Enhancing Partial-Order Reduction via Process Clustering

Twan Basten
Dept. of Electrical Eng., Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

Dragan Bošnački
Dept. of Computing Science, Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
e-mail: dragan@win.tue.nl

Abstract

Partial-order reduction is a well-known technique to cope with the state-space-explosion problem in the verification of concurrent systems. Using the hierarchical structure of concurrent systems, we present an enhancement of the partial-order-reduction scheme of [12, 19]. A prototype of the new algorithm has been implemented on top of the verification tool SPIN. The first experimental results are encouraging.

Key words: 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 explosion, caused by the arbitrary interleaving of independent actions of the various components of the system. Several techniques have been developed to cope with this problem. *Partial-order reduction* is a very prominent one (see, for example, [1, 7, 8, 11, 12, 18, 19, 20, 21, 22]). 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, a subset of the enabled actions satisfying certain criteria is chosen for further exploration. Following [12, 19], we call these sets *ample* sets.

The traditional approach to partial-order reduction (see, for example, [12, 19]) 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.

Our starting point is the partial-order algorithm of Holzmann and Peled [12, 19]. This algorithm is implemented in the verification tool SPIN [9] and has proven to be successful in coping with the state-space explosion. It is also sufficiently flexible to allow extensions (see for instance the extensions for timed systems in [4, 17]). 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. Our idea is 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 clustering 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 [12, 19]. It can also be seen as a version of the algorithm of Overman [18], adapted for clustering hierarchies and LTL model checking. We implemented our algorithm on top of the verification tool SPIN. The results obtained with the prototype are encouraging, in particular, considering that a visual language is being developed for SPIN [15] that combines naturally with our cluster-based reduction algorithm.

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 [12, 19]. In Section 4, we present our cluster-based partial-order-reduction algorithm. Section 5 gives experimental results. Finally, Section 6 contains concluding remarks.

Acknowledgments. Our work is inspired by the Next heuristic of Rajeev Alur and Bow-Yaw Wang presented in [2]. We thank Dennis Dams and Jeroen Rutten for their contributions.

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 `example`. 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 `example`.

```
System example

P0:   a0; a1;
P1:   b0; b1;
P2:   c0; c1;
```

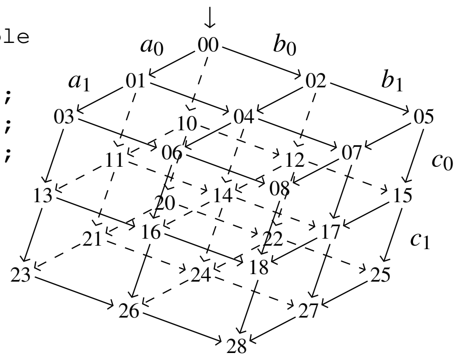


Figure 1. A simple concurrent system.

The numbers 00 through 08, 10 through 18, and 20 through 28 represent states of system `example`. The states encode all relevant information of `example` such as values of variables and local program counters. The initial state of the system is 00 (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 `example` can only be in three different states. However, Figure 1 shows that the complete state space of `example` consists of 27 (=3^3) states. The state space of a system with a fourth process (executing two actions) already consist 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. Obviously, the state-space explosion complicates verification techniques that are based on an exhaustive traversal of the entire state space of a concurrent system.

Labeled transition systems. The notion of a state space is the most 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. Propositions and state labels are included in the definition 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 $s \xrightarrow{a}$ iff $\tau(s, a)$ 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 iff *enabled*$(s) = \emptyset$.

In the previous paragraph, 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.

An *execution sequence* of LTS $\mathcal{T}$ is a (finite) sequence of subsequent transitions in T. Formally, for any natural number $n \in \mathbb{N}$, states $s_i \in S$ with $i \in \mathbb{N}$ and $0 \le i \le n$, and actions $a_i \in A$ with $i \in \mathbb{N}$ and $0 \le i < n$, the sequence $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \ldots s_{n-1} \overset{a_{n-1}}{\to} s_n$ is an execution sequence of length n of $\mathcal{T}$ iff $s_i \overset{a_i}{\to} s_{i+1}$ for all $i \in \mathbb{N}$ with $0 \le i < n$. State s_n is said to be *reachable* from s_0. A state is reachable in $\mathcal{T}$ iff it is reachable from $\hat{s}$. The LTS of Figure 1 has 90 execution sequences of length six all starting from the initial state and leading to deadlock state 28.

