Minimal State Graph Generation *

A. Bouajjani, J-C. Fernandez,
N. Halbwachs, P. Raymond
IMAG/LGI (U.A. CNRS 398)
B.P. 53 - 38041 Grenoble, France

C. Ratel
Merlin Gerin / SES
38050 Grenoble, France

May 4, 1994

Abstract

We address the problem of generating a minimal state graph from a program, without building the whole state graph. Minimality is considered here with respect to bisimulation. A generation algorithm is derived and illustrated. Applications concern program verification and control synthesis in reactive program compilation.

1 Introduction

This paper concerns the problem of explicitly building a state graph from a program, a formula or any implicit expression of a transition system. Such state graphs are used in program verification ("model checking" [7, 14], behavioral equivalence [12]) and compiling (control structure synthesis [1, 6]).

A crucial problem with state graph generation is the size of the graph, which can be prohibitive. This size can be large not only because of the intrinsic complexity of the program, but also because the graph contains a lot of states which are in some sense equivalent. Of course, the proliferation of equivalent states increases the number of states, but its consequences are even worse concerning the number of transitions: if any state has n equivalent copies, the number of transitions can be multiplied by n^2.

To explain the problem, let us give a very simple example, which will be considered throughout the paper. The following program could be a boolean abstraction of a more realistic program:

```
x := true; y := false; w:= true; read(a);
loop
   write(x or y);
   z := y; y := (x and w) or a;
   x := not z; w := (not w and x) or y; read(a);
end;
```

Now, assume we want to prove some property about the output of this program (e.g., the temporal formula $\Diamond\neg(x \vee y)$). We consider the program as a transition system, whose states are the values of the variables when the output is written. The standard model checking procedure

*This work was partially supported by ESPRIT Basic Research Action "SPEC"

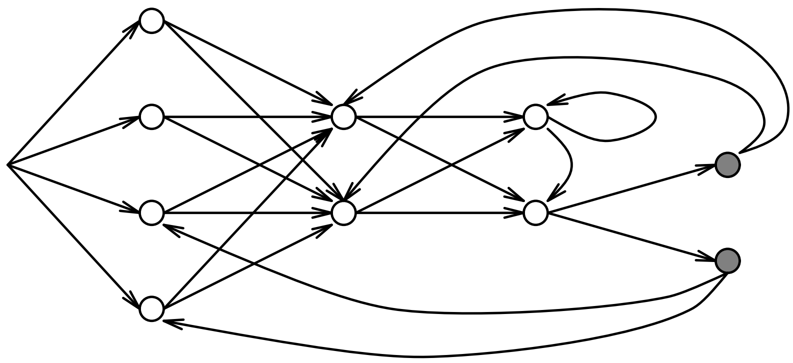

Figure 1: The complete state graph of the example

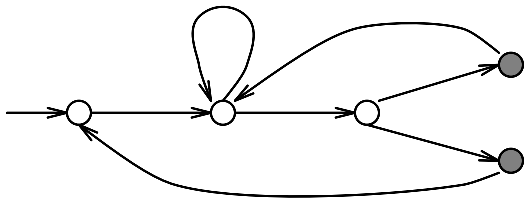

Figure 2: The minimal state graph of the example

consists of building first the state graph of this transition system: There are 4 initial states, depending on the initial values of a and z:

$$q_0 = (x\,\overline{y}\,z\,w\,a) \quad , \quad q_1 = (x\,\overline{y}\,\overline{z}\,w\,a) \quad , \quad q_2 = (x\,\overline{y}\,z\,w\,\overline{a}) \quad , \quad q_3 = (x\,\overline{y}\,\overline{z}\,w\,\overline{a})$$

These states, in which $x \vee y$ is true, can lead to two other states:

$$q_4 = (x\,y\,\overline{z}\,\overline{w}\,a) \quad , \quad q_5 = (x\,y\,\overline{z}\,\overline{w}\,\overline{a})$$

Continuing the simulation, we get a graph with 10 states and 20 transitions, represented by Fig. 1. In two of these states (filled in grey on the figure), $x \vee y$ is false. The point is that, as we are only interested in the value of $x \vee y$, we could have considered only a graph with 5 states and 7 transitions (represented by Fig. 2), which is equivalent to the former with respect to that value — in the sense that any property involving only $x \vee y$ is true on one graph if and only if it is true on the other.

