Cluster-based partial-order reduction


Automated Software Engineering, 11, 365–402, 2004

Cluster-Based Partial-Order Reduction

TWAN BASTEN a.a.basten@tue.nl
Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

DRAGAN BOŠNAČKI d.bosnacki@tue.nl
Department of Biomedical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

MARC GEILEN m.c.w.geilen@tue.nl
Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Abstract. The verification of concurrent systems through an exhaustive traversal of the state space suffers from the infamous state-space-explosion problem, caused by the many interleavings of actions of different processes in the system. Partial-order reduction is a well-known technique to tackle this problem. In this paper, we present an enhancement of the partial-order-reduction scheme of Holzmann and Peled that uses the hierarchical structure of concurrent systems. Our technique tries to contain dependencies between actions within clusters of processes, capitalizing on the independence of actions in different clusters to reduce the state space to be verified while preserving properties of interest. The paper starts with a formalization of the partial-order-reduction technique and continues with a presentation of our enhanced technique, including a correctness argument. The new technique has been implemented in the verification tool SPIN. We present implementation details, some small experiments, and one larger case study using a cache coherency protocol. The experimental results are encouraging. Compared to standard partial-order reduction, improvements in reductions are obtained from 21% up to 98% in the number of states and 34% up to 99% in the number of state transitions.

Keywords: concurrency, state explosion, formal verification, partial-order reduction, (LTL) model checking, SPIN

1. Introduction

Over the last decades, the complexity of computer systems has been increasing rapidly, with a tendency towards distribution and concurrency. The correct functioning of these complex systems is becoming an ever larger problem. Many verification and proof techniques have been invented to solve this problem. An important class of techniques are those based on a fully automatic, exhaustive traversal of the state space of a concurrent system. Well-known representatives are the various model-checking techniques.

An infamous problem complicating an exhaustive traversal of the state space of a concurrent system is the state-space explosion, caused by the arbitrary interleaving of independent actions of the various components of the system. Several techniques have been developed to address this problem. *Partial-order reduction* is a very prominent one (see, for example, Alur et al., 1997; Godefroid, 1996; Godefroid and Wolper, 1991; Holzmann et al., 1992;

Holzmann and Peled, 1995; Nalumasu and Gopalakrishan, 2002; Overman, 1981; Peled, 1994; Valmari, 1991, 1992; Willems and Wolper, 1996). It exploits the independence of actions to reduce the state space of a system while preserving properties of interest. When generating a state space, in each state, when possible, only a subset of the enabled actions satisfying certain criteria is chosen for further exploration. Following Holzmann and Peled (1995) and Peled (1994), we call these sets *ample* sets. Alternative terms appearing in the literature are stubborn sets, sleep sets, or persistent sets.

The traditional approach to partial-order reduction (see, for example, Holzmann and Peled (1995) and Peled (1994)) deals with systems seen as an unstructured collection of sequential processes running in parallel. However, many systems have inherent hierarchical structure, imposed either explicitly by the language used for the system specification that groups processes in blocks or similar constructs (e.g., SDL, UML), or implicitly by the interconnections among processes. In this paper, we present a partial-order-reduction algorithm that exploits the hierarchical structure of a system. The current paper builds on Basten and Bošnački (2001). The described partial-order-reduction technique is a slight generalization of the technique as it is explained in Basten and Bošnački (2001). Furthermore, the current paper presents the implementation in SPIN in more detail, and it presents an extra, more realistic case study testing our technique.

Our starting point is the partial-order algorithm of Holzmann and Peled (1995) and Peled (1994). This algorithm is implemented in the verification tool SPIN (Holzmann, 1991, 1997; SPIN, 2004) and has proven to be successful to some extent in coping with the state-space explosion. It is also sufficiently flexible to allow extensions (see for instance the extensions for timed systems in Bošnački and Dams (1998) and Minea (1999)). The algorithm uses a notion of *safety* to select ample sets. The safety requirement is imposed via syntactical criteria to avoid expensive computations during the state-space traversal. These criteria give ample sets containing either all enabled actions of a single process or all enabled actions of all processes.

It is our idea to introduce a gradation of the safety requirement based on the hierarchical structure of a system. To this end, we introduce the concept of a cluster hierarchy to capture the system hierarchy and the induced (in)dependencies among processes. This generalization allows ample sets consisting of actions from different, but not necessarily all, processes. Our cluster-based algorithm is a true generalization of the partial-order-reduction algorithm of Holzmann and Peled (1995) and Peled (1994). It can also be seen as a version of the algorithm of Overman (1981), adapted for cluster hierarchies and LTL model checking. We implemented our algorithm in the verification tool SPIN. The results obtained with the prototype are encouraging. The enhancement of partial-order reduction via process clustering fits with current trends in software engineering. The tendency is to develop visual specification languages that express both system structure and behavior (UML, SDL). The resulting hierarchical structure can be exploited in verification.

The remainder of this paper is organized as follows. Section 2 explains the state-space-explosion problem and the basic concepts that play a role in this paper. Section 3 presents some known theoretical results on partial-order reduction as well as the reduction algorithm of Holzmann and Peled (1995) and Peled (1994). In Section 4, we present our cluster-based partial-order-reduction algorithm. Section 5 gives some experimental results on small examples. Section 6 discusses a more-or-less realistic verification case concerning cache

coherency for a networks-on-silicon-based system-on-chip architecture. Section 7 contains concluding remarks. Finally, the appendix contains some code fragments of our experiments.

2. Preliminaries

State spaces of concurrent systems. Concurrent systems typically consist of a number of processes running in parallel. To exchange information, these processes communicate via messages and/or shared memory. The left part of figure 1 shows the very simple concurrent system `example0`. It consists of three processes, `P0`, `P1`, and `P2`, each one executing a sequence of two actions. Assuming that there is no communication and, thus, all actions can be executed independently, the right part of figure 1 shows the state space of system `example0`.

The numbers 000 through 222 represent states of system `example0`. The states encode all relevant information of `example0` such as values of variables and local program counters. The initial state of the system is 000 (marked with a small arrow). The labeled arrows in figure 1 correspond to state changes or *transitions* of the system. The labels link transitions to actions of the system. Note that parallel arrows in figure 1 are assumed to have identical action labels.

The example of figure 1 illustrates a well-known problem complicating the verification of concurrent systems. Clearly, each of the processes in `example0` can only be in three different states. However, figure 1 shows that the complete state space of `example0` consists of 27 ($=3^3$) states. The state space of a system with a fourth process (executing two actions) consists of 81 ($=3^4$) states, whereas the state space of a system consisting of two processes consists of only 9 ($=3^2$) states. This example illustrates that the state space of a concurrent system may grow exponentially if the number of processes in the system increases. This phenomenon is known as the state-space explosion. For realistic concurrent systems, the state-space explosion renders infeasible verification techniques that are based on an exhaustive traversal of the state space.

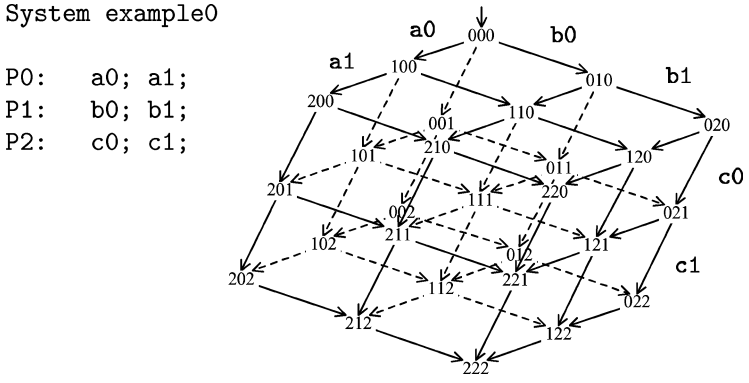

Figure 1. A simple concurrent system.

Labeled transition systems. The notion of a state space is an important concept in this paper. To formally reason about state spaces, we introduce the notion of a *labeled transition system*.

Definition 2.1 (Labeled transition system). A labeled transition system, or simply an LTS, is a 6-tuple $(S, \hat{s}, A, \tau, \Pi, L)$, where

- S is a finite set of *states*;
- $\hat{s} \in S$ is the *initial* state;
- A is a finite set of *actions*;
- $\tau : S \times A \to S$ is a (partial) *transition function*;
- Π is a finite set of boolean *propositions*;
- $L : S \to 2^{\Pi}$ is a *state labeling function*.

The first four elements in the definition of an LTS have already been discussed in the previous paragraph. Note that our definition of an LTS requires that state transitions are deterministic. This restriction is not really fundamental but it turns out that it is convenient in formalizing the partial-order-reduction framework in the remainder. Propositions and state labels are not standard in definitions of LTSs but they are well known from the model of Kripke structures. They are included in our definition of LTSs because they play a role when one is interested in verifying specific properties of a concurrent system. Before explaining these last two elements in some more detail, we introduce some auxiliary notions. Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be some LTS.

An action $a \in A$ is said to be *enabled* in a state $s \in S$, denoted $s \xrightarrow{a}$, if and only if $\tau(s, a)$ is defined. The set of all actions enabled in state s is denoted *enabled*(s): *enabled*$(s) = \{a \in A \mid s \xrightarrow{a}\}$. State s is a *deadlock* state if and only if *enabled*$(s) = \emptyset$.

The notion of a *transition* has already been mentioned. Formally, transition function τ of LTS $\mathcal{T}$ induces a set $T \subseteq S \times A \times S$ of transitions defined as $T = \{(s, a, s') \mid s, s' \in S \wedge a \in A \wedge s' = \tau(s, a)\}$. To improve readability, we write $s \xrightarrow{a} s'$ for $(s, a, s') \in T$. Besides the number of states of a concurrent system, also the number of transitions of the system is a factor complicating verification. The LTS of figure 1 has 54 transitions.

In the following, variable i ranges over the natural numbers $\mathbb{N}$. An *execution sequence* of LTS $\mathcal{T}$ is a (finite or infinite) sequence of subsequent transitions in T. Formally, for any natural number $n \in \mathbb{N}$, states $s_i \in S$ with $0 \leq i \leq n$, and actions $a_i \in A$ with $0 \leq i < n$, the sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \; s_{n-1} \xrightarrow{a_{n-1}} s_n$ is an execution sequence of length n of $\mathcal{T}$ if and only if $s_i \xrightarrow{a_i} s_{i+1}$ for all i with $0 \leq i < n$. State s_n is said to be *reachable* from s_0. A state is reachable in $\mathcal{T}$ if and only if it is reachable from $\hat{s}$. For states $s_i \in S$ and actions $a_i \in A$, the sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ is an infinite execution sequence of $\mathcal{T}$. The LTS of figure 1 has 90 execution sequences of length six all starting from the initial state and leading to deadlock state 222.

Properties of concurrent systems. There are many different kinds of properties of concurrent systems that designers are interested in. We mention three well-known classes of properties. For each of these classes, verification techniques exist that are based on an exhaustive traversal of the state space of a concurrent system. Hence, the state-space reduction technique presented in this paper can be helpful to extend the applicability of these verification

techniques to realistic systems. In the remainder, neither the details of the specification of properties nor the details of the verification techniques are very important. Hence, we only give an informal explanation.

The first class of properties is the presence or absence of deadlocks. It is clear that deadlock properties can be verified in a straightforward way by means of an exhaustive state-space traversal.

The second class of properties are the so-called *local* properties. Local properties of a concurrent system are properties that typically depend only on the state of a single (sequential) process of the system or on the state of a single shared object. The question whether or not a state satisfying some local property is reachable can also be verified by means of a state-space traversal. At this point, the reason for including a set of boolean propositions and an accompanying state labeling in the definition of an LTS becomes apparent. Local-property verification via a state-space traversal is convenient when all relevant information concerning the property is encoded in the state labeling of the LTS representing the state space of the system under investigation. The idea is that the label associated with a state contains precisely all propositions that are true in that state. For more details on the verification of local properties (see Godefroid, 1996; Holzmann et al., 1992; Valmari, 1991).

