Refining Dependencies Improves Partial-Order Verification Methods* (Extended Abstract)

Patrice Godefroid and Didier Pirottin

Université de Liège, Institut Montefiore B28, 4000 Liège Sart-Tilman, Belgium.
Email : {god,pirottin}@montefiore.ulg.ac.be

Abstract. Partial-order verification methods exploit "independency" between transitions of a concurrent program to avoid parts of the state space explosion due to the modeling of concurrency by interleaving. In this paper, we study the influence of refining dependencies between transitions of the program on the effectiveness of these methods. We show that carefully tracking dependencies can yield substantial improvements for their performances. For instance, we were able to decrease the memory requirements needed for the verification of a real-size protocol with such a method from a factor of 5 to a factor of 25 by only refining dependencies.

1 Introduction

The effectiveness of state-space exploration techniques for debugging and proving correct concurrent reactive systems is increasingly becoming established as tools are being developed. The number of "success stories" about applying these techniques to industrial-size systems keeps growing. The reason for which these techniques are so successful is mainly due to their simplicity: they are easy to understand, easy to implement and, last but not least, easy to use: they are fully automatic. Moreover, the range of properties that they can verify has been substantially broaden in the last decade thanks to the development of model-checking methods for various temporal logics.

The only real limit of state-space exploration verification techniques is the often excessive size of the state space. This observation has triggered the development of new methods to overcome this serious problem which limits both the applicability and the efficiency of the approach. Some of them tackle the *effects* of state explosion (e.g. "on the fly" methods [JJ89], bit-state hashing [Hol88], state-space caching [GHP92], state-compression techniques [HGP92]) while others rather tackle its *causes*.

Partial-order verification methods (e.g. [Val91, God90, PL90, McM92]) fit in the second category. Their aim is to avoid the part of the state explosion due to the modeling of concurrency by interleaving. With these methods, the behavior of the program

* This work was partially supported by the European Community ESPRIT BRA project REACT (6021) and by the Belgian Incentive Program "Information Technology" – Computer Science of the future, initiated by the Belgian State – Prime Minister's Service – Science Policy Office. The scientific responsibility is assumed by its authors.

being verified is described, explicitly [PL90, McM92] or implicitly [Val91, God90], by means of partial orders of transitions, rather than by sequences of transitions.

In this paper, we follow the approach of [God90, GW91b, HGP92] and use Mazurkiewicz's traces as a semantic model [Maz86]. Mazurkiewicz's traces are defined as equivalence classes of sequences of transitions that the system being analyzed can perform from its initial state. Two sequences of transitions are equivalent (belong to the same trace) if they can be obtained from each other by successively permuting adjacent symbols which are "independent". Intuitively, "independent" means that their occurrence does not affect each other. A trace corresponds to a partial order of transition occurrences: the set of all sequences belonging to the trace is the set of all linearizations of this partial order. If two independent transitions occur next to each other in a sequence of a trace, the order of their occurrence is irrelevant since they are unordered in the partial order corresponding to that trace.

Loosely speaking, the basic idea behind partial-order verification methods like the ones of [God90, GW91b, HGP92] can be stated as follows: given a *dependency relation* $D \subseteq T \times T$ on the set T of transitions in the program P, it is possible to verify *properties* of the program P by exploring *only one sequence* of each trace (partial order of transitions) the system can perform from its initial state. Thus, there are basically three questions that have to be answered in order to fully characterize such partial-order verification methods:

1. How does one minimize the number of sequences explored while exploring at least one sequence per trace?
2. Which properties can be verified?
3. Which dependency relation is used?

Several techniques that answer the first question [Ove81, Val91, GW91b] and the second one [Val90, GW91a, HGP92, Pel92] have been proposed and studied. In contrast, the third question has been little investigated so far. Existing related work deals essentially with semantic issues about various formal definitions of dependency (e.g. [KP92, Och90]).