Some solutions have been given to this problem, by applying reduction algorithms [11, 13, 9]. However, these algorithms can only be applied once the graph has been entirely generated. It is often the case that a tremendous amount of time and memory is necessary to generate a graph, which afterward reduces to a very simple one. It even happens that an infinite graph reduces to a finite one. So, it would be interesting to reduce the graph during the generation, on one hand to improve the performance of the generation, and on the other hand, to allow finite state graph generation from infinite systems.

This paper presents and illustrates an algorithm performing this task, when the equivalence considered on states is a bisimulation. The basis of our algorithm is twofold:

- It combines the construction of the graph of accessible states with its reduction by bisimulation. The termination of the algorithm imposes that the quotient of the whole space of (accessible and inaccessible) states by the bisimulation be finite.
- It considers state classes instead of states. For this purpose, a formal calculus over state classes is needed.

A typical case where the algorithm works well is when the state space is a boolean space $\{true, false\}^n$. Then the termination condition is obviously satisfied, and state classes may be handled as boolean formulas.

A first version of the algorithm has been published in [3]. In this paper, it will be formally derived from a fixpoint characterization.

The paper is organized as follows: Section 2 describes the theoretical framework of the problem; it shows that we have to combine the computations of a least fixpoint — the set of accessible states — and a greatest fixpoint — the bisimulation. From this formal setting, the algorithm is derived in Section 3. It is illustrated by an example in Section 4. In conclusion, we shall give some experimental results about the implementation of the algorithm, which has been inserted in the new compiler of the language LUSTRE [10]. All the proofs are given in the Appendix.

2 Theoretical Framework

2.1 Transition systems

In the following, we consider a transition system: $\mathcal{S} = (Q, \rightarrow, q_{init})$, where Q is a set of states, $\rightarrow \subseteq Q \times Q$ is a transition relation, and q_{init} is the initial state.

The pre- and post-condition functions, from 2^Q to 2^Q are defined as usual:

$$\begin{aligned} pre(X) &= \{q \in Q \mid \exists q' \in X \text{ such that } q \rightarrow q'\} \\ post(X) &= \{q \in Q \mid \exists q' \in X \text{ such that } q' \rightarrow q\} \end{aligned}$$

Transition systems are generally represented implicitly, for instance by programs. We shall call a *state graph* of $\mathcal{S}$ any explicit representation of $\mathcal{S}$. For instance, the "program"

```
n := 0 ; loop n := n+1 end
```

represents the transition system, whose states are memories $M \in (\{\mathtt{n}\} \mapsto \mathbb{Z})$, whose initial state is $[0/\mathtt{n}]$ (the function associating 0 with n) and whose transition relation is the function $\lambda M.M[(M(\mathtt{n})+1)/\mathtt{n}]$. Its (infinite) state graph is $[0/\mathtt{n}] \rightarrow [1/\mathtt{n}] \rightarrow [2/\mathtt{n}] \rightarrow \cdots$

2.2 Partitions, Partial Partitions

Throughout the paper, we consider partitions of non-empty subsets of Q instead of equivalence relations on Q. Moreover, we are only interested in partial partitions containing *accessible* classes (a class is accessible if it contains an accessible state, see formal definition in section 2.3). Let $\mathcal{P}$ be the set of sets of pairwise disjoint non-empty subsets of Q or partial partitions of Q:

$$\mathcal{P} = \{\pi \subseteq 2^Q \mid \forall X, Y \in \pi, X, Y \neq \emptyset \text{ and } X \cap Y \neq \emptyset \Rightarrow X = Y\}$$

$\pi \in \mathcal{P}$ is a partition if and only if it covers Q ($Q = \bigcup\{X \mid X \in \pi\}$).

In the remainder of the paper, $X, Y, Z, T, \ldots$ will be metavariables for subsets of Q, π for partial partitions on Q and ρ for partitions or partial partitions on Q.