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 useful to improve these verification techniques. 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 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 is essentially also verified by means of a state-space traversal. To verify whether a system state satisfies a local property, it is important to encode all relevant information concerning the property in the state labeling of the LTS representing the state space of the system. Thus, 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. For more details on the verification of local properties, see [7, 11, 20].

The third class of properties are those expressible in (next-time-free) Linear-time Temporal Logic (LTL). Also LTL properties are 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 [16]. 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, [12].

3 Partial-Order Reduction

Section 3.1 gives results, known from the literature on partial-order reduction (see, for example, [1, 7, 8, 11, 12, 18, 19, 20, 21, 22]), that are needed to prove that our reduction technique preserves deadlocks, local properties, and next-time-free LTL. Section 3.2 presents the partial-order-reduction algorithm of Holzmann and Peled [12, 19]. 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. This technique 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 $\tau_r(s, a) = \tau(s, a)$.

It may be clear that not all reductions preserve all properties of interest. Thus, depending on the properties that a reduction 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 iff, 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* iff, for all states $s \in S$ with $s \overset{a}{\to}$ and $s \overset{b}{\to}$,

- $\tau(s, a) \overset{b}{\to}$ and $\tau(s, b) \overset{a}{\to}$, and
- $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$.

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 $r(s) = \emptyset$ iff $enabled(s) = \emptyset$.

C1 **(persistency)** For any $s \in S$ and execution sequence $s = s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n$ 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 (Deadlock preservation [7, 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 [7] 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, [7] uses a slightly stronger definition of independence. However, the proof of Theorem 4.3 in [7] carries over to our setting. Finally, several authors presented state-space-reduction algorithms that preserve deadlocks [8, 18, 20].

Considering Figure 1, it is easy to define a reduction function satisfying provisos C0 and C1 that reduces the LTS of Figure 1 to a single execution sequence from state 00 to state 28. Clearly, this reduction preserves deadlock state 28.

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: That is, a property ϕ is *local* iff, for all states $s \in S$ and *independent* actions $a, b \in A$ such that $s \stackrel{a}{\to}$, $s \stackrel{b}{\to}$, and ϕ has different truth values in states s and $\tau(s, a)$, the truth values of ϕ in s and in $\tau(s, b)$ are the same. An LTS satisfies a local property ϕ iff there is a reachable state that satisfies ϕ. Typical examples of local properties are properties that depend only on the state of a single 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* $s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n = s_0$ of length $n \in \mathbb{N} \setminus \{0\}$, there is an $i \in \mathbb{N}$ with $0 \leq i < n$ such that $r(s_i) = enabled(s_i)$.

Theorem 3.4 (Local-property preservation) 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 iff the reduced LTS $\mathcal{T}_r$ satisfies ϕ.

Proviso C2 prevents the so-called 'ignoring problem' identified in [20]. Informally, this problem occurs when a reduction of a state space ignores the actions of an entire process. Proofs of (variants of) Theorem 3.4 can be found in [7, 11, 20]. 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, $prop(\phi)$ is the set of propositions in ϕ.

Definition 3.5 (Invisibility) An action $a \in A$ is *ϕ-invisible* in state $s \in S$ iff $\tau(s, a)$ is undefined or, for all $\pi \in prop(\phi)$, $\pi \in L(s) \Leftrightarrow \pi \in L(\tau(s, a))$. Action a is *globally ϕ-invisible* iff it is ϕ-invisible for all $s \in S$.

Informally, an action is globally ϕ-invisible iff 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 $r(s) = enabled(s)$.

Theorem 3.6 (Next-time-free LTL preservation [19, 21]**)** 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 iff the reduced LTS $\mathcal{T}_r$ satisfies ϕ. (See [16] 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 [12, 19]. 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 [12, 19]. 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 $P \neq Q$, $P \cap Q = \emptyset$. Function $Pid : A \rightarrow \mathcal{P}$ gives for each action the process it is contained in. Not every partitioning of actions is a meaningful process structure. Concurrency within processes is not allowed. 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 $a, b \in enabled(s)$, $b \notin enabled(\tau(s, a))$. 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 following definition is crucial in the formulation of the abovementioned reduction function. It depends on the class of properties one is interested in.

