalence class 1 and merging states h and i into equivalence class 2. States m, n, o, and p of N are forward-equivalent because they occur in none of the V_{upper} labels. Merging these states results in the quotient N_F shown in Figure 5.

To compute N_F, we proceed in two steps. In a first step, we compute those states $V_N^1 \subseteq V_N$ of N, which are in V_{upper} but not in V_{lower} for some state of $\mathcal{M} \| \overline{P}$. Since these states always form a quotient of their own, they can be excluded from further consideration. The construction of N_F is then completed by construction quotients for the states in $V_N \setminus V_N^1$ using the sorting approach suggested in the previous section. The overall construction again takes $O(\log |N_F| \cdot |M| \cdot |N| \cdot |P|)$ time.

The backward equivalence relation $\simeq_B$ can be defined and computed analog to the forward equivalence relation $\simeq_F$. Since the equivalences $\sim_F$ and $\sim_B$ obtained without abstraction (by the algorithms in Sections 3 and 4) are always coarser than the equivalences $\simeq_F$ and $\simeq_B$ obtained using $\mathcal{M}$, we again obtain invertible proof rules:

$$M \| N/\simeq_F \models P \text{ iff } M \| N \models P, \text{ and}$$
$$M \| N_F/\simeq_B \models P \text{ iff } M \| N_F \models P.$$

6 Abstraction Refinement

In this section, we give a construction for the equivalence $\approx$ on the states of M needed in the algorithms in Section 5. We begin with the trivial two-state abstraction (that merges all non-initial states) and then incrementally increase the size of the abstraction in an abstraction refinement loop.

Since the constructions in Section 5 produce some (not necessarily minimal) assumption for any abstraction, the loop can be interrupted at any time. Otherwise, the loop terminates as soon as the upper and lower bounds (V_{lower}, V_{upper}) coincide for all states of $M \| \overline{P}$.

As long as there is some state labeled with (V_{lower}, V_{upper}) such that $V_{lower} \neq V_{upper}$, we pick a *may* edge (s, a, s') of $\mathcal{M}_{may} \| \overline{P}$ that does not occur in $\mathcal{M}_{must} \| \overline{P}$.

To obtain a coarser forward equivalence relation $\simeq_F$, we refine $\approx$ by distinguishing any two states m_1 and m_2 represented by s' ($m_1', m_2' \in [s']_M$, where $[s]_M = [m]$ for $s = ([m], p)$ and $[\pi]_M = V_M$) if there is an edge (m_1, a, m_1') in M with $m_1 \in [s]_M$, but no edge (m_2, a, m_2') with $m_2 \in [s]_M$. I.e., the equivalence relation $\approx$ is refined into the new equivalence $\approx_{(s,a,s')}$, with

$$\approx_{(s,a,s')} = \approx \setminus \{(m_1, m_2) \in [s']_M^2 \mid (\exists m \in [s]_M . (m, a, m_1) \in E_M) \nleftrightarrow (\exists m \in [s]_M . (m, a, m_2) \in E_M)\}.$$

Note that the previously computed upper and lower bounds remain valid after the refinement of $\approx$. We preserve and use this information: The previous values of V_{lower} can be used as a starting point for the fixed point construction of the new V_{lower}. Since a split can introduce new may edges, this method does not only accelerate the computation of the fixed point, but also provides sharper lower bounds. The refinement loop is guaranteed to terminate: in the worst case, the number of refinement steps is equal to the size of M. How fast the loop terminates depends on the choice of the *may* edges to refine on.

We avoid the explicit computation of the upper bounds V_{upper} by choosing an edge (s, a, s') such that

- s and s' were labeled during the forward traversal with V_{lower} and V'_{lower}, respectively, and
- $V_{lower} \times \{a\} \times V_N \setminus V'_{lower}$ is not disjoint from the edges of N.

The second condition avoids the choice of edges that cause no difference in the labeling of s'. If there are multiple such edges, we pick one where the distance from the initial state to s is minimal in $\mathcal{M}_{must} \| \overline{P}$.