We consider two orders:

- the usual inclusion relations over 2^Q and 2^{2^Q}, which make them complete lattices, with usual least upper bound ($\cup$) and greatest lower bound ($\cap$) operators.
- the refinement relation, noted $\sqsubseteq$, over $\mathcal{P}$:

  $$\pi \sqsubseteq \pi' \text{ iff } \forall X \in \pi, \exists X' \in \pi' \text{ such that } X \subseteq X'$$

  With this order, $\mathcal{P}$ is a complete lattice, with greatest lower bound operator $\sqcap$ and least upper bound operator $\sqcup$, defined as follows:

  $$\bigsqcap_i \pi_i = \{T \neq \emptyset \mid T = \bigcap_i X_i \text{ and } X_i \in \pi_i\}$$

  $$\bigsqcup_i \pi_i = \bigsqcap\{\pi \mid \forall i, \pi_i \sqsubseteq \pi\}$$

  with the infimum $\emptyset$ and supremum $\{Q\}$. Notice that the sublattice of partitions is also a complete lattice with this order, but with infimum $\{\{q\} \mid q \in Q\}$.

We denote by $[q]_\pi$ the class of the partial partition π containing the state q, if such a class exists. Let $pre_\pi, post_\pi$ denote the pre- and post-condition functions corresponding to a partial partition π; these functions are overloaded

- from 2^Q to $\mathcal{P}$: $\quad pre_\pi(X) = \{[q]_\pi \mid q \in pre(X)\} \quad , \quad post_\pi(X) = \{[q]_\pi \mid q \in post(X)\}$
- from $\mathcal{P}$ to $\mathcal{P}$: $\quad pre_\pi(\pi') = \bigcup\{pre_\pi(X) \mid X \in \pi'\} \quad , \quad post_\pi(\pi') = \bigcup\{post_\pi(X) \mid X \in \pi'\}$

2.3 Accessible states and bisimulations

Let F be a monotonic total function, either from 2^Q to 2^Q or from 2^{2^Q} to 2^{2^Q} and let G be a monotonic total function from $\mathcal{P}$ to $\mathcal{P}$. We denote by

- $\mu\pi.F(\pi)$ the least fixpoint of F with respect to the ordering $\subseteq$
- $\nu\pi.G(\pi)$ the greatest fixpoint of G with respect to the ordering $\sqsubseteq$

Accessibility: A state q is *accessible* in $\mathcal{S}$ if and only if either $q = q_{init}$ or there exist $q_1, q_2, \ldots, q_n$ in Q, such that $q_{init} \rightarrow q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q$. With the above notations, the set Acc_0 of accessible states of $\mathcal{S}$ is $Acc_0 = \mu X.(\{q_{init}\} \cup post(X))$.

More generally, if X and X' belong to a partial partition π, X' is said to be directly π-accessible from X (noted $X \xrightarrow{\pi} X'$) iff there exists $q \in X, q' \in X'$, such that $q \rightarrow q'$. π-accessibility is defined to be the transitive closure of that direct π-accessibility. We define $Acc(\pi)$ to be the set of classes which are π-accessible from $[q_{init}]_\pi$. We have

$$Acc(\pi) = \mu\pi'.(\{[q_{init}]_\pi\} \cup post_\pi(\pi'))$$

Notice that the fact that a set X belongs to $Acc(\pi)$ doesn't mean that X contains an accessible state.

Splitting: We define a hierarchy of (overloaded) *splitting functions*:

- from $2^Q \times 2^Q$ to $\mathcal{P}$: $\forall X, Y \in 2^Q, split(X,Y) = \{X \cap pre(Y)\} \cup \{X \setminus pre(Y)\}$. In other words, this function splits its first argument into the set of states which respectively can or cannot lead to the second argument.
- from $2^Q \times \mathcal{P}$ to $\mathcal{P}$: $\forall X \in 2^Q, \forall \pi \in \mathcal{P}, split(X,\pi) = \bigsqcap\{split(X,Y) \mid Y \in \pi\}$
- from $\mathcal{P} \times \mathcal{P}$ to $\mathcal{P}$: $\forall \pi, \pi' \in \mathcal{P}, split(\pi,\pi') = \bigcup\{split(X,\pi') \mid X \in \pi\}$

Stability: X (respectively π') is said to be *stable* with respect to π, if and only if $\{X\} = split(X,\pi)$ (resp. $\pi' = split(\pi',\pi)$). A partition ρ is a bisimulation if and only if it is stable with respect to itself. By extension, a partial partition π is a bisimulation if and only if it is stable with respect to itself and if all accessible states belong to a class of π,i.e. $Acc_0 \subseteq \cup\{X \mid X \in \pi\}$. The greatest bisimulation refining a given partition ρ_{init} is the greatest fixpoint (with respect to $\sqsubseteq$) of the function $\lambda\rho.(\rho_{init} \sqcap split(\rho,\rho))$.