The third class of properties are those expressible in (next-time-free) Linear-time Temporal Logic (LTL). LTL properties are also formulated in terms of the propositions in an LTS. It is beyond the scope of this paper to give a formal definition of LTL; the interested reader is referred to Manna and Pnueli (1991). In this paper, we restrict ourselves to so-called next-time-free LTL. (In the context of concurrent systems, the next-time operator of LTL is not very meaningful.) A next-time-free LTL formula may contain only boolean propositions, the boolean operators $\wedge$ (and), $\vee$ (or), and $\neg$ (negation), and the temporal operators $\Box$ (always), $\Diamond$ (eventually) and U (until). The technique for verifying LTL formulae is referred to as (LTL) model checking. Again, the details are not important. For more information (see for example, Holzmann and Peled, 1995).

3. Partial-order reduction

Our techniques build upon the existing basis of partial-order reduction. Section 3.1 gives some theoretical results that are needed to prove that our reduction technique preserves deadlocks, local properties, and next-time-free LTL. These results are known from the literature (see, for example, Alur et al., 1997; Godefroid, 1996; Godefroid and Wolper, 1991; Holzmann et al., 1992; Holzmann and Peled, 1995; Nalumasu and Gopalakrishan, 2002; Overman, 1981; Peled, 1994; Valmari, 1991, 1992; Willems and Wolper, 1996). Section 3.2 presents the partial-order-reduction algorithm of Holzmann and Peled (1995) and Peled (1994). In Section 3.3, we briefly discuss implementation issues.

3.1. Basic theoretical framework

The basic idea of state-space reduction techniques for enhancing verification is to restrict the part of the state space of a concurrent system that is explored during verification in such a way that properties of interest are preserved. There are several types of reduction techniques.

In this paper, we focus on partial-order reduction, which exploits the independence of properties from the possible interleavings of the actions of the concurrent system. It uses the fact that the state-space explosion is often caused by the interleaving of independent actions of concurrently executing processes of the system (see figure 1).

To be practically useful, a reduction of the state space of a concurrent system must be achieved during the traversal of the state space. Thus, it must be decided *per state* which transitions, and hence which subsequent states, must be considered. Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be some LTS.

Definition 3.1 (Reduction). For any so-called *reduction* function $r : S \to 2^A$, we define the (partial-order) *reduction* of $\mathcal{T}$ with respect to r as the smallest LTS $\mathcal{T}_r = (S_r, \hat{s}_r, A, \tau_r, \Pi, L_r)$ satisfying the following conditions:

- $S_r \subseteq S$, $\hat{s}_r = \hat{s}$, $\tau_r \subseteq \tau$, and $L_r = L \cap (S_r \times 2^\Pi)$;
- for every $s \in S_r$ and $a \in r(s)$ such that $\tau(s, a)$ is defined, $\tau_r(s, a)$ is defined.

Note that these requirements imply that, for every $s \in S_r$ and $a \in A$, if $\tau_r(s, a)$ is defined, then also $\tau(s, a)$ is defined and $\tau_r(s, a) = \tau(s, a)$.

Figure 2 shows a reduction function for system `example0` of figure 1 and the corresponding reduced state space.

It may be clear that not all reductions preserve all properties of interest. Thus, depending on the properties that a reduction r must preserve, we have to define additional restrictions on r. To this end, we need to formally capture the notion of independence introduced earlier. Actions occurring in different processes may still influence each other, for example, when they access global variables. The following notion of independence defines the absence of such mutual influence. Intuitively, two actions are independent if and only if, in every state where they are both enabled, (1) the execution of one action cannot disable the other and (2) the result of executing both actions is always the same.

Definition 3.2 (Independence). Actions $a, b \in A$ with $a \neq b$ are *independent* if and only if, for all states $s \in S$ with $s \xrightarrow{a}$ and $s \xrightarrow{b}$ (i.e., $\tau(s, a)$ and $\tau(s, b)$ are both defined),

$$r(s) = \begin{cases} \{\texttt{a0}\} & \text{if } s = 000 \\ \{\texttt{a1}\} & \text{if } s = 100 \\ \{\texttt{b0}\} & \text{if } s = 200 \\ \{\texttt{b1}\} & \text{if } s = 210 \\ \{\texttt{c0}\} & \text{if } s = 220 \\ \{\texttt{c1}\} & \text{if } s = 221 \\ \mathit{enabled}(s) & \text{otherwise} \end{cases}$$

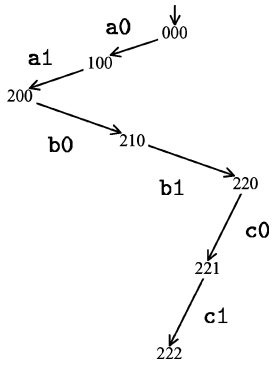

Figure 2. A reduced state space of system `example0`.

– $\tau(s, a) \xrightarrow{b}$ and $\tau(s, b) \xrightarrow{a}$, and
– $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$.

Actions that are not independent are called dependent.

An example of independent actions are two assignments to or readings from local variables in distinct processes. Note that two actions are trivially independent if there is no state in which they are both enabled. It is straightforward to see that, in our running example of figure 1, all actions are mutually independent.

The first class of properties we are interested in is the presence or absence of deadlocks. To preserve deadlock states of an LTS in a reduced LTS, the reduction function r must satisfy the following two conditions (called provisos):

C0 $r(s) = \emptyset$ if and only if $\mathit{enabled}(s) = \emptyset$.

C1 (Persistency). For any $s \in S$ and execution sequence $s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ of length $n \in \mathbb{N} \setminus \{0\}$ with $a_i \notin r(s)$ for all i $(0 \leq i < n)$, action a_{n-1} is independent of all actions in $r(s)$.

The basic idea behind the persistency proviso is that, during the state-space traversal, transitions caused by actions that are independent of all the actions chosen by the reduction function can be ignored.

Theorem 3.3 (*Deadlocks* (Godefroid, 1996, *Theorem* 4.3))**.** *Let r be a reduction function for LTS $\mathcal{T}$ that satisfies provisos C0 and C1. Any deadlock state reachable in $\mathcal{T}$ is also reachable in the reduced LTS $\mathcal{T}_r$ and vice versa.*

A few remarks are in order. First, Theorem 4.3 in Godefroid (1996) does not state that any deadlock reachable in a reduced LTS is also reachable in the original LTS. However, this result follows directly from proviso C0. Second, Godefroid (1996) uses a slightly stronger definition of independence. However, the proof of Theorem 4.3 in Godefroid (1996) carries over to our setting without change. Finally, several authors presented state-space-reduction algorithms that preserve deadlocks (Godefroid and Wolper, 1991; Overman, 1981; Valmari, 1991).

Let us return to figure 2. It is not difficult to verify that the given reduction function satisfies provisos C0 and C1. (Recall from above that all actions of `example0` are mutually independent.) Clearly, this reduction preserves deadlock state 222.

The second class of properties we discuss are the local properties. A local property is a boolean combination of propositions in Π whose truth value cannot be changed by two independent actions.

Definition 3.4 (Local property). A property ϕ built from propositions and boolean constants and operators is *local* if and only if for all states $s \in S$ and *enabled independent* actions $a, b \in A$ the following holds: if ϕ holds (does not hold) in state s and ϕ does not hold (holds) in state $\tau(s, a)$, then ϕ holds (does not hold) in $\tau(s, b)$ and does not hold (holds) in $\tau(\tau(s, b), a)$.

An LTS satisfies a local property ϕ if and only if there is a reachable state that satisfies ϕ. Typical examples of local properties are properties that depend only on the state of a single (sequential) process or shared object. To guarantee that a reduction of a state space preserves local properties, it suffices that the reduction function r satisfies the following requirement (in addition to C0 and C1).

C2 (Cycle proviso). For any *cycle* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n = s_0$ of length $n \in \mathbb{N} \setminus \{0\}$ with $a_i \in r(s_i)$ for all $i \in \mathbb{N}$ with $0 \leq i < n$, there is a $j \in \mathbb{N}$ with $0 \leq j < n$ such that $r(s_j) = \mathit{enabled}(s_j)$.

Theorem 3.5 (*Local properties*)**.** *Let r be a reduction function for LTS $\mathcal{T}$ satisfying provisos C0, C1, and C2; let ϕ be a local property. LTS $\mathcal{T}$ satisfies ϕ if and only if the reduced LTS $\mathcal{T}_r$ satisfies ϕ.*

Proviso C2 prevents the so-called 'ignoring problem' identified in Valmari (1991). Informally, this problem occurs when a reduction of a state space ignores the actions of an entire process. Proofs of (variants of) Theorem 3.5 can be found in Godefroid (1996), Holzmann et al. (1992) and Valmari (1991). In fact, these references show that C2 can be weakened if one is only interested in the preservation of local properties. The stronger proviso given above is needed for the preservation of next-time-free LTL properties, which is the third class of properties we are interested in. For any LTL formula ϕ, $\mathit{prop}(\phi)$ is the set of propositions in property ϕ.

Definition 3.6 (Invisibility). An action $a \in A$ is *ϕ-invisible* in state $s \in S$ if and only if $\tau(s, a)$ is undefined or, for all $\pi \in \mathit{prop}(\phi)$, $\pi \in L(s) \Leftrightarrow \pi \in L(\tau(s, a))$. Action a is *globally ϕ*-invisible if and only if it is ϕ-invisible for all $s \in S$.

Informally, an action is globally ϕ-invisible if and only if it cannot change the truth value of formula ϕ.

C3 (invisibility). For any state $s \in S$, all actions in $r(s)$ are globally ϕ-invisible or $r(s) = \mathit{enabled}(s)$.

Theorem 3.7 (*Next-time-free LTL* (Peled, 1994; Valmari, 1992))**.** *Let r be a reduction function for LTS $\mathcal{T}$ satisfying C0, C1, C2, and C3; let ϕ be a next-time-free LTL formula. $\mathcal{T}$ satisfies ϕ if and only if the reduced LTS $\mathcal{T}_r$ satisfies ϕ.* (*See Manna and Pnueli*, 1991) *for a definition of the satisfaction of an LTL formula by an LTS.*)

3.2. The algorithm of Holzmann and Peled

Given the three theorems of the previous subsection, the challenge is to find interesting reduction functions and efficient algorithms implementing the corresponding reductions. A well-known reduction algorithm is the one described in Holzmann and Peled (1995) and Peled (1994). The most important aspects of the algorithm are the following: (1) It is based

on a depth-first search (DFS) of the state space of a concurrent system and (2) it uses a reduction function based on the process structure of the system. For details, the reader is referred to the original references. In this paper, we concentrate on the reduction function. To this end, we introduce a notion of processes in our framework of LTSs.

Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be an LTS. We assume that a set of processes $\mathcal{P}$ is associated with $\mathcal{T}$ as follows. Each $P \in \mathcal{P}$ is a set of actions, i.e., $P \subseteq A$. We require that the processes partition the set of actions: $A = \cup_{P \in \mathcal{P}} P$ and, for all $P, Q \in \mathcal{P}$ with $P \neq Q$, $P \cap Q = \emptyset$. It is often assumed, among others in the work of Holzmann and Peled, that processes are sequential. Thus, we require that, for any pair of *distinct* actions $a, b \in A$ belonging to the same process in $\mathcal{P}$ and any state $s \in S$ such that $a, b \in \mathit{enabled}(s)$, $b \notin \mathit{enabled}(\tau(s, a))$. That is, each two actions from a single process that are simultaneously enabled in a given state must disable each other. Clearly, this restriction disallows concurrency within processes, whereas it does allow choices. The sequentiality requirement for processes is not essential for our techniques explained later. They generalize trivially to concurrent processes. Nevertheless, we assume sequential processes in order to align with the original work of Holzmann and Peled.