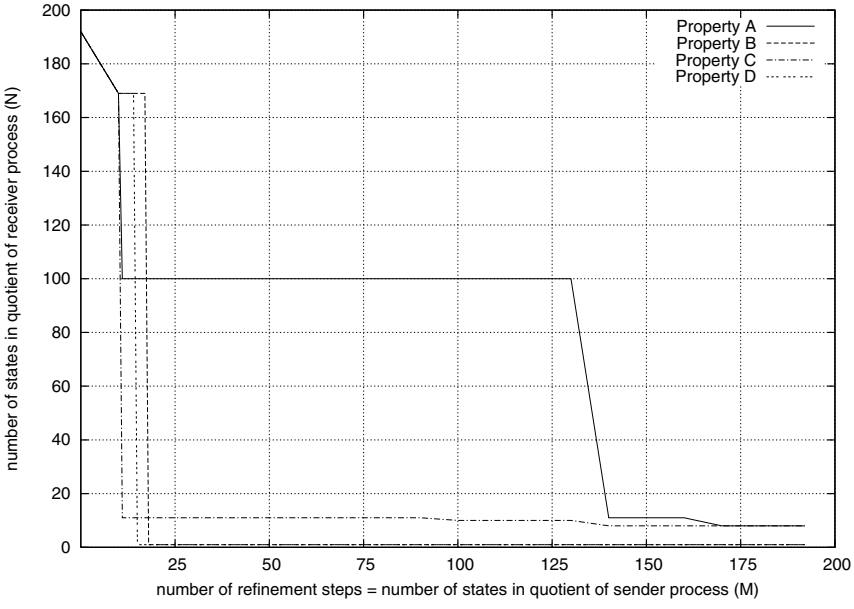


Fig. 7. Experimental data from the *sliding window protocol* benchmark. Property A expresses that the protocol does not invent messages, Property B and C express that the *sender* process and the receiver process, respectively, do not invent messages, and Property D expresses (incorrectly) that no messages are delivered. The figure shows the number of states in the quotient of the *receiver* process (N) after a given number of refinement steps. Since each refinement step introduces a new state in the abstraction of the *sender* process (M), the number of refinement steps is equal to the number of states in the quotient of M.

The refinement step for the backward equivalence $\simeq_B$ can be defined analogously.

7 Experimental Results

We have implemented the algorithms of this paper in a small prototype tool, which is intended as a front-end to the model checker SPIN [13]. Our tool reads a two-process system written in (a subset of) Promela and produces a modified system, where the second process is replaced by the assumption LTS. The tool applies the abstraction refinement algorithm and switches every ten steps between computing the forward and computing the backward equivalence. The process can be interrupted after an arbitrary number of refinement steps and terminates once the upper and lower bounds for the labels coincide in both constructions.

Figure 7 shows experimental data from the verification of the classic *sliding window protocol* benchmark. In the sliding window protocol, the *sender* (process M) transmits messages over an unreliable channel to the *receiver* (process N). To

ensure that no packets are lost, the sender stores the messages in a sliding buffer until acknowledgments are received. In our benchmark, there are three different types of messages (*red*, *white*, and *blue*) and the buffer stores two messages at a time, which results in 192 states each for the *sender* and the *receiver*.

We consider four properties: Property A expresses that the protocol does not invent messages ("if there is no *white* message in the input of the *sender*, then there will be no *white* message in the output of the receiver"). Properties B and C express the same condition locally for the two processes, i.e., Property B specifies that the *sender* does not invent messages ("if there is no *white* message in the input of the *sender*, then there will be no *white* message on the network"), Property C specifies that the *receiver* does not invent messages ("if there is no *white* message on the network, then there will be no *white* message in the output"). Property D expresses that the receiver never produces any output. While Properties A, B, and C are satisfied by the sliding window protocol, Property D is violated.

The refinement process terminates after 169 steps for Property A, 18 steps for Property B, 168 steps for Property C, and 15 steps for Property D. The resulting assumption has 8 states for Property A, 1 state for Property B, 8 states for Property C, and 1 state for Property D. Not surprisingly, replacing the *receiver* process with these assumptions reduces the model checking time (Property A: 4s instead of 19s, Property B: 3s instead of 19s, Property C: 4s instead of 18s, Property D: 3s instead of 18s, on an Athlon XP 2600+ with 2GB RAM).