In this paper, we address the third question mentioned above from a pragmatic point of view. Given a concurrent program, we study which dependency relation should be used in order to maximize the efficiency of partial-order verification algorithms. Our idea is that, by refining dependencies, one can increase the number of independent transitions, decrease the number of traces and thus improve the performances of the verification since the number of sequences that have to be explored (at least one per trace) can decrease. This can be done independently of the partial-order verification algorithm being used to perform the state-space exploration and independently of the type of property being checked.

2 Program and Semantics

Consider a program P describing a system composed of n interacting concurrent processes P_i. Each process P_i is represented by a transition system A_i. Formally, a transition system is a tuple $A = (\Sigma, S, \Delta, s_0)$, where Σ is an alphabet of commands, S is a finite set of states, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation, and $s_0 \in S$ is the initial state.

We assume that processes can communicate asynchronously with each other by performing *operations* on *shared objects*. Each object consists of a value and a set of

operators that test and/or modify the value of the object. Formally, we define an object O as a tuple $O = (Dom, OP)$, where Dom is the set of all possible values for the object, and OP is the set of operations that can be performed on the object. Each operation $op_i \in OP$ is a function $IN_i \times Dom \rightarrow OUT_i \times Dom$, where IN_i represents the set of possible inputs and OUT_i the set of possible outputs of this operation. Note that an operation on an object may not necessarily be defined for all inputs $in \in IN_i$ and for all values $v \in Dom$ of the object. $op_i(in, v) \rightarrow (out, v')$ will denote the execution of the operation $op_i \in OP$ with $in \in IN_i$ as input, $out \in OUT_i$ as output, while $v \in Dom$ and $v' \in Dom$ will represent the value of the object respectively before and after the execution of op_i. If an operation takes no input or returns no output, we will use "–" instead of *in* or *out* to denote that the input or output value is not meaningful.

Example 1. Consider an object whose domain Dom is the set of the integers. We define two operations *Read* and *Write* on this object as follows. Let $v \in Dom$,

- $Read(-, v) \rightarrow (v, v)$: a *Read* operation takes no input and returns the value of the object. A *Read* operation is always defined.
- $Write(v', v) \rightarrow (-, v')$: a *Write* operation takes $v' \in Dom$ as input and sets the value of the object to v'. It has no output. A *Write* operation is defined for any input $v' \in Dom$.

A transition of a process P_i is defined as a *guarded command* (*cond* $\rightarrow$ *actions*), where *cond* is a conjunction of boolean expressions c_j, and *actions* denotes a sequence of operations on objects. We assume that the truth value of *cond* in a given state determines if the transition is enabled or not in that state. We also assume that, for each operation *op* that appears in the *actions* part of the guarded command, if *op* is not defined for all inputs and values of the object, there is a condition c_j (expressed by using operations on the object) in the *cond* part of the guarded command such that *op* is defined iff c_j is true. For the sake of uniformity and to simplify what follows, the state of each process P_i is represented by an object "program counter" PC_i (for example, PC_i can be viewed as an integer variable). Each transition $t = (s, a, s')$ of P_i has a condition c_j in its *cond* part which is evaluated by an operation on PC_i (a *Read* operation), such that c_j is true iff the value of the program counter PC_i corresponds to state s. Similarly, the execution of an enabled transition of P_i involves an operation on PC_i (a *Write* operation). In the sequel, all operations that appear either in the *cond* part or the *actions* part of a transition are said "to be used by" this transition.

A transition system $A_P = (\Sigma_P, S_P, \Delta_P, s_{0P})$ representing the joint global behavior of the processes P_i can be computed by simulating all possible sequences of transitions the system can perform from its initial state. Σ_P is the set of transitions that appear in the code of the program P, S_P is the set of states that the system can reach from the initial state s_{0P}, and each transition of Δ_P corresponds to a transition between two states that the system can perform by executing a single transition of one of the processes of the program[2]. A_P is called the "state space" of the program P.

In practice, the limits of verification methods based on state-space exploration techniques come essentially from the often excessive size of the state space A_P. However, experiments with partial-order verification methods have shown that most of this explosion can be avoided. Therefore, let us turn to partial-order semantics, namely, Mazurkiewicz's traces [Maz86].