The reduction of a transition system $\mathcal{S} = (Q, \rightarrow, q_{init})$ according to a bisimulation π of Q, is the transition system $\mathcal{S}/\pi = \{\pi, \xrightarrow{\pi}, [q_{init}]_\pi\}$,where

- $[q_{init}]_\pi$ is the class of the initial state in π
- $X \xrightarrow{\pi} Y$ if and only if $\exists q \in X,\ q' \in Y$ such that $q \rightarrow q'$

Let us state some useful properties of *split*:

Proposition 1 *(properties of split)*

1. $split(\rho,\rho) \sqsubseteq \rho$
2. $\rho_1 \sqsubseteq \rho_2 \Longrightarrow split(\rho_1,\rho) \sqsubseteq split(\rho_2,\rho)$
3. $\rho_1 \sqsubseteq \rho_2 \Longrightarrow split(\rho_1,\rho_1) \sqsubseteq split(\rho_2,\rho_2)$
4. $\rho_1 \sqsubseteq \rho_2 \Longrightarrow split(Acc(\rho_1),\rho_1) \sqsubseteq split(Acc(\rho_2),\rho_2)$

2.4 Reduction of a transition system

Our problem of explicit generation of the minimal transition system of $\mathcal{S}$, with respect to the greatest bisimulation included in a given equivalence ρ_{init}, may be solved by two different strategies. For each of them, we propose a set of equations and the method for solving it.

Strategy 1: Accessible states followed by refinement

Let

$$\begin{aligned} F_1 &= \lambda X.(\{q_{init}\} \cup post(X)) \\ G_1 &= \lambda(\rho,\pi).(\rho_{init} \sqcap split(\rho \sqcap \pi, \rho \sqcap \pi)) \end{aligned}$$

F_1 is monotonic on 2^Q. $\lambda\pi.G_1(\rho,\pi)$ is monotonic on $\mathcal{P}$ from proposition 1, part 3. The first strategy consists of computing first the set of accessible states of $\mathcal{S}$, that is the least fixpoint $Acc_0 = \mu X.F_1(X)$ in the lattice 2^Q. Then, this set is refined according to the greatest bisimulation included in ρ_{init}, i.e., the greatest fixpoint $\overline{\pi} = \nu\pi.G_1(\{Acc_0\},\pi)$ is computed in $\mathcal{P}$. The result is the quotient of $\mathcal{S}$ by the greatest bisimulation included in ρ_{init}. This is the classical method, which leads to state explosion if Acc_0 is generated by enumeration.

Strategy 2: Refinement followed by accessible classes

Let

$$\begin{aligned} G_2 &= \lambda\rho.(\rho_{init} \sqcap split(\rho, \rho)) \\ F_2 &= \lambda(\pi, \rho).([q_{init}]_\rho \cup post_\rho(\pi)) \end{aligned}$$

G_2 is monotonic on $\mathcal{P}$ from proposition 1, part 1. $\lambda\pi.F_2(\pi, \rho)$ is monotonic on 2^{2^Q}. The second strategy consists of refining first the whole set of states, without regards to accessibility, i.e., the greatest fixpoint $\overline{\rho} = \nu\rho.G_2(\rho)$ is computed first in $\mathcal{P}$. Then, the set of accessible classes, i.e., the least fixpoint $Acc(\overline{\rho}) = \mu\pi.F_2(\pi, \overline{\rho})$ is computed in the lattice 2^{2^Q}. So, the quotient $Q/\overline{\rho}$ of the whole set of states by the bisimulation is computed first, and then the set $Acc(\overline{\rho})$ of accessible states of the quotient system $\mathcal{S}/\overline{\rho}$ is explored. Here, the drawback is that the bisimulation is uselessly built over unaccessible states.

So, it is interesting to look for intermediate methods, combining the exploration of accessible states or classes with the computation of the bisimulation. First notice that the problem is not trivial, since we have to combine the computation of least and greatest fixpoints. In the following section, we shall describe an algorithm, derived from strategy 2.