The following definition is crucial in the formulation of the above mentioned reduction function. It depends on the class of properties one is interested in.

Definition 3.8 (Safety). An action $a \in A$ of some process P is *safe* if and only if it is independent of any (other) action $b \in A \setminus P$, i.e., of any action not in P. Action a of P is *safe for a given next-time-free LTL formula* ϕ if and only if it is independent of any action $b \in A \setminus P$ and globally ϕ-invisible.

As mentioned, the algorithm of Holzmann and Peled (1995) and Peled (1994) performs the reduction of the state space during a DFS. A DFS uses a *stack* to store partially investigated states. The reduction is obtained by defining for each state a so-called *ample set*. Note that the definition uses safety and, hence, depends on the particular class of properties to be verified.

Definition 3.9 (Reduction function *ample*). Let $s \in S$ be a state. Let $P \in \mathcal{P}$ be some process such that

- $\mathit{enabled}(s) \cap P \neq \emptyset$,
- for all $a \in \mathit{enabled}(s) \cap P$, a is safe (for some given next-time-free LTL formula ϕ), and,
- for all $a \in \mathit{enabled}(s) \cap P$, $\tau(s, a)$ is not on the DFS stack.

If no such process P exists, then define $\mathit{ample}(s) = \mathit{enabled}(s)$; otherwise, choose an arbitrary process P satisfying the above requirements and define $\mathit{ample}(s) = \mathit{enabled}(s) \cap P$. Set $\mathit{ample}(s)$ is said to be the ample set for s.

As an example, consider the reduction function r given in figure 2. It is clear that the singleton sets defined by r for the states 000, 100, 200, 210, 220, and 221, as well as the empty set defined for state 222 are ample sets.

It is not difficult to verify that reduction function *ample* satisfies provisos C0 through C3 given in the previous subsection. C0 follows easily from Definition 3.9. C1 and C3 follow from the safety requirement in Definition 3.9. Finally, C2 follows from the requirement in Definition 3.9 that the state resulting from the execution of an action in the ample set cannot be on the DFS stack unless the ample set consists of the entire set of enabled actions. As a result, the reduction via *ample* preserves deadlocks (Theorem 3.3), local properties (Theorem 3.5), and next-time-free LTL (Theorem 3.7).

As already mentioned in the previous subsection, proviso C2 can be weakened if one is only interested in (deadlock and) local-property preservation. Since the definition of reduction function *ample* is directly inspired by the four provisos, also the requirements on *ample* can be weakened in this case. It is then sufficient to require that at least one of the selected actions is not on the DFS stack, relaxing the third requirement in Definition 3.9 (Holzmann et al., 1992).

3.3. Implementation in SPIN

SPIN (Holzmann, 1991, 1997; SPIN, 2004) is a tool supporting the automatic verification of deadlock-, local-, and next-time-free LTL properties. Specifications of concurrent systems are written in the language PROMELA. The partial-order-reduction algorithm of Holzmann and Peled (1995) and Peled (1994) has been implemented in SPIN. To allow for the efficient computation of ample sets during a DFS, sufficient conditions for the safety of actions are derived directly from the PROMELA specification before starting the DFS (see Holzmann and Peled, 1995). For instance, a sufficient condition for the safety of an action that can be derived from a PROMELA specification is that it does not touch any global objects such as variables or communication channels. As explained, the precise requirements on ample sets as defined in Definition 3.9 (Reduction function *ample*) depend on the type of property that is being verified. SPIN automatically applies the weakest possible requirements when deriving ample sets.

4. Cluster-based partial-order reduction

4.1. The technique

We motivate our improvement of partial-order reduction by means of two different specifications of the same concurrent system given in figure 3. Specification `example1` has two global variables and four processes, each of them executing a single action assigning a value to one of these variables. Clearly, none of the actions is independent of all the other actions, which by Definition 3.8 (Safety) means that none of the actions is safe. Thus, reduction function *ample* of Definition 3.9 yields no reduction. The interested reader could verify that the LTS corresponding to the concurrent system has 25 states and 40 transitions.

Specification `example2` in figure 3 is a variant of `example1` with the processes *clustered* in pairs. The two clusters encapsulate the dependencies between processes. As a result, all actions within one cluster are independent of all actions within the other one. Our idea is to augment an LTS with a hierarchy of clusters and to generalize Definitions 3.8 and 3.9 to

```
System example1

int u, v;
P0:   a(u:=0);
P1:   b(u:=1);
P2:   c(v:=0);
P3:   d(v:=1);
```

```
System example2

C0:  int u;
     P0:   a(u:=0);
     P1:   b(u:=1);
C1:  int v;
     P2:   c(v:=0);
     P3:   d(v:=1);
```

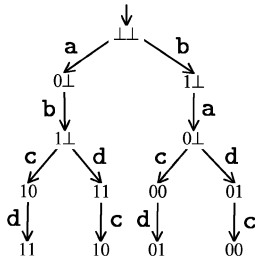

Figure 3. Two specifications of a concurrent system and a reduced state space.

clusters. Thus, it is possible to reduce the state space of the system of figure 3 to an LTS that includes all interleavings of actions within clusters `C0` and `C1` but only a single interleaving of actions from different clusters (see figure 3), while preserving all properties of interest.

Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be an LTS with processes $\mathcal{P}$. Recall that a process has been defined as a set of actions. The following definition builds upon that approach.

Definition 4.1 (Clustering). A *cluster* is a set of actions from A. Given a cluster C, a *clustering* $\mathcal{C} \subseteq 2^A$ of cluster C is a set of clusters partitioning C, i.e., $C = \cup_{D \in \mathcal{C}} D$ and, for all $D_0, D_1 \in \mathcal{C}$ with $D_0 \neq D_1$, $D_0 \cap D_1 = \emptyset$. A *cluster hierarchy* for the LTS $\mathcal{T}$ is a finite *tree* of clusters such that

– the tree has root A (i.e., the cluster containing all actions);
– for any node C in the tree, either C is a single process or the set of its children is a clustering of C.

Figure 4 shows a cluster hierarchy for system `example2` of figure 3. Note that the requirements in the above definition of a cluster hierarchy guarantee that a cluster in a

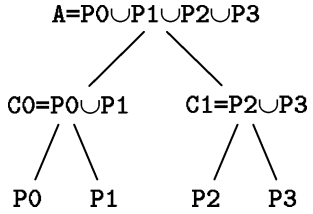

Figure 4. A cluster hierarchy for system `example2` of figure 3.

hierarchy always corresponds to a set of processes, and that a hierarchy always has the trivial cluster of all processes as its root and precisely all the clusters corresponding to all individual processes as its leafs. It turns out that this allows a straightforward generalization of safety and the *ample* function towards clusters.

Definition 4.2 (Cluster safety). Let C be some cluster in some cluster hierarchy. Action a in A is *cluster-C safe* if and only if it it is an element of C and it is independent of any action b not in C. Action a is cluster-C safe *for a given next-time-free LTL formula ϕ* if and only if it is independent of any action b not in C and globally ϕ-invisible.

Given some state, a cluster C is said to be safe (for a given next-time-free LTL formula ϕ) if and only if all its enabled actions are cluster-C safe (for the given next-time-free LTL formula ϕ).

Definition 4.2 states that an action is cluster-C safe for some cluster C if and only if it is independent of any action *outside the cluster*. Thus, since processes are clusters as well, Definition 4.2 is a true generalization of Definition 3.8 (Safety), which requires independence of any action *outside the process* executing the action. The same is true for safety with respect to a given next-time-free LTL formula because we did not change the invisibility requirement. For any process P, safety as defined in Definition 3.8 coincides with cluster-P safety as defined above. Further note that each action that is cluster-C safe with respect to some cluster C is also trivially safe with respect to any cluster higher in the hierarchy (i.e., closer to the root).

In the cluster hierarchy for `example2` shown in figure 4, actions `a` and `b` from processes `P0` and `P1` are cluster-`C0` and (trivially) cluster-`A` safe; actions `c` and `d` are cluster-`C1` and cluster-`A` safe. None of the actions is safe with respect to the cluster corresponding to its own process; this follows immediately from the fact that no action is safe according to Definition 3.8 (Safety).

Definition 4.3 (Reduction function *cample*). Let $s \in S$ be some state. Let C be some cluster from a cluster hierarchy for LTS $\mathcal{T}$ such that

- $enabled(s) \cap C \neq \emptyset$,
- C is safe (for a given next-time-free LTL formula ϕ), and,
- for all $a \in enabled(s) \cap C$, $\tau(s, a)$ is not on the DFS stack.

If no such cluster C exists, then define $cample(s) = enabled(s)$; otherwise, define $cample(s) = enabled(s) \cap C$.

Reduction function *cample* tries to find a cluster that has enabled actions that are furthermore safe with respect to that cluster and for which the resulting state is not on the DFS stack. If such a cluster exists, the enabled actions of the cluster form an acceptable ample set; if such a cluster does not exist, the set of all enabled actions is chosen as the ample set. Note that Definition 4.3 defines in fact a family of possible reduction functions, because of the freedom that (generally) exists in the choice of an appropriate cluster. It is interesting to observe that function *ample* of Definition 3.9 is a special case of function *cample* of

Definition 4.3 with a trivial hierarchy consisting only of the root cluster of all processes and the trivial clusters corresponding to the individual processes. As an example, consider again the concurrent system of figure 3. The figure also shows a reduced state space of this system. In each state, the values of variables u and v are given with $\perp$ meaning that a value is undefined. The sets of actions chosen in each state are ample sets as defined by function *cample* of Definition 4.3. In all states, the C0 and C1 clusters form the basis for the ample set.

Theorem 4.4 (Property preservation)**.** *The reduction of an LTS obtained via reduction function cample of Definition 4.3 preserves deadlock properties*, *local properties*, *and next-time-free LTL properties.*

Proof: It is straightforward to prove that *cample* satisfies provisos C0, C1, C2, and C3 of Section 3.1. The arguments are the same as in Section 3.2, where it is argued that function *ample* of Definition 3.9 satisfies these provisos. The desired result follows from Theorems 3.3, 3.5, and 3.7. □

As explained before, the requirements on function *ample* can be relaxed if one is only interested in (deadlocks and) local properties. The same relaxation is possible for function *cample*, because the relaxation only affects the stack-related requirements in Definitions 3.9 and 4.3. Our generalization of function *ample* generalizes the notion of safety only and does not affect this stack-related requirement.

Let us return to the example of figure 3 one more time. The reduced state space has 13 states and 12 transitions. This means reductions of the complete state space, consisting of 25 states and 40 transitions, of 48% in states and 70% in transitions. (Recall that standard partial-order reduction yields no reductions.) The above theorem shows that this reduction preserves all properties of interest.

The largest state-space reductions can be expected by choosing the ample sets as small as possible. One way to obtain small ample sets is to order the clusters in a hierarchy according to size (in terms of the number of processes in the clusters), and to search that ordering accordingly for an ample set. For our running example of figure 3, assuming the cluster hierarchy given in figure 4, an example ordering would be P0, P1, P2, P3, C0, C1, A. This ordering provides the ample sets underlying the state-space reduction shown in figure 3. Note that this strategy is not necessarily optimal. The size of a cluster is not necessarily a good measure for the number of actions enabled in that cluster, but there is usually a strong correlation.