Definition 3.7 (safety) An action $a \in A$ is *safe* iff it is independent from any (other) action $b \in A$ with $Pid(b) \neq Pid(a)$. An action $a \in A$ is *safe for a given next-time-free LTL formula* ϕ iff it is independent from any action $b \in A$ with $Pid(b) \neq Pid(a)$ and globally ϕ-invisible.

As mentioned, the algorithm of [12, 19] 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 3.8 (Reduction function *ample***)** Let $s \in S$. Consider the set SP of processes $P \in \mathcal{P}$ such that $enabled(s) \cap P \neq \emptyset$, for all $a \in enabled(s) \cap P$, a is safe (for some next-time-free LTL formula ϕ), and $\tau(s, a)$ is not on the DFS stack. If SP is empty, then define $ample(s) = enabled(s)$; otherwise, choose an arbitrary element P of SP and define $ample(s) = enabled(s) \cap P$. Set $ample(s)$ is said to be the ample set for s.

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 3.8. C1 and C3 follow from the safety requirement in Definition 3.8. Finally, C2 follows from the requirement in Definition 3.8 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 3.4), and next-time-free LTL (Theorem 3.6).

3.3 Implementation in SPIN

SPIN [9] is a tool supporting the automatic verification of deadlock-, local-, and next-time-free LTL properties. Specifications of concurrent system are written in the language PROMELA. The partial-order-reduction algorithm of [12, 19] 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 from the PROMELA specification before starting the DFS (see [12]). 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. Another good reference for readers interested in implementation issues concerning partial-order reduction is [7].

4 Cluster-based Partial-Order Reduction

We motivate our improvement of partial-order reduction by means of two different specifications of the same concurrent system given in Figure 2. 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 3.7 means that none of the actions is safe. Thus, reduction function *ample* of Definition 3.8 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 2 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 3.7 and 3.8 to clusters. Thus, it is possible to reduce the state space of the system of Figure 2 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 2), while preserving all properties of interest.

Let $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$ be an LTS with processes $\mathcal{P}$.

Definition 4.1 (Clustering) A *cluster* of processes in $\mathcal{P}$ is simply a set of processes. A *clustering* $\mathcal{C} \subseteq 2^{\mathcal{P}}$ of $\mathcal{P}$ is a set of clusters partitioning $\mathcal{P}$, i.e., $\mathcal{P} = \cup_{C \in \mathcal{C}} C$ and, for all $C, D \in \mathcal{C}$ with $C \neq D$, $C \cap D = \emptyset$. A *clustering hierarchy* $\mathcal{H}$ for $\mathcal{P}$ is a finite ordered set $\{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{n-1}\}$, where $n \in \mathbb{N} \setminus \{0\}$, of clusterings of $\mathcal{P}$ such that, for all $i \in \mathbb{N}$ with $0 < i < n$,

- $|\mathcal{C}_i| < |\mathcal{C}_{i-1}|$ and
- for each $C \in \mathcal{C}_{i-1}$, there exists a $D \in \mathcal{C}_i$ such that $C \subseteq D$.

For any $i \in \mathbb{N}$ with $0 \leq i < n$, clustering $\mathcal{C}_i$ is called *level*

```
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);
```

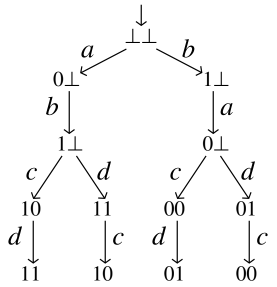


Figure 2. Two specifications of a concurrent system and a reduced state space.

i of the hierarchy. Level $\mathcal{C}_i$ is *above* level $\mathcal{C}_j$ iff $i > j$; it is *below* $\mathcal{C}_j$ iff $i < j$.

Clearly, Definition 4.1 implies that each level in a clustering hierarchy is a coarsening of all lower levels. Also note that the maximum number of levels in a clustering hierarchy is limited by the number of processes in $\mathcal{P}$. A clustering hierarchy for `example2` consisting of three levels (numbered 0, 1, and 2) is the following: $\{\{\{\mathtt{P0}\},\{\mathtt{P1}\},\{\mathtt{P2}\},\{\mathtt{P3}\}\},\{\mathtt{C0} = \{\mathtt{P0},\mathtt{P1}\},\mathtt{C1} = \{\mathtt{P2},\mathtt{P3}\}\},\{\{\mathtt{P0},\mathtt{P1},\mathtt{P2},\mathtt{P3}\}\}\}$.