We first state that both strategies lead to the same graph, as far as accessible states are concerned:

Definition: Let $\mathcal{S} = (Q, \rightarrow, q_{init})$ and $\mathcal{S}' = (Q', \rightarrow', q'_{init})$ be two transition systems, $Acc_\mathcal{S}$ and $Acc_{\mathcal{S}'}$ be their respective sets of accessible states. $\mathcal{S}$ and $\mathcal{S}'$ are said to be *isomorphic* if and only if there exists a bijection f from $Acc_\mathcal{S}$ to $Acc_{\mathcal{S}'}$ such that $f(q_{init}) = q'_{init}$ and $q_1 \rightarrow q_2 \iff f(q_1) \rightarrow' f(q_2)$.

Proposition 2 *Let $\overline{\pi} = \nu\pi.G_1(\{Acc_0\}, \pi)$ and $\overline{\rho} = \nu\rho.G_2(\rho)$. The two transition systems*

$$\mathcal{S}_1 = (\overline{\pi}, \xrightarrow{\overline{\pi}}, [q_{init}]_{\overline{\pi}}) \quad \text{and} \quad \mathcal{S}_2 = (Acc(\overline{\rho}), \xrightarrow{\overline{\rho}}, [q_{init}]_{\overline{\rho}})$$

are isomorphic.

This proposition results from the three following lemmas:

Lemma 1 *If ρ_1 and ρ_2 are two bisimulations on Q such that $\rho_1 \sqcap \{Acc_0\} = \rho_2 \sqcap \{Acc_0\}$, then $\mathcal{S}/\rho_1$ and $\mathcal{S}/\rho_2$ are isomorphic.*

Lemma 2 *The two following partitions of Acc_0 are equal:*

$$A = \{Acc_0\} \sqcap \nu\rho.G_1(\{Acc_0\}, \rho) \qquad B = \{Acc_0\} \sqcap \nu\rho.G_2(\rho)$$

Lemma 3

1. *For any $\pi, \rho \in \mathcal{P}$, $\{Acc_0\} \sqcap split(\pi, \rho) = split(\{Acc_0\} \sqcap \pi, \rho)$*
2. *For any $\pi, \rho \in \mathcal{P}$, $split(\{Acc_0\} \sqcap \pi, \rho) = split(\{Acc_0\} \sqcap \pi, \{Acc_0\} \sqcap \rho)$*

3 Generation Algorithm

In this section, we shall derive a generation algorithm based on "stepwise refinement on accessible classes" strategy described below, which is an intermediate method between the two previous ones. The idea is twofold: On one hand, we only need to refine accessible classes, and on the other hand, only accessible classes can involve such a refinement.

3.1 Basic solution

Stepwise refinement on accessible classes
Let

$$G_3 \quad = \quad \lambda(\pi, \rho).\rho_{init} \sqcap \mathit{split}(\pi, \rho)$$

$G_3(Acc(\rho), Acc(\rho))$ is monotonic in ρ by proposition 1. The strategy consists in computing the greatest fixpoint $\nu\rho.G_3(Acc(\rho), Acc(\rho))$, in the lattice $\mathcal{P}$. Given an initial partition, we compute the set of its accessible classes, and then we apply one step of refinement of this set with respect to itself, and so on, until all the classes are stable. Thus, $\nu\rho.G_3(Acc(\rho), Acc(\rho))$ is the limit of the sequence:

$$\rho_0 \quad = \quad \rho_{init} \qquad \rho_{n+1} \quad = \quad \rho_{init} \sqcap \mathit{split}(\pi_n, \pi_n)$$

where

$$\pi_n \quad = \quad Acc(\rho_n)$$

The correctness of this strategy is expressed by the following proposition:

Proposition 3 *Let $\rho' = \nu\rho.(\rho_{init} \sqcap \mathit{split}(Acc(\rho), Acc(\rho)))$. Then $\rho' = Acc(\rho') = Acc(\overline{\rho})$.*

The proof of this proposition is based on the following lemma:

Lemma 4 *For any ρ,*

1. $\mathit{split}(Acc(\rho), \rho) = \mathit{split}(Acc(\rho), Acc(\rho))$.
2. $Acc(\rho_{init} \sqcap \mathit{split}(\rho, \rho)) = Acc(\rho_{init} \sqcap \mathit{split}(Acc(\rho), \rho))$.