Another practical issue is how to obtain useful cluster hierarchies. Intuitively, such a clustering should maximize the dependencies between processes within a cluster and minimize the dependencies between clusters. It is our aim to obtain such hierarchies statically, derived from the concurrent-system specification. In that way, we avoid the overhead of forming a hierarchy on-the-fly by inspecting dependencies between processes during the DFS. One possibility to derive a hierarchy in a static way is to preprocess the system specification and to cluster processes based on shared objects, using for example heuristics such as the one presented in Möller and Alur (2001). Another option is to use the existing hierarchical or modular structure of a specification; many contemporary specification and modeling languages such as UML and SDL include standard hierarchical structuring mechanisms. A

final option could be the use of advanced statistical clustering techniques based on run-time information as described in Kunz and Black (1995).

4.2. Implementation in SPIN

To validate our cluster-based reduction technique, we implemented a prototype in the verification tool SPIN, version 3.2.4. PROMELA, the input language of SPIN, does not (yet) support modular design. A process in PROMELA models is an instance of a `proctype`—a process template. To represent system hierarchy, we extended PROMELA with a `cluster` keyword. The `cluster` construct groups proctype definitions, clusters, and variables in a straightforward way. The extension is purely syntactical, exploited only in our enhanced partial-order-reduction algorithm during verification, and does not change the semantics of the PROMELA language. We assume that the designer provides the cluster hierarchy, as is common in contemporary modular or component-based design. Figure 5 shows how system `example2` of figure 3 can be coded in extended PROMELA. Note that the trivial root and leaf clusters of the cluster hierarchy of figure 4 need not be specified explicitly via the `cluster` keyword.

An important issue is the derivation of cluster-safety information and the handling of safety information at run-time during verification for the computation of ample sets. Cluster-safety information is determined at compile time based on syntactic criteria. We use the scopes of variables and other global objects such as communication channels for this purpose. If an object is completely within the scope of some cluster C in the hierarchy, it is said to be a cluster-C object. Note that the properties of a cluster hierarchy (Definition 4.1) imply that a cluster-C object is also a cluster-D object for any cluster D higher in the

```
/* system example2 */

cluster C0 {
   int u;
   proctype P0() { u = 0; } /* action a */
   proctype P1() { u = 1; } /* action b */
}

cluster C1 {
   int v;
   proctype P2() { v = 0; } /* action c */
   proctype P3() { v = 1; } /* action d */
}

init { atomic {
   run P0(); run P1(); /* cluster C0 */
   run P2(); run P3(); /* cluster C1 */
} }
```

Figure 5. System `example2` of figure 3 in extended PROMELA.

hierarchy than cluster C. A PROMELA statement is cluster-C safe if and only if it refers only to cluster-C objects. A run-time action inherits the safety properties of the statement it is derived from.

At run-time, the basis for the computation of an ample set is a safe cluster, i.e., a cluster whose enabled actions are all safe for that cluster. We need an efficient way of coding safety information of statements and the derived actions for a fast run-time computation of ample sets. Attaching to each statement simply all clusters for which it is safe is not efficient. The alternative of attaching to each statement only the lowest cluster in the hierarchy for which it is safe is also not acceptable because that may require expensive run-time traversals of the cluster hierarchy for determining an ample set.

A solution is found in the observation that all information from a cluster hierarchy needed for the computation of ample sets can be coded straightforwardly in terms of natural numbers. Assume that clusters are numbered in such a way any cluster receives a higher number than its children. Furthermore, assume that each statement is assigned the lowest number of any of the clusters for which it is safe. Note that such a number always exists because a statement is always safe for the root cluster of any hierarchy. Such a numbering scheme for clusters and statements implies that a statement in the scope of some cluster C is cluster-C safe if and only if its number is at most the number assigned to cluster C. A run-time action derived from the statement inherits its number. Thus, this action is cluster-C safe if and only if its inherited number is at most the number assigned to C. Finally, a cluster is safe if and only if all its enabled actions have a safety number that is at most the cluster number. It is clear that the described coding scheme requires very little storage overhead and it allows for very fast run-time safety checks.

In our implementation, we assign to each cluster a number equal to the height of the cluster hierarchy minus the depth of the cluster in the tree. This effectively structures a cluster hierarchy into levels. All clusters with the same depth in the cluster tree form a level. As an example, consider the hierarchy of figure 4. This hierarchy has height two. As a result, the hierarchy can be divided into three levels, the first one consists of clusters `P0`, `P1`, `P2`, and `P3` that are all numbered zero, the second one consists of the clusters `C0` and `C1` numbered one, and the third one consists of the trivial cluster `A` that is numbered two. Figure 6 in Section 5.2 shows another example of a subdivision of a cluster hierarchy into levels. Returning to the example of figure 4, the fact that actions `a` and `b` from processes `P0` and `P1` are cluster-`C0` and cluster-`A` safe is now simply represented by the number one that is assigned to these actions following our proposed numbering scheme. Actions `c` and `d` are also assigned a one which conforms to the fact that they are cluster-`C1` and cluster-`A` safe. The fact that in the initial state of system `example2`, cluster `C0` is safe follows now immediately from the fact that the two enabled actions of the cluster in that state, `a` and `b`, are numbered one which is equal to the number of cluster `C0`.

Note that the chosen coding of clusters into levels is just one possible coding scheme. Any scheme satisfying the requirement that a child node receives a lower number than its parent node will do. If one prefers, numbering schemes in which a cluster has always a larger number than its parent (e.g., when each cluster is assigned its depth in the cluster tree) is also possible when of course the run-time computation of ample sets is adapted accordingly.

For determining the order in which clusters are tested at run-time during the state-space traversal, we simply follow the scheme proposed in the previous subsection. Clusters are sorted at compile time based on their size in terms of the number of constituent processes, and tested at run-time in increasing order.

It is important to note that the prototype implementation of our techniques in SPIN only adapts the computation of ample sets. The reduction engine of SPIN is not changed and fully reused. From the computation of ample sets, only the part that determines the safety of all actions in the ample set is adapted. The parts that check the invisibility requirement (in case an LTL property is being verified) and the stack requirement for ample sets have not been changed. As a result, the computation of ample sets by the prototype automatically takes advantage of the weaker requirements for ample sets that are sufficient when one is only verifying deadlocks and/or local properties.

As a final remark, it should be mentioned that our implementation of extended PROMELA does not enforce the scopes implied by clusters. Thus, it is up to the designer to make sure that all variables and other objects like communication channels are declared in the appropriate clusters such that statements only touch variables that are within scope as defined by the cluster hierarchy. However, this requires little effort, and in conclusion we can say that the prototype allows for easy experimentation with our novel techniques.

5. Some state-space reduction experiments

As explained in the introduction, our main focus is on reducing the state spaces generated for exhaustive verification techniques; therefore, in this section, we focus on state-space generation, and we do not verify any properties. We performed three small experiments to validate the feasibility of our techniques. In the next section, we turn to a larger, more realistic verification case study.

The goal of the experiments in this section is to compare reductions in states and transitions obtained via our algorithm with reductions obtained via SPIN's standard partial-order reduction. As may be expected, similar to standard partial-order reduction, our algorithm performs best for systems with a large amount of concurrency. In all three experiments, we observe significantly larger reductions than the ones obtained with standard partial-order reduction. Moreover, our algorithm reduces verification times. The (small) overhead of the upgraded standard partial-order-reduction engine is marginalized by the gain in time because of the smaller number of generated states and transitions.

In the remainder, we show the results of our three state-space-generation experiments. The results for the same experiments reported in the earlier version of this paper, Basten and Bošnački (2001), deviate in some cases slightly from the results reported in this paper. For the experiments of Basten and Bošnački (2001), safety information for statements in the PROMELA models was determined manually, whereas for the current paper safety information was determined fully automatically from the cluster hierarchy. The order of magnitude of the obtained reductions is in all cases the same. The experiments for this paper were performed on a Linux PC with two 1 GHz Pentium-III processors and 4 GByte of SDRAM. All the shown verification times are in seconds.

Table 1. Results for the best-case example.

	Without POR		Standard POR		Cluster-based POR			
N	States	Trans	States	Trans	States	%	Trans	%
2	65	108	60	76	30	50	30	61
3	329	784	304	480	66	78	66	86
4	1657	5216	1532	2908	138	91	138	95
5	8313	32680	7688	17064	282	96	282	98

5.1. Best-case example

Our first experiment consists of a few variants of system `example2` of figures 3 and 5. The systems we verified consist of N pairs of processes and N variables, each pair of processes sharing one of these variables. The system with $N = 2$ corresponds to system `example2` of figure 5. Table 1 shows the results, including the reductions in states and transitions in percentages compared to SPIN with standard partial-order reduction. The numbers deviate from the theoretical results given in Section 4 due to implementation details of SPIN. SPIN adds to each process a special end transition that is independent of all other transitions. The standard SPIN partial-order reduction captures the independence of these special end transitions. Our prototype implementation takes, in addition, advantage of the independence of some transitions involving shared variables, as explained in Section 4. For each of the experiments, we used a hierarchy of three levels, with the nontrivial middle level consisting of clusters that coincide with process pairs. All verification times are less than 0.1 seconds.

5.2. Parity computer

The second example, taken from Alur and Wang (1999), models a parity computer with a tree structure. The system consists of a root module, N client modules as leafs, and join modules as intermediate nodes. The system with $N = 4$ is shown in figure 6. The figure also shows a cluster hierarchy (using self-explanatory abbreviations), that is immediately derived from the modular structure of the parity computer, and the structuring of this hierarchy into levels.

A client process starts with nondeterministically generating a bit value that it puts into the request variable that it shares with its parent join module. Subsequently, it continuously waits for an acknowledgment from its parent. Each time it receives an acknowledgment, it again sends an arbitrary bit value to its parent. A join process computes the parity (XOR) of its two inputs and transmits it upwards to its parent, while sending an acknowledgment to its children. Eventually, parity bits are delivered to the root process. The appendix contains some fragments of the (extended-)PROMELA code for the parity computer for $N = 4$.

Consider again the example in figure 6. Our cluster-based partial-order-reduction algorithm takes advantage of the fact that a join process communicates with its parent and its

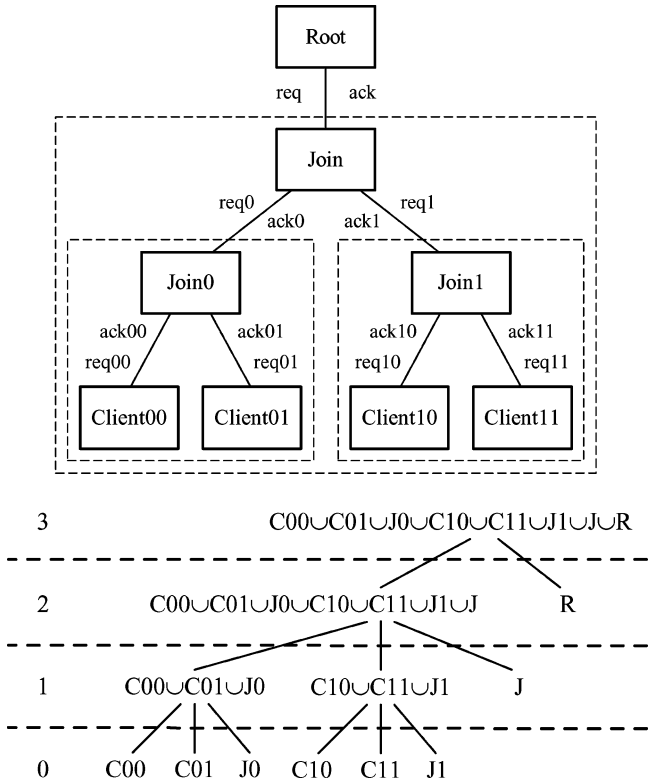

Figure 6. The parity computer with an associated cluster hierarchy.

children in alternating order. Because the cluster with root Join0 is independent of the cluster with root Join1, we can reduce the state space by basically serializing the transitions internal to one of these clusters with those internal to the other one. SPIN's standard reduction algorithm does not give any reduction because all transitions involve global variables. As Table 2 shows, the reduction with cluster-based partial-order reduction is quite impressive.