Traces are defined as equivalence classes of sequences of transitions. Given a set T and a symmetrical binary relation $D \subseteq T \times T$ called dependency relation, two sequences

[2] In the case of synchronous communications, a transition of A_P may correspond to the execution (synchronization) of several transitions of different processes. We do not consider this case here for the sake of simplicity.

over T belong to the same trace with respect to D (are in the same equivalence class) if they can be obtained from each other by successively permuting adjacent symbols which are not dependent, i.e. independent, according to D. For instance, if t_1 and t_2 are two transitions of T which are independent according to D, the sequences t_1t_2 and t_2t_1 belong to the same trace. A trace is fully characterized by only one of its sequences. Therefore, a trace is usually represented by one of its elements enclosed within brackets and, when necessary, subscripted by the alphabet and the dependency relation. Thus the trace containing both t_1t_2 and t_2t_1 could be represented by $[t_1t_2]_{(T,D)}$. A trace corresponds to a partial order of transition occurrences: the set of all sequences belonging to the trace corresponds to the set of all linearizations of this partial order.

3 Dependency Relations

In the context considered here, the set T is defined as the union of the sets Δ_i of transitions of the processes P_i. In other words, T corresponds to the set of *transitions* that appear in the code of the program. The following definition (adapted from [KP92]) characterizes the properties of possible "valid" dependency relations for the transitions of a given program P:

Definition 1. Let T be the set of transitions in the program P and $D \subseteq T \times T$ be a symmetrical binary relation. D is a *valid dependency relation* for P iff for all $t_1, t_2 \in T$, $(t_1, t_2) \notin D$ (t_1 and t_2 are independent) implies that the two following properties hold for all states $s \in S_P$ of A_P:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then $s \overset{t_1t_2}{\Rightarrow} s'$ and $s \overset{t_2t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

To determine if two given transitions are dependent or not, it is not practical to enumerate all reachable states of the program and to check the two above properties for all these states and for all pairs of transitions. Fortunately, it is possible to define a valid dependency relation between transitions by considering the operations on shared objects they perform.

We first define a dependency relation between the operations on an object as follows:

Definition 2. Let $O = (Dom, OP)$ be an object and $D_O \subseteq OP \times OP$ be a symmetrical binary relation. D_O is a *valid dependency relation* for O iff for all $op_1, op_2 \in OP$, $(op_1, op_2) \notin D_O$ (op_1 and op_2 are independent) implies that the two following properties hold for all values $v \in Dom$, and for all inputs in_1 and in_2:

1. if $op_1(in_1, v)$ is defined, with $op_1(in_1, v) \rightarrow (out_1, v_1')$, then $op_2(in_2, v)$ is defined iff $op_2(in_2, v_1')$ is defined; and
2. if $op_1(in_1, v)$ and $op_2(in_2, v)$ are defined, then $\exists out1, out2, v_1', v_2', v''$ such that:
   - $op_1(in_1, v) \rightarrow (out_1, v_1')$ and $op_2(in_2, v_1') \rightarrow (out_2, v'')$
   - $op_2(in_2, v) \rightarrow (out_2, v_2')$ and $op_1(in_1, v_2') \rightarrow (out_1, v'')$

   (commutativity of operations, with the same outputs)

Example 2. Consider again the example of an object representing an integer value. A valid dependency relation between the operations on this object is given in the following table, where "+" means that operations are dependent, while "–" denotes the fact that operations are independent:

DEP.	*Write*	*Read*
Write	+	+
Read	+	-

Two *Write* operations are dependent because they can leave the object with different values depending on the order of their execution. A *Read* and a *Write* operations are dependent because the output of the *Read* can be different depending on the order of execution of these operations. Two *Read* operations are independent because they are always defined and return the same output independently of the order of their execution.

Now, we can define a dependency relation between transitions of a program P from dependency relations between operations as follows:

Definition 3. Let T be the set of transitions in the program P. Two transitions $t_1, t_2 \in T$ are independent iff $\forall op_1$ used by t_1 and $\forall op_2$ used by t_2, if op_1 and op_2 are two operations on the same object, then op_1 and op_2 are independent.