If the purpose of computing the assumption is to improve the time and memory performance of a *single* model checking run, it appears to be beneficial to interrupt the refinement process early. Figure 7 shows the number of states in the quotient of N that is reached when the refinement process is interrupted after a certain number of steps. A significant drop in the number of states in the assumption occurs already very early on, when only a small percentage of the states of M have been considered.

8 Conclusions and Future Work

Compositionality and abstraction are generally considered the two key methods in avoiding the state-space explosion problem. The combination of the two methods in our assumption synthesis algorithm adds a new twist to classic abstraction refinement: rather than starting with a coarse abstraction of process N, which would need to be corrected through a successive elimination of spurious counter examples, we start with an abstraction of its environment (M), which always (at any point in the refinement cycle) allows us to produce an assumption that is free of spurious counter examples.

Our approach has several advantages. First, and perhaps most important, the resulting assumption is *acceptance preserving*. The result of model checking is the same if we use the assumption or the original process. Second, while using the assumption may significantly accelerate the model checking, there is

no penalty in the form of increased complexity as introduced by the intermediate state explosion problem [10,8] or by using deterministic automata [6,2,1,8]. In the worst case, the generated assumption is as large as the process itself. Even this, however, is unlikely to occur for well-designed software architectures.

A third advantage of our approach is that the generated assumptions have applications beyond classic model checking. They are well-suited as certificates. Using an arbitrary assumption automaton A for N, the language containment check is PSPACE-hard in the size of N and EXPSPACE-hard in the size of A. Since our method generates a homomorphic abstraction of N, language containment can be checked in linear time. For similar reasons, the generated abstraction is useful both in the documentation of a process and in the maintenance phase.

In future work, we intend to expand on our prototype tool implementation. In particular, the application to larger systems needs good heuristics for the refinement of the modal LTSs. An interesting open question is the extension of our method to obtain assumptions for more than one process. It is always possible to replace one process after another by a homomorphic abstraction, but more experience is needed to determine the sequence in which the processes should be considered and to decide whether it is worthwile to alternate between the processes during the refinement cycle.

References

1. Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proc. POPL*, pages 98–109, New York, NY, USA, 2005. ACM Press.
2. Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *Proc. CAV*, volume 3576 of *LNCS*, pages 548–562, 2005.
3. Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.
4. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proc. LICS*, pages 353–362, 1989.
5. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In 19^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 343–354, 1992.
6. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. TACAS*, pages 331–346, 2003.
7. Luca de Alfaro, Patrice Godefroid, and Radha Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proc. LICS*, pages 170–179, 2004.
8. Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Proc. ASE*, pages 3–12, Washington, DC, USA, 2002. IEEE Computer Society.
9. Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. CONCUR*, pages 426–440. Springer-Verlag, 2001.

10. Susanne Graf, B. Steffen, and G. Lüttgen. Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computation*, 8, September 1996.
11. Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
12. Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. CAV*, pages 440–451, 1998.
13. G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
14. Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proc. European Symposium on Programming*, pages 155–169. Springer-Verlag, 2001.
15. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
16. Kim G. Larsen. Modal specifications. In *Proc. Automatic Verification Methods for Finite State Systems*. Springer-Verlag, 1989.
17. Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proc. LICS*, pages 203–210. IEEE Computer Society Press, 1988.
18. Kedar S. Namjoshi. Certifying model checkers. In *Proc. CAV*, pages 2–13. Springer-Verlag, 2001.
19. Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In *Proc. CAV*, pages 139–153. Springer-Verlag, 2000.
20. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer-Verlag, 1985.
21. Krishan. K. Sabnani, Aleta M. Lapone, and M. Ümit Uyar. An algorithmic procedure for checking safety properties of protocols. *IEEE Trans. Commun.*, 37(9):940–948, September 1989.
22. Kuo-Chung Tai and Pramod V. Koppol. An incremental approach to reachability analysis of distributed programs. In *Proc. IWSSD*, pages 141–150, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
23. Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.