It may appear that more fine-grained clusterings than the one shown in figure 6 might improve the results even further. One could for example combine join processes with one of their children, e.g., one could insert clusters C00 ∪ J0, C10 ∪ J1, and C00 ∪ C01 ∪ J0 ∪ J in the hierarchy. However, this does not improve results because join nodes communicate with both their children simultaneously in one synchronized communication. Our model follows in this respect the original model of Alur and Wang (1999). If communication is made asynchronous, such more fine-grained clusterings do improve the reduction (but the state space of the model also increases).

The reduction with cluster-based partial-order reduction is worse than the reduction reported in Alur and Wang (1999) for the same examples obtained with the Next heuristic: The Next heuristic provides reductions in the number of states of the original model from

Table 2. Results for the parity computer.

	No/Standard POR			Cluster-based POR				
N	States	Trans	Time	States	%	Trans	%	Time
4	1749	4798	<0.1	1214	31	2148	55	<0.1
5	7933	27012	0.1	3910	51	6830	75	<0.1
6	69615	288678	1.1	21620	69	38372	87	0.2
7	320095	1.53×10^6	6.7	25228	92	44730	97	0.3
8	2.78×10^6	15.4×10^7	127.5	46806	98	85060	>99	0.6

71% for $N = 4$ up to over 99% for $N = 8$. The Next heuristic is a state-space reduction heuristic for invariant verification for concurrent systems (meaning that it targets a smaller class of properties than (cluster-based) partial-order reduction). The state space of a system is traversed through some kind of meta-steps; each such a step consists of a sequence of safe transitions of a single system component followed by a single unsafe transition of that component. In this way, it avoids some of the interleavings of safe transitions of different parallel components. Thus, the Next heuristic captures some of the same redundancies as (both standard and cluster-based) partial-order reduction.

It is difficult to draw any final conclusions about the comparative efficiency of the Next heuristic and cluster-based partial-order reduction based only on the parity-computer example. First, the modeling languages and the tools used for the experiments of Alur and Wang (1999) and for our experiments are different; this inevitably leads to (small) differences in the models, as well as differences in the precise notion of, for example, safety (which is derived using syntactic criteria). Second, the parity-computer example is one of the best cases for the Next heuristic; the authors provide themselves in Alur and Wang (1999) an example for which standard SPIN partial-order reduction gives better results than the Next heuristic, and they state that there are many more. Since our cluster-based partial-order reduction generalizes standard partial-order reduction, also our technique outperforms the Next heuristic in those cases. Thus, it is hard to compare the efficiency of the Next heuristic with cluster-based partial-order reduction. It is interesting to study their comparative performance in more detail, and to see whether the two techniques can be combined, which would possibly lead to even larger reductions.

5.3. Concurrent Alternating-Bit Protocol

As a final state-space generation experiment, we consider the Concurrent Alternating-Bit Protocol (CABP) of Koymans and Mulder (1990). The CABP has six components, as depicted in figure 7. Each component is modeled as a separate process.

The CABP uses the standard alternating-bit scheme to avoid communication errors. Component A fetches data from its environment and transmits this data repeatedly through channel K until an acknowledgment is received from D. Component A does not wait for a negative response before retransmitting. Channel K is unreliable in the sense that it can

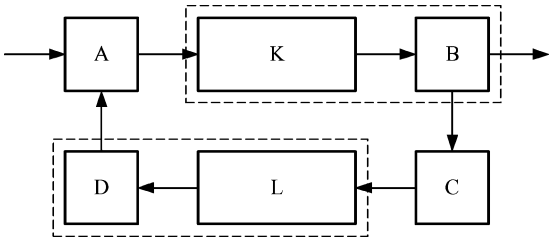

Figure 7. Concurrent Alternating-Bit Protocol.

Table 3. Results for the CABP.

No/Standard POR			Cluster-based POR				
States	Trans	Time	States	%	Trans	%	Time
16385	67074	0.3	12898	21	44461	34	0.2

corrupt or lose data. The role of B is to forward successfully received data to the environment; each correct reception is acknowledged to C. C transmits acknowledgments repeatedly via unreliable channel L. D receives acknowledgments from L and passes them to A.

As for the parity computer, also for the CABP the standard reduction algorithm does not produce any reduction; all transitions involve communications through global synchronous (rendez-vous) channels. The cluster-based reduction algorithm, however, capitalizes on part of the concurrency between the system modules, as the results shown in Table 3. The hierarchy consists of three levels, with clusters $K \cup B$ and $L \cup D$ being the only nontrivial clusters (see figure 7). This clustering exploits the independence between some of the actions in cluster $K \cup B$ and those outside this cluster, as well as the independence between some actions in cluster $L \cup D$ and those outside $L \cup D$. The implementation is such that no other process clustering improves the results.

6. A more realistic verification case study

6.1. Systems on a chip

To test our cluster-based technique in a more realistic context, we turn to modern system-on-chip design. Today's systems-on-chip (SoC) tend to become more and more often multi-processor systems. Single-chip multiprocessors are a natural evolution of single-processor solutions, to cope with the challenges in hardware design caused by the ever further miniaturization. To obtain scalable communication between processing elements, the replacement of bus-based communication techniques with network-based techniques is being investigated (Benini and De Micheli, 2002). These novel communication techniques build on techniques known from wide-area networks such as router-based, packet- or circuit-based communication. Figure 8 shows an example architecture where (on-chip) processing elements (PEs)

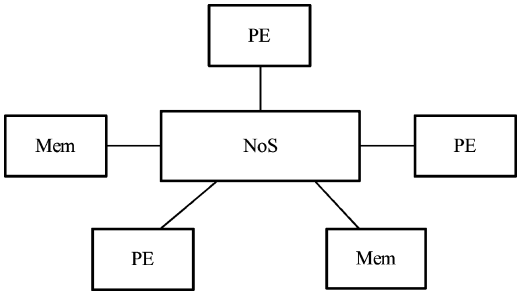

Figure 8. A SoC with NoS-based communication.

are connected via a network-on-silicon (NoS). Processing elements typically consist of a processor core with their own local memory which may or may not be shared with other processing elements. Some PEs may be only memories.

Although NoS-based communication scales more easily to larger numbers of PEs than bus-based communication, it has the disadvantage that communication is slow compared to bus-based communication with a limited number of PEs. In typical examples of the NoSs under study, connections need to be opened for communication between PEs, and data packets must travel through multiple routers to reach their destination. Thus, caching of data is of crucial importance to prevent unacceptable delays. As a case study, we investigate a directory-based cache-coherency protocol (Culler et al., 1999, Chapter 8) for NoS-based SoCs. Note that we do not aim at a complete verification of the cache coherency protocol. Our aim is simply to test our technique in a more-or-less realistic setting. We focus on the generation of state spaces and the verification of a few simple properties.

The experiment consists of three parts. In the first part, we consider a simple SoC with caches for the local memories of PEs. The architecture is such that a cache-coherency protocol is not required. In the second part, we consider a SoC architecture with cache coherency. The idea behind this split is that the PEs in the first architecture are completely independent, which should give good reductions with our technique, whereas in the second architecture the cache-coherency protocol introduces a lot of global synchronization, which is expected to have a negative impact on the results. The possibility to study systems with and without much global synchronization in one setting makes this case study attractive. In the third part of the case study, we investigate the influence of cluster hierarchies on our verification results in some detail. For the interested reader, the appendix contains some fragments of the (extended-)PROMELA source code used in the first part of this case study. These code fragments provide some insight in the details of the models used in this case study. The model used in the second part of the case study is too large to include in the paper; it is available through a web site, as explained at the end of the paper.

6.2. A SoC without cache coherency

We first consider a 2-PE SoC as illustrated in figure 9. Each of the PEs consists of a processor core, a memory, a cache and a communication assist. The memories are shared between the

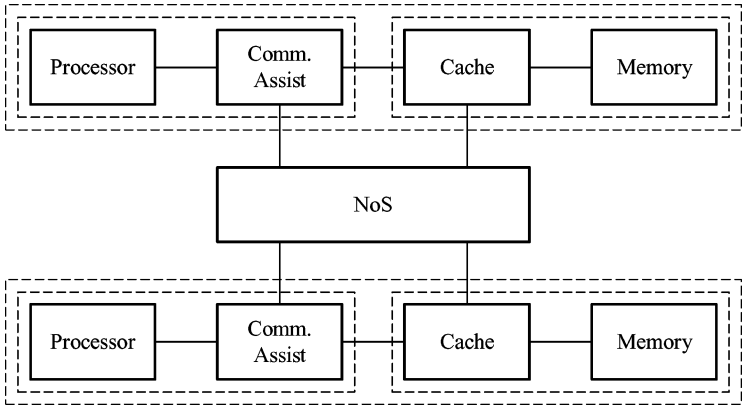

Figure 9. A 2-PE SoC without cache coherency, including an associated clustering.

two PEs. However, in this simple setup, the cache only caches data from the local memory, so no cache coherency is required. The communication assist directs reads and writes of its corresponding processor core either to the local cache or to the NoS when a read or write accesses the other memory. In the latter case, the NoS forwards the read or write to the cache at the receiving side, which in turn may or may not contact the memory at that side. In case of a read, the read data travels from the cache or memory where it was stored via the communication assist to the processor requesting the data.

The clustering shown in figure 9 intends to cluster the local communication between processors and their communication assists, the local communication between caches and memories, and on a higher level the communication within one PE. It intends to capitalize on the relative independence of the clustered components from their environments.

The (extended-)PROMELA model that we built from this system models each of the depicted components as a process. The processors continuously perform random reads and writes. To keep the state space within reasonable limits that still allow verification of results, we had to assume that each memory only has two addresses and each cache has one entry. Although this is a very simple, in practice unrealistic situation, it is still of interest because the memories are shared implying that each processor can access four different memory locations. Moreover, using SPIN with standard partial-order reduction (our reference), larger cache and/or memory sizes could simply not be verified on our machine (with 4 GByte of memory).

Table 4 shows the results of our experiments with the system of figure 9. The first entry of both parts of the table shows the results of state-space generation using both standard partial-order reduction and our cluster-based technique. The analysis showed that our model does not have deadlock, which is of course a desirable property of the modeled system. The very large reduction that is obtained (compared to standard partial-order reduction!) is not really surprising because the various (clustered) components in the system are relatively independently performing concurrent actions. As said before, our technique performs well for such systems. The second entry in both parts of the table shows the results of verifying

Table 4. Results for a 2-processing-element SoC/no cache coherency.

	Standard POR		
	States	Trans	Time
State space/deadlocks	35.0×10^6	73.5×10^6	966.4
$\neg(\neg$ request U addrcached)	30.6×10^6	73.4×10^6	2396.6

	Cluster-based POR				
	States	%	Trans	%	Time
State space/deadlocks	3.7×10^6	90	7.7×10^6	90	92.7
$\neg(\neg$ request U addrcached)	16.6×10^6	46	41.2×10^6	44	1278.7

the LTL property $\neg(\neg$request U addrcached), which states that no addresses are cached if there is no pending request for this address. This case also shows a good reduction, although not as good as the reduction obtained for state-space generation. The reason is that the property refers to variables within clusters at the bottom of the hierarchy meaning that actions touching these variables can no longer be considered safe with respect to any cluster because of the invisibility requirement in Definition 4.2 (Cluster safety). Note that the verified property is a safety property (nothing bad will happen). We did not verify any liveness properties (something good will happen, e.g., any read will eventually result in the receipt of data) because those would require a fairness assumption. However, the combination of enforcing fairness in a context with SPIN rendez-vous communication and partial-order reduction is not possible in SPIN (Bošnački, 1999). Thus, we could not verify any liveness properties because none of these will be true when fairness is not guaranteed. However, as said before, our goal is not to exhaustively verify the system under study. We simply want to test our techniques in realistic settings.