One can easily check that the dependency relation on transitions obtained with this definition is a valid one.

4 Towards More Independency

Since partial-order verification methods have to explore at least one sequence per trace, the smaller is the number of traces, the smaller can be the number of explored sequences, and the more efficient can be the verification. One way to decrease the number of traces is to reduce the size of the dependency relation.

Theorem 4. *Let T be an alphabet, and $D \subseteq T \times T$ and $D' \subseteq T \times T$ be two valid dependency relations. Let L be a set of sequences over T, L_D and L'_D be the set of traces (equivalence classes) induced respectively by D and D' on L. If $D' \subset D$, then D' induces less traces on L than D: $|L'_D| \leq |L_D|$.*

Proof. Straightforward. (See full paper.)

It is therefore desirable to have as few dependencies as possible between transitions of a program, and thus between operations on objects, in order to improve the effectiveness of partial-order verification methods.

In practice, there are essentially two ways of refining dependencies between operations: by *refining the operations* themselves and by *using conditional dependency.*

Refining an operation op_i consists of splitting the operation viewed as a set of pairs $(IN_i \times Dom, OUT_i \times Dom)$ in several parts, and considering these different parts as being different operations, between which some independency may arise.

Example 3. Consider again the example of the object corresponding to an integer variable. We saw that, in general, two *Write* operations are dependent. But there are special cases of *Write* operations that can be considered as being independent: for instance, two incrementation operations *Incr*, formally defined by $Incr(-, v) \rightarrow (-, v + 1)$ (always defined), can be considered as being independent according to Definition 2. We obtain a new dependency relation:

DEP.	*Write*	*Incr*	*Read*
Write	+	+	+
Incr	+	-	+
Read	+	+	-

In the previous example, the new dependency relation obtained after refining the operation *Write* may yield less dependencies between the transitions of the program. It is thus preferable to use *Incr* rather than *Write* whenever possible. In practice, this can be done by adding the operation *Incr* to the protocol modeling language and by using it explicitly in the code of the program, or the verification tool could detect automatically when a *Write* operation actually performs an *Incr* operation.

The second way of refining dependency relations is to define them as being *conditional*: instead of defining a dependency relation that holds for all states $s \in S_P$, it is possible to define a dependency relation for each state individually. Definition 1 then becomes [KP92]:

Definition 5. Let T be the set of transitions in the program P and $D \subseteq T \times T \times S_P$. D is a *valid conditional dependency relation* for P iff for all $t_1, t_2 \in T$, $s \in S_P$, $(t_1, t_2, s) \notin D$ (t_1 and t_2 are independent in s) implies that $(t_2, t_1, s) \notin D$ and that the two following properties hold in state s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

Definitions 2 and 3 can be adapted in a similar way.

Example 4. Consider a bounded FIFO channel (buffer) of size N. The domain *Dom* of possible values for this object is the set of sequences of messages $M \cup M^2 \cup \ldots \cup M^N$, where M is the set of possible messages that can be transmitted via the channel. We define three operations on this object:

- $Send(v, v_1 v_2 \ldots v_n) \rightarrow (-, v_1 v_2 \ldots v_n v)$ defined if $n < N$ and $v \in M$,
- $Receive(-, v_1 v_2 \ldots v_n) \rightarrow (v_1, v_2 \ldots v_n)$ defined if $n > 0$,
- $Length(-, v_1 v_2 \ldots v_n) \rightarrow (n, v_1 v_2 \ldots v_n)$ always defined.

The following tables give respectively a constant and a conditional dependency relation between these operations. If the condition given in the row *op* and column *op'* of the table is true for the value $v \in Dom$ considered (n is the number of messages in the channel), then *op* and *op'* are dependent for v. Otherwise, they are independent. A "–" in the table represents a condition which is always false (operations always independent).

DEP.	*Send*	*Receive*	*Length*
Send	+	+	+
Receive	+	+	+
Length	+	+	–

DEP.	*Send*	*Receive*	*Length*
Send	$n < N$	$n = 0$ or $n = N$	$n < N$
Receive	$n = 0$ or $n = N$	$n > 0$	$n > 0$
Length	$n < N$	$n > 0$	–

Thanks to conditional dependency, operations that are dependent for some but not all values $v \in Dom$ are no more considered as dependent for all values.