Let $\mathcal{H} = \{\mathcal{C}_0, \ldots, \mathcal{C}_{n-1}\}$, with $n \in \mathbb{N} \setminus \{0\}$, be a clustering hierarchy for $\mathcal{P}$. For all $i \in \mathbb{N}$ with $0 \leq i < n$, function $Cid^{(i)} : \mathcal{P} \to \mathcal{C}_i$ yields for a given process the level-i cluster it belongs to; that is, given a process $P \in \mathcal{P}$ and level $i \in \mathbb{N}$ with $0 \leq i < n$, $Cid^{(i)}(P) = C$ with $C \in \mathcal{C}_i$ the unique cluster such that $P \in C$.

Definition 4.2 (Level-*i* safety) Let $i \in \mathbb{N}$ with $0 \leq i < n$. Action $a \in A$ is *level-i safe* (for a given next-time-free LTL formula ϕ) iff it is independent of any action $b \in A$ with $Cid^{(i)}(Pid(b)) \neq Cid^{(i)}(Pid(a))$ (and globally ϕ-invisible).

In the above clustering hierarchy for `example2`, all actions are level-1 and (trivially) level-2 safe, but not level-0 safe.

Given a cluster $C \subseteq \mathcal{P}$, let $actions(C) = \cup_{P \in C} P$. Furthermore, for any state $s \in S$, let $enabled(s, C) = actions(C) \cap enabled(s)$.

Definition 4.3 (Reduction function *cample*) Let $s \in S$. Let, for each level $\mathcal{C}_i \in \mathcal{H}$, SC_i be the set of clusters $C \in \mathcal{C}_i$ such that $enabled(s, C) \neq \emptyset$, for all $a \in enabled(s, C)$, a is level-i safe (for some next-time-free LTL formula ϕ), and $\tau(s, a)$ is not on the DFS stack. If the set SC_i is empty for all levels $\mathcal{C}_i$, then define $cample(s) = enabled(s)$; otherwise, choose a level $\mathcal{C}_i$ such that the set SC_i is nonempty, select an arbitrary element C of SC_i, and define $cample(s) = enabled(s, C)$.

It is interesting to observe that function *ample* of Definition 3.8 is a special case of function *cample* of Definition 4.3 with a trivial hierarchy of only one level that consists of trivial clusters each containing only one process.

Theorem 4.4 The reduction of an LTS obtained via reduction function *cample* of Definition 4.3 preserves deadlock-, local-, 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 3.8 satisfies these provisos. The desired result follows from Theorems 3.3, 3.4, and 3.6. □

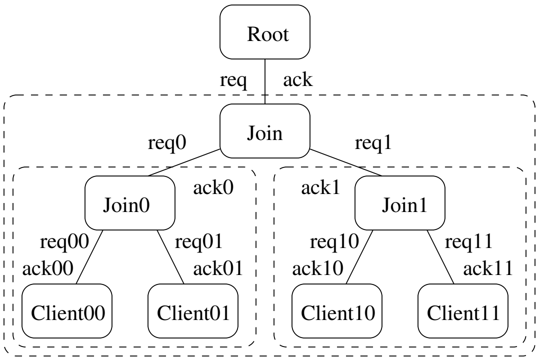
Consider again the concurrent system of Figure 2. The figure shows a reduced state space of this system. In each state, the values of variables `u` and `v` are given with $\perp$ meaning that a value is undefined. The sets of actions chosen in each state are ample sets as defined by function *cample* of Definition 4.3. 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.)

In practice, the largest state-space reductions are obtained by choosing the ample sets as small as possible. A simple way to obtain small ample sets is to search the levels in a clustering hierarchy for suitable clusters in increasing order.

Another practical issue is how to obtain useful clustering hierarchies. 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. 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 [14].

	without POR			standard POR			cluster-based POR				
N	states	trans	time	states	trans	time	states	%	trans	%	time
2	65	108	0.1	60	76	<0.1	30	50	30	61	<0.1
3	329	784	0.1	304	480	<0.1	66	78	66	86	<0.1
4	1657	5216	0.1	1532	2908	0.1	138	91	138	95	<0.1
5	8313	32680	0.9	7688	17064	0.7	282	96	282	98	<0.1

Table 1. Results for the best-case example.



level	clusters
2	{C00, C01, C10, C11, J0, J1, J}, {R}
1	{C00, C01, J0}, {C10, C11, J1}, {J}, {R}
0	{C00}, {C01}, {C10}, {C11}, {J0}, {J1}, {J}, {R}