6.3. A SoC with cache coherency

The second part of our case study concerns a SoC architecture that includes a cache-coherency protocol. Figure 10 shows the architecture of the system verified in this part of the experiment. The reason to turn to cache coherency is that we want to investigate how our technique performs if a system exhibits more global communication. Directory-based cache coherency (Culler et al., 1999, Chapter 8) is an excellent test-case for this purpose because it requires quite a lot of communication between PEs to guarantee that the caches are kept consistent.

In a SoC as depicted in figure 8, where multiple processing elements employ caches that cache data from shared memory, the need arises for coordination between these caches to guarantee that all data is kept consistent. Particularly, when one of the PEs wants to write data to a cache for a certain memory location, other caches may have a copy of that memory location. At that point, the contents of these other caches for that address become

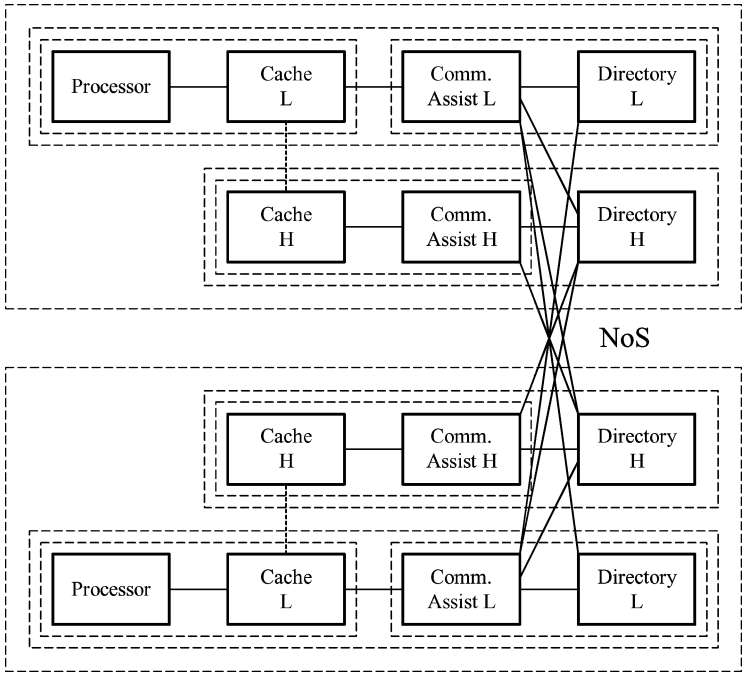

Figure 10. A 2-PE SoC with cache coherency, including an associated clustering.

invalid. In traditional bus-based systems, communication between processors and caches is carried out via a shared bus. Caches can snoop this bus to check if other caches are written for an address they contain. This system does not work in a network-based system, because snooping of network-based communication is not possible (efficiently). Instead, a directory-based cache-coherency protocol is used. A directory is located with every memory to maintain information about which caches have copies of a particular memory location at any given time. When some cache intends to change the contents of a memory location, it has to acquire the exclusive right to do so from the directory.

The structure of our (extended-)PROMELA model of an architecture with a directory-based cache-coherency protocol is visualized in figure 10 and consists of two large and identical PE clusters. To limit the size of the state space of our model, the NoS has been modeled as a collection of asynchronous, buffered channels instead of as a process; the model corresponds to the fact that communication over NoSs goes typically via connections with implicit buffers. (All other communication in the model is by means of synchronous rendez-vous.) A single PE is modeled with a processor, issuing random read and write operations to a cache, that now caches data from both the local and the remote memory, a communication assist, to direct communication to the correct PE, and a directory which maintains the information about caching of blocks of the memory with which the directory is associated. In this model, we have abstracted from the actual memory process to save on the size of the state space to be verified. For the same reason, caches have only one entry, and

each memory is assumed to have only one address. Note that each processor can still access two memory locations because of the sharing. The cache, the communication assist, and the directory have been modeled with two PROMELA processes. A read or write operation from a processor may lead to a chain of events from cache via communication assist to a directory. The directory may decide to invalidate the content of certain caches, leading to another chain of events back to some cache(s). To allow this to happen, the components have been split in a high-priority part that deals with the invalidation and a low-priority part that deals with the other communications. Note that the Cache L and Cache H parts have access to shared variables in which the status of the cache is stored; this is indicated by the dotted line in figure 10 between the processes because it indicates a dependency which may interfere with the clustering technique. The cluster structure shown by the dashed lines in figure 10 is inspired by the fact that the communication in the form of chains of events goes from processor to directory through the low-priority parts and from directory to cache through the high-priority parts.

Table 5 shows the results for the cache-coherency model. For the state-space-generation and deadlock-detection experiment, the results are still quite good, despite the fact that the inter-cluster dependencies and the amount of global communication have increased a lot compared to the model without cache coherency. We also verified a mutual-exclusion property stating that the two caches cannot simultaneously gain exclusive access to the same memory block. The results for this property show smaller gains than obtained so far (although still acceptable). The reason is, as explained earlier, caused by the invisibility requirement. Many actions that are in the state-space-generation experiment safe for clusters low in the hierarchy, are no longer safe in the property-verification experiment because they touch the variables involved in the property. In our experiment, the mutual-exclusion property refers to variables that are local to the Cache L and H processes. Thus, cache actions that touch these variables are no longer safe, reducing the gain that can be obtained with our clustering technique.

The two experiments with the verification of LTL properties of this subsection and the previous one suggest to have a closer look at the interaction between our clustering technique and property scopes, where the scope of some property can be defined as the set of processes

Table 5. Results for a 2-processing-element SoC/cache coherency.

	Standard POR				
	States		Trans		Time
State space/deadlocks	2.52×10^6		4.30×10^6		97.4
Mutual exclusion	3.21×10^6		6.61×10^6		103.7
	Cluster-based POR				
	States	%	Trans	%	Time
State space/deadlocks	1.14×10^6	55	1.57×10^6	64	35.9
Mutual exclusion	2.16×10^6	33	4.29×10^6	35	66.8

possibly affecting the objects referred to in the property. We briefly do so in the next subsection.

As a final remark, during the verification of the mutual-exclusion property, we discovered a subtle modeling error that allowed for a race condition between an invalidation message and the acquisition of exclusive access to a particular memory location, thus leading to the opportunity for two caches to have exclusive access to a single location in memory at the same time. Of course, this error is not inherent in the original protocol described in Culler et al. (1999). It does point out, however, the intricacies involved in implementing such protocols, which is the underlying motivation for our work. The current experimental results are based on a corrected version of the model.

6.4. The influence of cluster hierarchies

It is clear that the effectiveness of our reduction technique heavily depends on the quality of a cluster hierarchy. So far, our cluster hierarchies have been inspired entirely by the relative independence of clusters. However, as already mentioned, when verifying LTL properties, the invisibility requirement in the definition of safety for an LTL property may interfere with the independence requirement in determining good clusters. Ideally, a cluster hierarchy takes into account both independence and invisibility aspects. Note that this implies property-specific clusterings, which may not always be feasible. However, in an implementation where automatic clustering heuristics are available, property-specific clusterings may be feasible. The clustering heuristics should then of course take into account both independence and invisibility aspects.

In this subsection, we briefly consider property-specific clusterings for the two SoCs of the previous two subsections. As a first attempt, one may consider to put all processes in the scope of a property into one cluster, where, as mentioned, the scope of a property consists of all processes that touch variables or other objects in the property. This cluster may be subdivided further into clusters to capitalize on potential independence of actions within this cluster from their context (insofar such actions satisfy the invisibility requirement of course). The idea behind clustering all processes in the scope of a property into one cluster is that it should maximize the potential for reduction in the remaining processes, that can be clustered focusing entirely on independence. Figure 11 shows such clusterings for the two SoCs of the previous subsections, based on the two properties verified in the reported experiments. The clustering shown in figure 11(a) combines the Processor and Cache processes of one of the two PEs of the SoC of figure 9 into a single cluster. This immediately raises the question whether it would not be better to include also the communication assist in that cluster. The clustering of figure 11(a) does not appear to be very appropriate from the independence perspective. The clustering of figure 11(b) puts all cache processes into a single cluster. However, this clustering can also be improved from the independence perspective, namely by integrating the Processor processes inside the cache cluster. Figure 12 gives adapted clusterings.

The results of the verification experiments for the alternative clusterings of figures 11 and 12 are shown in Table 6. The top part of the table repeats the results of the original experiments. The bottom part of the table shows the results for the alternative clusterings,

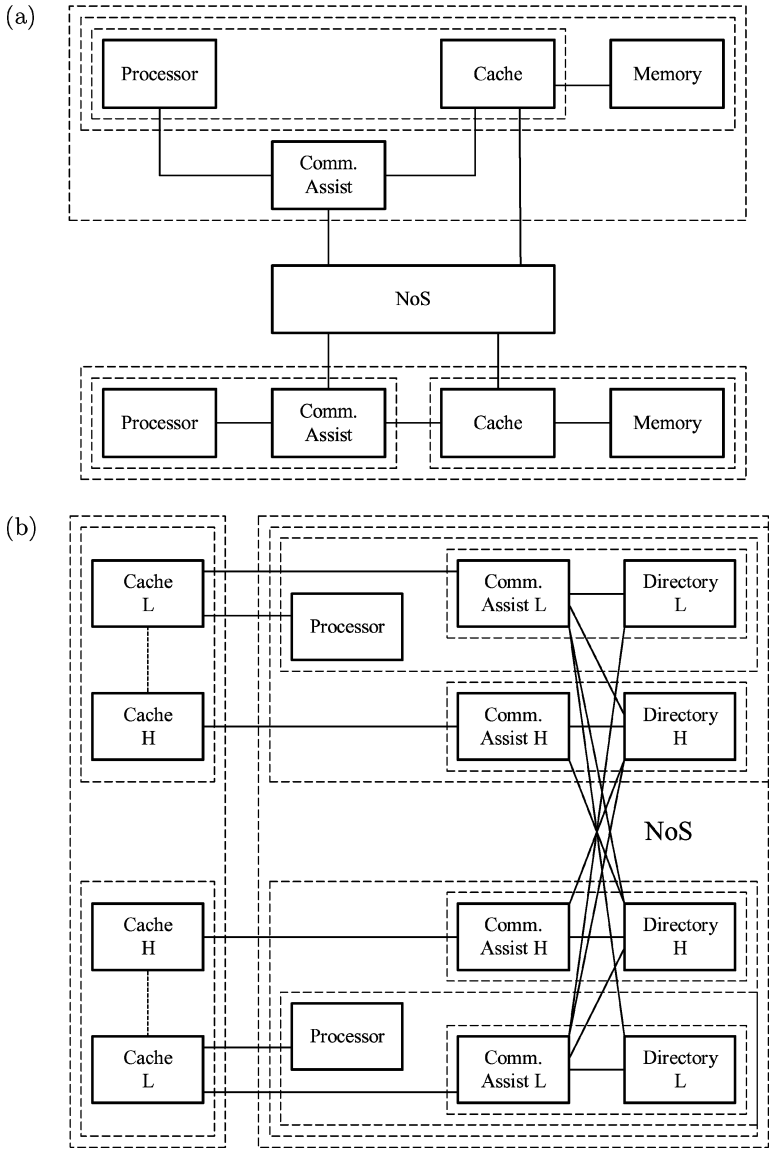

Figure 11. (a) The 2-PE SoC without cache coherency, alternative clustering one. (b) The 2-PE SoC with cache coherency, alternative clustering one.

including the obtained 'reductions' in comparison with the original independence-driven clusterings. The results show that none of the alternative clusterings improve the verification results. They confirm however the earlier observation that the purely property-driven clusterings of figure 11 can be improved from the independence perspective, as shown in figure 12.