We can still reduce dependencies between operations by simultaneously refining the operations and by using a conditional dependency.

Example 5. Consider the previous example. In real protocol models, the operation *Length* is often used to test if a channel is empty or full. Let us introduce two new operations *Empty* and *Full* defined as follows:

- $Empty(-, v_1 v_2 \ldots v_n) \rightarrow$ (if $(n = 0)$ then *true* else *false*, $v_1 v_2 \ldots v_n$) always defined.
- $Full(-, v_1 v_2 \ldots v_n) \rightarrow$ (if $(n = N)$ then *true* else *false*, $v_1 v_2 \ldots v_n$) always defined.

A new dependency relation can then be defined:

DEP.	*Send*	*Receive*	*Length*	*Empty*	*Full*
Send	$n < N$	$n = 0$ or $n = N$	$n < N$	$n = 0$	$n = N - 1$
Receive	$n = 0$ or $n = N$	$n > 0$	$n > 0$	$n = 1$	$n = N$
Length	$n < N$	$n > 0$	–	–	–
Empty	$n = 0$	$n = 1$	–	–	–
Full	$n = N - 1$	$n = N$	–	–	–

Note that, when using a conditional dependency relation, the definition of a trace has to be modified: a conditional trace is defined *with respect to a state* of A_P. Two sequences $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{i-1}} s_i \xrightarrow{a} s_{i+1} \xrightarrow{b} s_{i+2} \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ and $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{i-1}} s_i \xrightarrow{b} s'_{i+1} \xrightarrow{a} s'_{i+2} \ldots \xrightarrow{t_{n-1}} s'_n \xrightarrow{t_n} s'_{n+1}$ belong to the same "conditional trace from state s_0" iff a and b are independent in state s_i. Maybe surprisingly, a conditional trace does not necessarily correspond anymore to a partial order of transitions: the set of sequences in a trace does not always correspond to the set of all linearizations of a partial order [KP92]. However, the following theorem is still satisfied by conditional traces:

Theorem 6. *Consider a conditional trace* $[w]$ *from* $s \in A_P$. *If* $s \stackrel{w}{\Rightarrow} s'$, *then* $\forall w' \in [w]$: $s \stackrel{w'}{\Rightarrow} s'$.

Proof. Follows from Definition 5. (See full paper.)

Since the preservation of this theorem is the only assumption about traces which is needed in the sequel, we will not distinguish traces from conditional traces.

5 Using Refined Dependencies

In this section, we show how refined dependencies can be used by existing state-space exploration methods using partial-order techniques. These methods do not explore the whole state space of the program being verified, but only parts of it. They proceed as follows: at each state s reached during the search, they compute a subset T of the set of all enabled transitions and explore only the transitions of this subset T, the other enabled transitions are not explored.

Two main techniques have been proposed in the literature for computing such sets T. One of them is the sleep set technique (see [GW93]). With this technique, information about the past of the search is used to compute such sets T. This technique is fully compatible with the refined dependencies of the previous Section and will not be discussed further here. The second technique is actually a whole family of algorithms [Ove81, Val91, GW91b] that compute "persistent sets".

We define persistent sets of transitions as follows:

Definition 7. A set T of transitions that are enabled in state s is said to be *persistent* in s iff, for all sequences $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions $t_0, t_1, t_2, \ldots, t_n \notin T$ from s in A_P, t_n is independent in s_n with respect to all transitions in T.

Intuitively, a persistent set T in a state s is a set of enabled transitions whose occurrence can not be affected by the evolution of the system by transitions outside this set T from state s. Note that the set of all enabled transitions is trivially persistent.

It can be shown that, at each state reached during the search, it is sufficient to explore only the transitions of a (nonempty) persistent set rather than all enabled transitions in order to detect all deadlocks of the program. Other properties than deadlock detection can be verified by using additional conditions that must be met at each state reached during the search [Val91, Val90, HGP92].