Figure 3. The Parity Computer

5 Experiments

To validate our cluster-based reduction algorithm, we implemented a prototype on top of the verification tool SPIN, version 3.2.4. We applied our prototype implementation to several examples. Since PROMELA, the input language of SPIN, does not support modular design, we provided the clustering hierarchies ourselves (based on a straightforward informal analysis of the examples). Our focus is on the generation of state spaces; we did not verify any properties. The goal of the experiments 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 cases, we observe significantly larger reductions than the ones obtained with standard partial-order reduction. Moreover, in all cases, our algorithm reduces verification times. The overhead of upgrading the 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 three case studies. The experiments were performed on a Sun Ultra-10 machine, with a 299 MHz UltraSPARC-IIi processor and 128 MB of main memory, running the SunOS 5.6 operating system. All the verification times are given in seconds.

Best-case example. Our first case study consists of variants of system `example2` of Figure 2. The systems we verified consist of N pairs of processes and N variables, each pair sharing one of these variables. The system with N=2 corresponds to system `example2`. 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. We used a hierarchy of two layers, with the nontrivial layer consisting of clusters that coincided with process pairs.

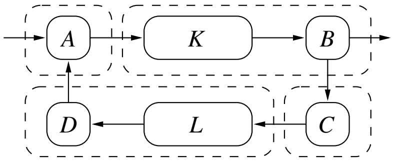
Parity Computer. The second example, taken from [2], models a Parity Computer with a tree structure. The system consists of a root module, N client modules as leaves, and join modules as intermediate nodes. The system with N=4 is shown in Figure 3. The figure also shows a clustering hierarchy that is immediately derived from the modular structure of the Parity Computer.

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, simultaneously, sending an acknowledgment to its children. Eventually, parity bits are delivered to

	standard POR			cluster-based POR				
N	states	trans	time	states	%	trans	%	time
4	1749	4798	0.1	1294	26	2352	51	0.1
5	7933	27012	0.8	3938	50	6775	75	0.3
6	69615	288678	10.0	21620	69	38372	86	0.6
7	320095	1213520	47.3	25228	92	44730	98	1.6
8[a]	2782640	15381300	621.2	30377	99	55828	>99	2.1

[a] Because of the large memory requirements, this experiment was performed on a Sun Ultra-Enterprise machine with three 248 MHz UltraSPARC-II processors and 2304 MB of main memory, running the SunOS 5.5.1 operating system.

Table 2. Results for the Parity Computer.



standard POR			cluster-based POR				
states	trans	time	states	%	trans	%	time
16384	67073	3.6	12897	21	44460	34	1.1

Figure 4. Concurrent Alternating-Bit Protocol

the root process.

Consider again the example in Figure 3. Our cluster-based partial-order-reduction algorithm takes advantage of the fact that a join process communicates with its parent and its 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.

The reduction with cluster-based partial-order reduction is slightly worse (though of the same order of magnitude) than the reduction reported in [2] for the same examples, obtained with the Next heuristic. However, it is difficult to draw any final conclusions about the comparative efficiency of the two techniques based only on this one example. First, the input languages in which the models are specified are different, which inevitably leads to differences in the model. Second, the Parity-Computer example is one of the best cases for the Next heuristic and, as the authors note themselves in [2], there are many examples for which partial-order reduction gives better results than the Next heuristic.

Concurrent Alternating-Bit Protocol. Finally, we consider the Concurrent Alternating-Bit Protocol (CABP) of [13]. The CABP has six components, as depicted in Figure 4. 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. A does not wait for a negative response before retransmitting. Channel K is unreliable in the sense that it can 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 in Figure 4 show. The hierarchy consists of two levels, with nontrivial level $\{A\}$, $\{K, B\}$, $\{C\}$, and $\{L, D\}$. The implementation of A and C is such that each action in these processes depends on both its neighbors. Thus, any other process clustering does not improve the results.

6 Conclusion

The main contribution of this paper is an enhancement of the partial-order-reduction scheme of [12, 19]. Using the inherent structure of concurrent systems, we improve the way the safety (i.e., independence and invisibility) of actions is determined syntactically during the compilation of the system specification. The resulting ample sets may contain actions from more than one process. Although ample sets with actions from several processes have been considered earlier (e.g., [1]), to the best of our knowledge, the idea of exploiting hierarchical system structure is not present in the literature. We implemented our algorithm on top of SPIN, by upgrading SPIN's standard partial-order-reduction engine. The prototype implementation has been tested on several examples known from the literature and 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.