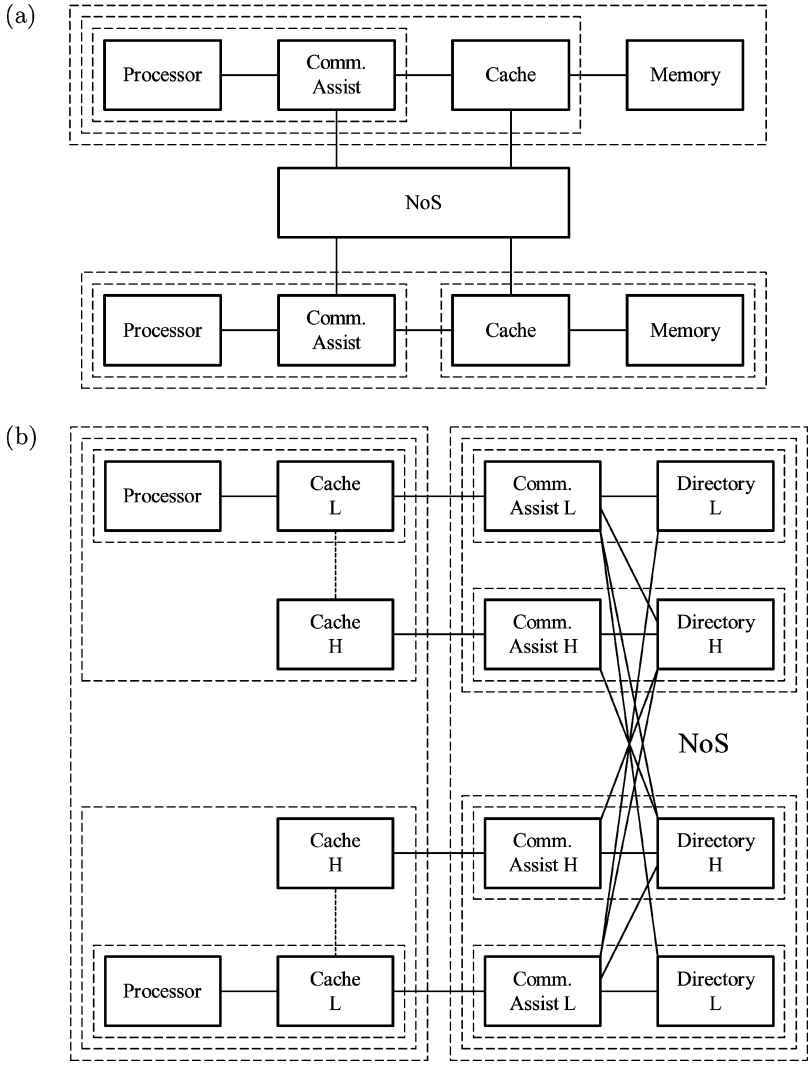

Figure 12. (a) The 2-PE SoC without cache coherency, alternative clustering two. (b) The 2-PE SoC with cache coherency, alternative clustering two.

The experiment with the alternative clusterings shows that it is not in general a good idea to focus a cluster hierarchy entirely around property scopes. Although in both our experiments an independence-driven clustering gives the best results, we believe it is too early to draw the conclusion that this is generally the case. It is an interesting topic for future work to investigate (automatic) clustering heuristics that take into account both independence and invisibility, and to experiment more extensively with clusterings.

Table 6. Results for alternative clusterings.

	Original independence-driven clusterings		
	States	Trans	Time
No cache coherency	16.6×10^6	41.2×10^6	1278.7
Cache coherency	2.16×10^6	4.29×10^6	66.8

	Alternative clusterings				
	States	%	Trans	%	Time
No cache coh., alt. 1	18.8×10^6	−13	46.0×10^6	−12	1436.6
No cache coh., alt. 2	18.6×10^6	−12	45.8×10^6	−11	1430.7
Cache coh., alt. 1	3.00×10^6	−39	5.96×10^6	−39	95.5
Cache coh., alt. 2	2.79×10^6	−29	5.45×10^6	−27	85.8

7. Conclusion

The main contribution of this paper is an enhancement of the partial-order-reduction scheme of Holzmann and Peled (1995) and Peled (1994). Using the inherent structure of concurrent systems, we improve the way the safety of actions is determined syntactically during the compilation of the system specification. The structure of a system is captured in a cluster hierarchy, where a cluster corresponds to a set of processes. The ample sets resulting from our technique contain the enabled actions of a cluster, i.e., they may contain the enabled actions from more than one process. Although ample sets with actions from several processes have been considered earlier (e.g., Alur et al., 1997), to the best of our knowledge, the idea of exploiting hierarchical system structure for determining ample sets was first published in an earlier version of this paper namely Basten and Bošnački (2001).

We implemented our algorithm in the SPIN tool (Holzmann, 1991, 1997; SPIN, 2004), by upgrading SPIN's standard partial-order-reduction engine. SPIN's input language PROMELA has been extended with a `cluster` keyword to allow users to specify cluster hierarchies. The implementation has been tested on several small examples and one more realistic verification case study, all known from the literature. The obtained results are encouraging: Compared to SPIN's standard partial-order-reduction algorithm, significantly larger reductions of state spaces are obtained and verification times are decreased. Only for the verification of LTL properties that involve variables from several largely independent processes, the gains obtained may not always be very large. The reason is the fact that safety of actions with respect to some LTL property requires that the actions do not touch objects referred to by the property (invisibility). It is possible to take into account the scope of an LTL property (i.e., the processes possibly affecting the property) in the cluster hierarchy. Our initial experiments with property-driven clusterings are however not encouraging, but it is too early to draw definite conclusions.

This brings us immediately to the most interesting topic for future research, namely clustering heuristics (see, e.g., Kunz and Black, 1995; Möller and Alur, 2001). We obtained the

best results with cluster hierarchies that try to capture dependencies within clusters, thus maximizing independence between clusters. For modeling languages that contain modularity constructs, cluster hierarchies can be derived from the hierarchical structure of a system as it is specified by a system designer, because this structure typically emphasizes (in) dependencies. However, good clustering heuristics possibly yield better results. Such heuristics are also useful when a modeling language does not have modularity constructs (such as SPIN's PROMELA) and they could furthermore take into account the scopes of LTL properties in case of LTL verification. Since the clusterings resulting from heuristics taking into account the property being verified are by definition property specific, automating the derivation of clusterings is important.

Another interesting direction for further work is to generalize our techniques. On the one hand, we have the general theory of partial-order reduction that allows arbitrary ample sets satisfying certain provisos. On the other hand, we have techniques that restrict ample sets to the enabled actions of only a single process (or all processes). In between these extremes, we can position the class of techniques that consider ample sets consisting always of all the enabled actions of an arbitrary *set* of processes, or cluster in our terminology.[1] Our technique building on cluster hierarchies is a prime example of this class. However, in principle, there is no real reason to limit the sets of processes considered as ample sets to a cluster hierarchy in the form of a tree of clusters as is done in Definition 4.1. The only reason not to consider all possible subsets of processes as potential ample sets during a state-space traversal is that in general the number of possible subsets is prohibitively large. However, any means of specifying a meaningful set of interesting clusters will do. Consider for example the two alternative clusterings of the same system depicted in figures 10 and 12(b), the first one independence driven and the second one property driven. The union of the two sets of clusters specified by these two clusterings is a perfectly valid set of clusters to consider for determining ample sets. In fact, it would combine the advantages of both the independence- and the property-driven clusterings. It is clear though that this set of clusters does no longer fit our definition of a cluster hierarchy: It is not a tree.

The consequences of generalizing a cluster hierarchy to an arbitrary set of clusters are not trivial but appear to be manageable. It is still possible to order clusters according to size in terms of the number of constituent processes; as explained, this is important for obtaining small ample sets. Unfortunately, it is not possible to specify arbitrary sets of clusters with the `cluster` keyword that we added to SPIN's PROMELA language for specifying cluster hierarchies. Clusterings specified via the `cluster` keyword or other standard modularity constructs always result in trees. However, automatic clustering heuristics as discussed before could easily be used to derive arbitrary sets of clusters, increasing the potential interest of automatic clustering heuristics even further. As a final consequence of the proposed generalization, it is important to notice that the coding scheme for cluster hierarchies and for determining safe clusters that we use in our SPIN implementation, as discussed in Section 4.2, does no longer work for arbitrary sets of clusters. However, this is not really a fundamental issue; other efficient coding schemes exist.

Apart from a further generalization of our work, it will also be interesting to see how our approach works in combination with other state-space-reduction heuristics. Following

Emerson et al. (1997), it is easy to show that our technique is fully compatible with symmetry reduction. The two techniques are orthogonal because they exploit different features of concurrent systems. We agree with the conjecture in Alur and Wang (1999) that partial-order reduction (and also our enhancement) is compatible with the Next heuristic. It seems though that cluster-based partial-order reduction and the Next heuristic are not fully orthogonal, because they capture to some extent the same redundancies in state spaces. It is interesting to study the relation between the Next heuristic and cluster-based partial-order reduction in more detail.

We implemented our clustering technique in SPIN 3.2.4, which was the latest release of SPIN at the time we started our implementation. It remains to implement the technique in the latest release of SPIN to take advantage of the improved partial-order reduction (Holzmann, 1999) in that release. Furthermore, the combination of our techniques with v-Promela (Leue and Holzmann, 1999; Kamel and Leue, 2000) is also promising; v-Promela is a visual formalism that extends PROMELA with hierarchical modeling and other concepts from object-oriented design. The v-Promela formalism supports the structuring of a model into a hierarchy of capsules, which directly corresponds to our notion of a cluster hierarchy. Finally, our cluster-based algorithm is compatible with the upgrade of SPIN's engine for timed systems from Bošnački and Dams (1998). It is very likely that it can be combined with techniques for partial-order reduction for timed automata (Bengtsson et al., 1998; Dams et al., 1998; Minea, 1999).

Remark. The SPIN implementation of our clustering technique and the models used for the experimental results presented in this paper are available through http://www.win.tue.nl/~dragan/CPOR-Spin.

Appendix

For the interested reader, this appendix provides fragments of the (extended-)PROMELA sources of two of our experiments.