We can derive a naive algorithm for computing the limit ρ':

$\rho = \rho_{init}$; $\pi = Acc(\rho)$;
while $\rho \neq \mathit{split}(\pi, \pi)$ do
  $\rho = \mathit{split}(\pi, \pi)$;
  $\pi = Acc(\rho)$;
od

3.2 Optimizations

The algorithm derived so far presents some obvious weaknesses:

- At each step, the whole set of accessible classes is computed again. In the following, we shall try to compute π_{n+1} by updating π_n.

- In the same way, ρ_{n+1} can be derived from ρ_n without considering again all of its classes. When a class X is split during the computation of ρ_n, one can remember that its predecessors are likely to be split at the next step.

- The fact that a class belongs to $Acc(\rho_n)$ does not imply that it contains an accessible state. As a consequence, it may happen that a class is uselessly considered and split.

Let us consider the elementary *split* operation applied in updating the partition: Given a partition ρ and a set π of accessible classes of ρ, each class X in π is replaced by $split(X, \rho)$. Now, two cases can occur:

- either X is effectively split into several subclasses. Then, for deciding which of these subclasses remain accessible, we have to examine the predecessor classes of X. On the other hand, these predecessors are precisely the classes which may have to be split at the next step (in the sense of the fixpoint computation sequence) because of the splitting of X. So, we can first try to split the predecessor classes of X.
- or X is stable with respect to ρ ($\{X\} = split(X, \rho)$). Then, all the states in X lead exactly to the same classes in ρ. So, if X contains one accessible state q, all the classes accessible from X are accessible from q, and thus contain one accessible state. So, it is a good point to update the set π of accessible classes.

The set σ of stable classes with respect to the current partition ρ is interesting not only for determining accessible classes. Accessible unstable classes (those in $\pi \setminus \sigma$) are those which are likely to be split, and the knowledge of σ also provides a good termination criterion: when all accessible classes are stable, the generation is complete. A class enters σ when it is found to be stable when trying to split it; it must be removed from σ whenever one of its successor classes is split into several subclasses.

Taking these remarks into account, let us define the sequences $\{\rho''_n\}$, $\{\pi''_n\}$ and $\{\sigma_n\}$ as follows:

$$\rho''_0 = \rho_{init} \quad , \quad \sigma_0 = \emptyset$$

$$\pi^0_n = [q_{init}]_{\rho''_n} \qquad \sigma^0_n = \emptyset$$

$$\pi^{p+1}_n = \pi^p_n \cup post_{\rho''_n}(\sigma^p_n)$$

$$\sigma^{p+1}_n = \sigma^p_n \cup \{X \in \pi^{p+1}_n \mid split(X, \rho''_n) = \{X\}\}$$

$$\pi''_n = \lim_p \pi^p_n$$

$$\sigma_n = \lim_p \sigma^p_n$$

$$\rho''_{n+1} = \rho''_n \sqcap split(\pi''_n \setminus \sigma_n, \rho''_n) \cup \sigma_n \cup (\rho''_n \setminus \pi''_n)$$

At each step, π''_n contains all the classes accessible from a class of σ_n, which is the set of all the classes of π''_n which are stable with respect to ρ''_n.

Proposition 4 *If $\pi''_n = \sigma_n$ then ρ''_n is the limit ρ'' of the sequence $\{\rho''_i\}$ and $Acc(\rho'') = \rho'$.*

We get the following generation algorithm, which is exactly the one presented in [3], and which is proved correct in the Appendix:

$$
\begin{array}{l}
\rho = \rho_{init}; \pi = \{[q_{init}]_\rho\}; \sigma = \emptyset; \\
\text{while } \pi \neq \sigma \text{ do} \\
\quad \text{choose } X \text{ in } \pi \setminus \sigma; \\
\quad \text{let } \pi' = \mathit{split}(X, \rho); \\
\quad \text{if } \pi' = \{X\} \text{ then} \\
\quad\quad \sigma := \sigma \cup \{X\};\;\; \pi := \pi \cup \mathit{post}_\rho(X); \\
\quad \text{else} \\
\quad\quad \pi := \pi \setminus \{X\}; \\
\quad\quad \text{if } \exists Y \in \pi' \text{ such that } q_{init} \in Y \text{ then } \pi := \pi \cup \{Y\}; \\
\quad\quad \sigma := \sigma \setminus \mathit{pre}_\rho(X); \\
\quad\quad \rho := (\rho \setminus \{X\}) \cup \pi'; \\
\quad \text{fi} \\
\text{od}
\end{array}
$$