The basic idea of all the algorithms [Ove81, Val91, GW91b] that compute persistent sets is the same: they use information about the static structure (code) of the program

being verified. They differ by the type of information about the program they use (e.g. "distinction between local and global transitions", "which process can access which variable", "which process can access which variable from its current location", etc.) and therefore also by their time complexity. Indeed, analyzing more information about the program requires more time but can yield smaller persistent sets. See [HGP92] for a quick comparison between these algorithms. Note that exploring the smallest number of enabled transitions at each step of the search is a heuristics: it does not necessary lead to the exploration of the smallest number of states.

The most elaborated technique of this family that has been proposed so far is the stubborn set technique of Valmari. Stubborn sets[3] are formally defined as follows [Val91]:

Definition 8. A set T_s of transitions is a stubborn set in state s if T_s contains at least one enabled transition and $\forall t \in T_s$:

1. if t is disabled in s, and c_i is a necessary condition for t to be enabled which is false in s, then all transitions t' whose execution can make c_i *true* are also in T_s;
2. if t is enabled in s, then all transitions t' such that t and t' are dependent are also in T_s.

It can be shown that, by taking all enabled transitions of a stubborn set, one obtains a persistent set. From the above definition, one can derive algorithms for computing stubborn sets. In [Val91] two such algorithms are given.

When considering processes communicating with each other via shared objects, we have shown that dependencies between transitions arise from dependencies between operations on shared objects. To use the previous stubborn set definition, we need the following information:

1. By point 1 of the definition, we need to determine the set of transitions t' whose execution can change the truth value of a condition c_i from false to true. Thus, for each operation *op* used to compute c_i, we have to determine the operations op' on the same object that could change the value returned by *op*, i.e. the operations that are *dependent* with *op*, and next determine all transitions t' that can perform at least one such operation op'.
2. By point 2 of the definition, we need to determine the set of transitions that are dependent with t, i.e. the set of transitions that use at least one operation op' that are *dependent* with one of the operations used by t.

Both cases involve the notion of dependency expressed as a *constant* property between transitions. Definition 8 is still valid in the context considered here but cannot gain from the use of conditional dependency. Therefore, algorithms derived from this definition will produce unnecessarily large persistent sets.

In order to avoid this, we now give a new yet more general definition inspired from the stubborn set definition that can be used to compute smaller persistent sets. Unlike the above definition, the new definition takes conditional dependency into account.

Definition 9. A set T_s of transitions is a *conditional stubborn set* in state s if T_s contains at least one enabled transition and $\forall t \in T_s$:

[3] "Stubborn sets in the strong sense" according to Valmari's terminology. "Stubborn sets in the weak sense" will not be considered here.

1. if t is disabled in s, then for all sequences $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions from s in A_P such that t is enabled in s_{n+1} (which implies that $\exists i, 0 \leq i \leq n$, such that t and t_i are dependent in s_i), at least one of the $t_0, t_1, \ldots, t_n$ is also in T_s.
2. if t is enabled in s, then for all sequences $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions from s in A_P such that t and t_n are dependent in s_n, at least one of the $t_0, t_1, \ldots, t_n$ is also in T_s.

Theorem 10. *The set of all enabled transitions in a conditional stubborn set T_s is persistent in s.*

Proof. See full paper.

The differences between Definition 8 and Definition 9 are that dependencies between transitions are not considered in all states but only for successor states of s, and that one adds to T_s one of the t_i transitions of the path leading to a dependent transition t_n, instead of t_n. It can be proved that all sets T_s satisfying Definition 8 also satisfy Definition 9, while the converse is not true. Therefore, Definition 9 is finer and can be used to produce smaller persistent sets than Definition 8.

However, it is not obvious to develop a practical algorithm that would produce automatically sets T_s according to Definition 9. Indeed, this definition uses information about sequences of transitions in the state space, about which no assumption can be made!