Parity computer

```
mtype = { none, zero, one, ready, idle, wait, null, pos };
mtype req = none, ack = idle;
proctype Root() {
start:
   do :: atomic { req == zero -> ack = wait; goto req_zero }
      :: atomic { req == one -> ack = wait; goto req_one }
   od;
req_zero:
   atomic { req == ready -> ack = null; goto wait_req }
req_one:
   atomic { req == ready -> ack = pos; goto wait_req }
```

```
wait_req:
   atomic { req == none -> ack = idle; goto start }
}
cluster C {
mtype req0 = none, ack0 = idle, req1 = none, ack1 = idle;
proctype Join() {
start:
   do :: atomic { (req0 == zero) && (req1 == zero) ->
         ack0 = wait; ack1 = wait; goto parity_zero }
      :: atomic { (req0 == zero) && (req1 == one) ->
         ack0 = wait; ack1 = wait; goto parity_one }
      :: atomic { (req0 == one) && (req1 == one) ->
         ack0 = wait; ack1 = wait; goto parity_zero }
      :: atomic { (req0 == one) && (req1 == zero) ->
         ack0 = wait; ack1 = wait; goto parity_one }
   od;
parity_zero:
   atomic { (req0 == ready) && (req1 == ready) ->
      req = zero; goto wait_ack }
parity_one:
   atomic {(req0 == ready) && (req1 == ready) ->
      req = one; goto wait_ack }
wait_ack:
   atomic { ack == wait -> req = ready }
   do :: atomic { ack == null -> req = none; goto null_ack }
      :: atomic { ack == pos -> req = none; goto pos_ack }
   od;
null_ack:
   atomic { ack == idle ->
      ack0 = null; ack1 = null; goto wait_reqs }
pos_ack:
   atomic { ack == idle ->
      ack0 = pos; ack1 = pos; goto wait_reqs }
wait_reqs:
   atomic { (req0 == none) && (req1 == none) ->
      ack0 = idle; ack1 = idle; goto start }
}
cluster C0 {
mtype req00 = none, ack00 = idle, req01 = none, ack01 = idle;
proctype Join0() { /* snip */ }
```

```
proctype Client00() {
start:
   do :: atomic { true -> req00 = zero; break }
      :: atomic { true -> req00 = one; break }
   od;
   atomic { ack00 == wait -> req00 = ready}
   atomic { (ack00 == pos) || (ack00 == null) -> req00 = none}
   ack00 == idle -> goto start;
}
proctype Client01() { /* snip */ }
} /* end C0 */
cluster C1 { /* snip */ }
} /* end C */
init { atomic {
   run Root();
   /* cluster C */
      run Join();
      /* cluster C0 */
         run Join0(); run Client00(); run Client01();
      /* cluster C1 */
         run Join1(); run Client10(); run Client11();
} }
```

SoC without cache coherency

```
/* 2-processing-element SoC without cache coherency */
/* parameter definitions omitted
   bl(a): memory block of address a
   ln(b): cache line for memory block b
   cc(b): memory block b currently in the cache */
mtype = {rd,wr,data}
chan ca_to_n0, n_to_c0, ca_to_n1, n_to_c1 = [0] of {mtype,byte};
chan n_to_ca0, c_to_n0, n_to_ca1, c_to_n1 = [0] of {mtype};
cluster node0 { /* Processor+CommAssist+Cache+Memory */
chan ca_to_c0 = [0] of {mtype,byte};
chan c_to_ca0 = [0] of {mtype};
cluster pca0 { /* Processor+CommAssist */
chan p_to_ca0 = [0] of {mtype,byte};
chan ca_to_p0 = [0] of {mtype};
proctype Process0(){
byte a;
```

```
do :: atomic{
      a = 0;
      do :: break;
         :: a<2*MEMSIZE-1 -> a++;
      od;}
      if :: p_to_ca0!wr(a);
         :: p_to_ca0!rd(a); ca_to_p0?data;
      fi
od
}
proctype CommAssist0(){
byte addr;
do :: p_to_ca0?rd(addr) ->
      if :: addr<MEMSIZE -> ca_to_c0!rd(addr); c_to_ca0?data;
         :: else -> ca_to_n0!rd(addr-MEMSIZE); n_to_ca0?data;
      fi;
      ca_to_p0!data;
   :: p_to_ca0?wr(addr) ->
      if :: addr<MEMSIZE -> ca_to_c0!wr(addr);
         :: else -> ca_to_n0!wr(addr-MEMSIZE);
      fi
od
}
} /* end of pca0 (Processor+CommAssist)*/
cluster cm { /* Cache-Memory 0 */
chan c_to_m0, m_to_c0 = [0] of {mtype};
proctype Cache0() {
byte cached[NRLINES], status[NRLINES];
byte addr,block,line;
do :: ca_to_c0?rd(addr) ->
      block = bl(addr);
      if :: !cc(block) ->
            line = ln(block);
            if :: status[line]==modified -> c_to_m0!wr;
               :: else ->
            fi;
            c_to_m0!rd; m_to_c0?data;
            cached[line]=block; status[line]=clean;
         :: else->
      fi;
      c_to_ca0!data;
   :: ca_to_c0?wr(addr) ->
```

```
      block = bl(addr); line = ln(block);
      if :: !cc(block) ->
            if :: status[line]==modified -> c_to_m0!wr;
               :: else ->
            fi;
            c_to_m0!rd; m_to_c0?data;
            cached[line]=block;
         :: else ->
      fi;
      status[line] = modified;
   :: n_to_c0?rd(addr) ->
      block = bl(addr);
      if :: !cc(block) ->
            line = ln(block);
            if :: status[line]==modified -> c_to_m0!wr;
               :: else ->
            fi;
            c_to_m0!rd; m_to_c0?data;
            cached[line]=block; status[line]=clean;
         :: else->
      fi;
      c_to_n0!data;
   :: n_to_c0?wr(addr) ->
      block = bl(addr); line = ln(block);
      if :: !cc(block) ->
            if :: status[line]==modified ->
                  c_to_m0!wr;
               :: else ->
            fi;
            c_to_m0!rd; m_to_c0?data;
            cached[line]=block;
         :: else ->
      fi;
      status[line] = modified;
od
}
proctype Memory0(){
do :: c_to_m0?wr;
   :: c_to_m0?rd -> m_to_c0!data;
od
}
} /* end of cm cluster (Cache-Memory 0) */
} /* end of node0 (Processor+CommAssist+Cache+Memory)*/
```

```
proctype Network(){
byte addr;
do :: ca_to_n0?rd(addr) -> n_to_c1!rd(addr);
         c_to_n1?data; n_to_ca0!data;
   :: ca_to_n0?wr(addr) -> n_to_c1!wr(addr);
   :: ca_to_n1?rd(addr) -> n_to_c0!rd(addr);
         c_to_n0?data; n_to_ca1!data;
   :: ca_to_n1?wr(addr) -> n_to_c0!wr(addr);
od
}
cluster node1{ /* snip */ }
init{ atomic{
   /* cluster node0 */
      /* pca0 */
         run Process0(); run CommAssist0();
      /* cm0 */
         run Cache0(); run Memory0();
   run Network();
   /* cluster node1 */
      /* cluster pca1 */
         run Process1(); run CommAssist1();
      /* cm1 */
         run Cache1(); run Memory1();
} }
```

Acknowledgments

Our work is inspired by the Next heuristic of Rajeev Alur and Bow-Yaw Wang presented in Alur and Wang (1999). We thank Dennis Dams, Jeroen Rutten, and the referees of earlier versions of this paper for their contributions and suggestions. Gerard Holzmann helped us through the implementation of our technique in SPIN, for which we are very grateful.

Note

1. In fact, assuming sequentiality of processes (see Section 3.2), ample sets containing actions of a certain process will always contain all actions of that process, i.e., ample sets not corresponding to a set of processes do not exist.

References

Alur, R., Brayton, R., Henzinger, T., Qadeer, S., and Rajamani, S. 1997. Partial-order reduction in symbolic state-space exploration. In O. Grumberg, editor, *Computer Aided Verification, CAV'97, Proceedings*, Lecture Notes in Computer Science 1254, Springer, pp. 340–351.

Alur, R. and Wang, B.-Y. 1999. "Next" heuristic for on-the-fly model checking. In J. Baeten and S. Mauw, editors, *Concurrency Theory, CONCUR'99, Proceedings*, Lecture Notes in Computer Science 1664, Springer, pp. 98–113.

Basten, T. and Bošnački, D. 2001. Enhancing partial-order reduction via process clustering. In *Automated Software Engineering, ASE 2001, 16th. IEEE International Conference, Proceedings*, IEEE Computer Society Press, pp. 245–253.

Bengtsson, J., Jonsson, B., Lilius, B., and Yi, W. 1998. Partial order reductions for timed systems. In D. Sangiorgi and R. de Simone, editors, *Concurrency Theory, CONCUR'98, Proceedings*, Lecture Notes in Computer Science 1466, Springer, pp. 485–501.

Benini, L. and De Micheli, G. 2002. Networks on chip: A new SOC paradigm. *IEEE Computer*, 35(1):70–78.

Bošnački, D. 1999. Partial order reduction in presence of rendez-vous communications with unless constructs and weak fairness. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops, Proceedings*, Lecture Notes in Computer Science 1680, pp. 40–56.

Bošnački, D. and Dams, D. 1998. Integrating real time into Spin: A prototype implementation. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE/PSTV, Proceedings*. Kluwer, pp. 423–439.

Culler, D., Singh, J., and Gupta, A. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

Dams, D., Gerth, R., Knaack, B., and Kuiper, R. 1998. Partial-order reduction techniques for real-time model checking. *Formal Aspects of Computing*, 10(5–6):469–482.

Emerson, E., Jha, S., and Peled, D. 1997. Combining partial order and symmetry reductions. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97, Proceedings*, Lecture Notes in Computer Science 1217. Springer, pp. 19–34.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Lecture Notes in Computer Science 1032, Springer.

Godefroid, P. and Wolper, P. 1991. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K. Larsen and A. Skou, editors, *Computer Aided Verification, CAV'91, Proceedings*, Lecture Notes in Computer Science 575. Springer, pp. 332–342.

Holzmann, G. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall.

Holzmann, G. 1997. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295.

Holzmann, G. 1999. The engineering of a model checker: The Gnu i-Protocol case study revisited. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, Proceedings*, Lecture Notes in Computer Science 1680. Springer, pp. 232–244.

Holzmann, G., Godefroid, P., and Pirottin, D. 1992. Coverage preserving reduction strategies for reachability analysis. In R. Linn, Jr. and M. Uyar, editors, *Protocol Specification, Testing and Verification, XII, Proceedings*. Elsevier, pp. 349–363.

Holzmann, G. and Peled, D. 1995. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *Formal Descriptions Techniques VII, FORTE'94, Proceedings*. Chapman & Hall, pp. 197–211.

Kamel, M. and Leue, S. 2000. VIP: A visual editor and compiler for v-Promela. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Proceedings*, Lecture Notes in Computer Science 1785. Springer, pp. 471–486.

Koymans, C. and Mulder, J. 1990. A modular approach to protocol verification using process algebra. In J. Baeten, editor, *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, pp. 261–306.

Kunz, T. and Black, J. 1995. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527.

Leue, S. and Holzmann, G. 1999. V-Promela: A visual, object-oriented language for Spin. In *Object-Oriented Real-Time Distributed Computing, ISORC'99, Proceedings*. IEEE Computer Society Press, pp. 14–23.

Manna, Z. and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer.

Minea, M. 1999. Partial order reduction for model checking of timed automata. In J. Baeten and S. Mauw, editors, *Concurrency Theory, CONCUR'99, Proceedings*, Lecture Notes in Computer Science 1664. Springer, pp. 431–446.

Möller, M. and Alur, R. 2001. Heuristics for hierarchical partitioning with applications to model checking. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Proceedings*, Lecture Notes in Computer Science 2144. Springer, pp. 71–85.

Nalumasu, R. and Gopalakrishan, G. 2002. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247.

Overman, W. 1981. Verification of concurrent systems: Function and timing. Ph.D. thesis, UCLA, Los Angeles.

Peled, D. 1994. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *Computer Aided Verification, CAV'94, Proceedings*, Lecture Notes in Computer Science 818. Springer, pp. 377–390.

SPIN 2004. http://spinroot.com/.

Valmari, A. 1991. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, Lecture Notes in Computer Science 483. Springer, pp. 491–515.

Valmari, A. 1992. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322.

Willems, B. and Wolper, P. 1996. Partial-order methods for model checking: From linear time to branching time. In *Logic in Computer Science, LICS'96, Proceedings*. IEEE Computer Society Press, pp. 294–303.