It will be interesting to see how our approach works in combination with other state-space-reduction heuristics. Following [6], 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 [2] 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.

The main task in the near future is to fully automate our implementation. In the current prototype, the clustering hierarchy and safety levels must be included manually. A fully automatic implementation will allow us to test the implementation on larger, real-world examples. We also plan to take advantage of the improved partial-order reduction [10] introduced in the latest releases of SPIN, as well as the introduction of V-Promela [15]. Another interesting topic is the study of clustering heuristics (see, e.g., [14]). Good clustering heuristics might actually yield better clustering hierarchies than the ones obtained from the hierarchical structure specified by a system designer. Finally, our cluster-based algorithm is compatible with the upgrade of SPIN's engine for timed systems from [4]. It is very likely that it can be combined with techniques for partial-order reduction for timed automata [3, 5, 17].

References

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. In O. Grumberg, editor, *Computer Aided Verification, CAV '97*, LNCS 1254, pages 340–351. Springer, 1997.

[2] R. Alur and B.-Y. Wang. "Next" Heuristic for On-the-fly Model Checking,. In J.C.M. Baeten and S. Mauw, editors, *Concurrency Theory, CONCUR '99*, LNCS 1664, pages 98–113. Springer, 1999.

[3] J. Bengtsson, B. Jonsson, B. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In D. Sangiorgi and R. de Simone, editors, *Concurrency Theory, CONCUR '98*, LNCS 1466, pages 485–501. Springer, 1998.

[4] D. Bošnački and D. Dams. 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*, pages 423–439. Kluwer, 1998.

[5] D. Dams, R. Gerth, B. Knaack, and R. Kuiper. Partial-order Reduction Techniques for Real-time Model Checking. *Formal Aspects of Computing*, 10(5–6):469–482, 1998.

[6] E.A. Emerson, S. Jha, and D. Peled. Combining Partial Order and Symmetry Reductions. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '97*, LNCS 1217, pages 19–34. Springer, 1997.

[7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer, 1996.

[8] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, CAV '91*, LNCS 575, pages 332–342. Springer, 1991.

[9] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[10] G.J. Holzmann. 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*, LNCS 1680. Springer, 1999.

[11] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In R.J. Linn, Jr. and M.Ü. Uyar, editors, *Protocol Specification, Testing and Verification, XII*, pages 349–363. Elsevier, 1992.

[12] G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In D. Hogrefe and S. Leue, editors, *Formal Descriptions Techniques VII, FORTE '94*, pages 197–211. Chapman & Hall, 1995.

[13] C.P.J. Koymans and J.C. Mulder. A Modular Approach to Protocol Verification Using Process Algebra. In J.C.M. Baeten, editor, *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, pages 261–306. Cambridge University Press, 1990.

[14] T. Kunz and J.P. Black. Using Automatic Process Clustering for Design Recovery and Distributed Debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, 1995.

[15] S. Leue and G.J. Holzmann. V-Promela: A Visual, Object-Oriented Language for Spin. In *Object-Oriented Real-Time Distributed Computing, ISORC '99*, pages 14–23. IEEE Computer Society Press, 1999.

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

[17] M. Minea. Partial Order reduction for Model Checking of Timed Automata. In J.C.M. Baeten and S. Mauw, editors, *Concurrency Theory, CONCUR '99*, LNCS 1664, pages 431–446. Springer, 1999.

[18] W.T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, UCLA, Los Angeles, 1981.

[19] D. Peled. Combining Partial Order Reductions with On-the-fly Model Checking. In D.L. Dill, editor, *Computer Aided Verification, CAV '94*, LNCS 818, pages 377–390. Springer, 1994.

[20] A. Valmari. Stubborn Sets for Reduced State Space Generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, LNCS 483, pages 491–515. Springer, 1991.

[21] A. Valmari. A Stubborn Attack on State Explosion. *Formal Methods in System Design*, 1:297–322, 1992.

[22] B. Willems and P. Wolper. Partial-Order Methods for Model Checking: From Linear Time to Branching Time. In *Logic in Computer Science, LICS '96*, pages 294–303. IEEE Computer Society Press, 1996.