Nevertheless, this more general definition can be profitably used to define a relation which models very finely the possible interactions between operations on a same object. More precisely, our idea is to define a relation $\rhd_s$ between operations on an object that would tell us for each operation used by a transition in T_s which other operations might "interact" with it, and thus which other transitions should be added to T_s as well. The relation "might interact" is represented by the relation $\rhd_s$ which is formally defined as follows:

Definition 11. Let op and op' be two operations on a same object O and s be a reachable state. If it is impossible to have a sequence $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions from s in A_P such that $\forall 0 \leq i < n : \forall op''$ on O used by t_i: op and op'' are independent in state s_i, t_n uses op', and op and op' are dependent in s_n, then $op \not\rhd_s op'$; else $op \rhd_s op'$.

Given such a $\rhd_s$ relation for all operations that can be performed on shared objects, one can proceed as follows to compute conditional stubborn sets:

1. If $t \in T_s$ is disabled in s and c_i is a necessary condition for t to be enabled which is false in s, then, for all operations op used to evaluate c_i, all transitions t' that use an operation op' such that $op \rhd_s op'$ are also in T_s.
2. If $t \in T_s$ is enabled in s, then, for all operations op used by t, all transitions t' that use an operation op' such that $op \rhd_s op'$ are also in T_s.

Theorem 12. *The previous procedure produces conditional stubborn sets as defined in Definition 9.*

Proof. See full paper.

To use the above procedure, we finally have to determine for each type of shared object what the relation $\triangleright_s$ is for each pair (op, op') of possible operations on this object. According to Definition 11, we have $op \triangleright_s op'$ unless it can be proved that it is impossible to have a sequence $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions from s in A_P such that $\forall 0 \leq i < n : \forall op''$ used by t_i: op and op'' are independent in state s_i and t_n uses op' with op and op' dependent in s_n.

The following table represents the relation $\triangleright_s$ for the channel example. For two operations op and op' on a same channel, if the condition given in row op and column op' in the table is true in a state s, then we have $op \triangleright_s op'$, while "–" denotes the fact that $op \not\triangleright_s op'$.

$\triangleright_s$	*Send*	*Receive*	*Length*	*Empty*	*Full*
Send	$n < N$	$n = N$	$n < N$	$n < N$	$n = N - 1$
Receive	$n = 0$	$n > 0$	$n > 0$	$n = 1$	$n > 0$
Length	$n < N$	$n > 0$	–	–	–
Empty	$n = 0$	$n > 0$	–	–	–
Full	$n < N$	$n = N$	–	–	–

For instance, let us show how to determine when *Send* $\triangleright_s$ *Receive*. One has to determine when it is impossible to find a sequence $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{m-1}} s_m \xrightarrow{t_m} s_{m+1}$ of transitions from s such that the *Send* and *Receive* operations are dependent in s_m, and $\forall 0 \leq i < m : \forall op''$ used by t_i: *Send* and op'' are independent in state s_i. Since *Send* and *Receive* are dependent in s_m, we obtain from the conditional dependency relation between *Send* and *Receive* (see Section 4) that either $n = 0$ or $n = N$ in s_m. If $n = 0$ in s_m, the *Receive* operation is not defined in s_m and there can not be a transition t_m executing a *Receive* operation such that $s_m \xrightarrow{t_m} s_{m+1}$. If $n = N$ in s_m, the *Receive* operation is defined. If $n < N$ in s, and since $n = N$ in s_m, at least one transition t_i in the sequence from s to s_m executes an operation that changes the value of n from $n < N$ to N. This operation can only be a *Send* operation and is performed from state s_i such that $n < N$. Therefore, we obtain from the conditional dependency relation between *Send* and *Send* when $n < N$ that the two *Send* operations are dependent. It is thus impossible to find a sequence satisfying Definition 11 when $n < N$ in s. One concludes that *Send* $\triangleright_s$ *Receive* only when $n = N$.

Note that it would not have been possible to obtain such a proof without using conditional dependency and conditional stubborn sets.

6 Experiments

We have implemented the algorithms discussed in the previous Section in an automated protocol validation system called SPIN [Hol91], which accepts PROMELA as modeling language. We present experiments made with our implementation on two sample real protocols[4]:

- URP is AT&T's Universal Receiver Protocol, modeled in 419 lines of PROMELA. It consists of three processes communicating via FIFO channels.
- DTP is a data transfer protocol modeled in 391 lines of PROMELA. It consists of three processes communicating via FIFO channels.

Experiments were performed using three different algorithms:

[4] We thank Gerard J. Holzmann for providing us with these examples.

- Algo 1: a classical depth-first search (exploring all reachable states).
- Algo 2: a partial-order verification algorithm with an *unrefined* dependency relation (all operations on a same channel are dependent).
- Algo 3: the same partial-order verification algorithm with a *refined* dependency relation (*Length* refined with *Empty* and *Full*, use of a conditional dependency relation).

The results obtained with these three algorithms for the URP and DTP protocols are presented in the following table. Experiments were performed on a SPARC2 workstation (64 Megabytes of RAM). For each run, the numbers of visited states and traversed transitions are given. Time (in seconds) is user time plus system time as reported by the UNIX system time command.

Protocol	Algorithm	Stored States	Transitions	Time
URP	Algo 1	19,515	47,836	4.9
	Algo 2	6,759	7,779	11.1
	Algo 3	4,430	4,659	7.9
DTP	Algo 1	251,409	648,467	59.8
	Algo 2	56,626	65,710	35.8
	Algo 3	9,920	10,367	7.0

These results clearly show that refining dependencies can yield substantial improvements for both the time and memory requirements of the partial-order verification algorithm. For the DTP example, using Algo 2 reduces the number of stored states, i.e. the memory requirements, by a factor of 5 with respect to the classical state-space exploration performed by Algo 1. By carefully defining dependencies, Algo 3 can again reduce the memory requirements by another factor of 5.

7 Conclusions

The results of Section 6 demonstrate that tracking dependencies in a concurrent program is a basic issue that strongly influences the performances of partial-order verification techniques. It is therefore very important to define dependencies as finely as possible. However, as illustrated in Section 5, carefully tracking and exploiting dependencies between operations on a same object is by no means a trivial task. Fortunately, this has to be done only once for each type of object.

Therefore, we advocate the use of object libraries where classic high-level communication objects (such as various definitions of communications channels including lossy channels, shared variables, semaphores, etc), operations on these objects, the dependency and $\triangleright_s$ relations are defined as carefully as possible once for all. One can then specify concurrent systems by using these object libraries and thus gain from the refined dependencies during verification which is still fully automatic. In contrast, we discourage the opposite approach consisting of defining "everything", including objects, by processes (for instance, a transmission medium is usually modeled by a process that transmits messages).

Note that all this is quite natural. Indeed, when using such objects, one indirectly provides more information to the verification tool about the structure of the state space of the program being verified. If the tool is clever enough to be able to use these information (as it is the case with a partial-order verification tool), it is not surprising

that the verification can be performed more efficiently and becomes applicable to larger systems.

We have implemented partial-order verification techniques like the ones discussed in Section 5 and such object libraries in an add-on package for the validation tool SPIN [Hol91]. This Partial-Order Package is available free of charge for educational and research purposes by anonymous ftp from montefiore.ulg.ac.be from the /pub/po-package directory.

References

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992. Lecture Notes in Computer Science, Springer-Verlag.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990.

[GW91a] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 1991.

[GW91b] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342, Aalborg, July 1991.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design, Kluwer Academic Publishers*, 2(2):149–164, April 1993.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[Hol88] G. J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, 1988.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[JJ89] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic specifications. In *Workshop on automatic verification methods for finite state systems*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196, Grenoble, June 1989.

[KP92] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, 1986.

[McM92] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[Och90] E. Ochmanski. Semi-commutation and deterministic petri nets. In *Proc. Symposium on Mathematical Foundations of Computer Science*, volume 452, pages 430–438. Lecture Notes in Computer Science, 1990.

[Ove81] W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California Los Angeles, 1981.

[Pel92] D. Peled. All from one, one for all: on model checking using representatives. Technical report, AT&T Bell Laboratories, 1992.

[PL90] D. K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 146–155, Rutgers, June 1990.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.