4 Example

Let us consider come back to our example program:

```
x := true; y := false; w:= true; read(a);
loop
   write(x or y);
   z := y; y := (x and w) or a;
   x := not z; w := (not w and x) or y; read(a);
end;
```

We start with the initial partition:

$$\{(a, w, x, y, z) \mid x \vee y = \mathit{true}\}, \{(a, w, x, y, z) \mid x \vee y = \mathit{false}\}$$

In the following, classes are represented by their characteristic formulas. The initial partition will be noted:

$$\rho_{init}: \quad C_1 = \{x \vee y\} \quad C_2 = \{\neg x \wedge \neg y\}$$

Standard rules of weakest precondition provide the precondition of a class X, with respect to the body of the loop:

$$\mathit{pre}(X) = (X \downarrow a)[((\neg w \wedge x) \vee y) \;/\; w][\neg z \;/\; x][((x \wedge w) \vee a) \;/\; y][y \;/\; z]$$

where $X \downarrow a = \exists a_0 X[a_0 \;/\; a] = X[\mathit{false} \;/\; a] \vee X[\mathit{true} \;/\; a]$

So, $\mathit{pre}(C_1) = \{(x \wedge w) \vee \neg y \vee a\}$, $\mathit{pre}(C_2) = \{(\neg x \vee \neg w) \wedge y \wedge \neg a\}$

The successive partitions built by the algorithm are illustrated on figure 3.

Step 1: The only reachable class is C_1, since x is initially true. For splitting it, we compute:

$$C_1 \wedge pre(C_1) = \{x \wedge (\neg y \vee w \vee a) \vee (y \wedge a)\}$$

$$C_1 \wedge \neg pre(C_1) = \{(\neg x \vee \neg w) \wedge y \wedge \neg a\}$$

and

$$C_1 \wedge pre(C_1) \wedge pre(C_2) = \mathit{false}$$

$$C_1 \wedge pre(C_1) \wedge \neg pre(C_2) = C_1 \wedge pre(C_1)$$

$$C_1 \wedge \neg pre(C_1) \wedge pre(C_2) = C_1 \wedge \neg pre(C_1)$$

$$C_1 \wedge \neg pre(C_1) \wedge \neg pre(C_2) = \mathit{false}$$

So, C_1 is split into:

$$C_{11} = \{x \wedge (\neg y \vee w \vee a) \vee (y \wedge a)\}$$

$$C_{12} = \{(\neg x \vee \neg w) \wedge y \wedge \neg a\}$$

and only C_{11} is accessible. We have:

$$pre(C_{11}) = \{(x \wedge w) \vee \neg y \vee a\};$$

$$pre(C_{12}) = \{((x \wedge w) \vee a) \wedge y\}$$

Step 2: For splitting C_{11}, we compute:

$$C_{11} \wedge pre(C_{11}) = \{x \wedge (\neg y \vee w \vee a) \vee (y \wedge a)\} = C_{11}$$

So, C_{11} is split into:

$$C_{111} = C_{11} \wedge pre(C_{11}) \wedge pre(C_{12}) = \{((x \wedge w) \vee a) \wedge y\}$$

$$C_{112} = C_{11} \wedge pre(C_{11}) \wedge \neg pre(C_{12}) = \{x \wedge \neg y\}$$

and only C_{112} is accessible. We have:

$$pre(C_{111}) = \{(x \wedge w) \vee a\}$$

$$pre(C_{112}) = \{(\neg x \vee \neg w) \wedge \neg y \wedge \neg a\}$$

Step 3: For splitting C_{112}, we compute:

$$C_{112} \wedge pre(C_{111}) \wedge pre(C_{112}) = \mathit{false}$$

$$C_{112} \wedge pre(C_{111}) \wedge \neg pre(C_{112}) = x \wedge \neg y \wedge (w \vee a)$$

$$C_{112} \wedge \neg pre(C_{111}) \wedge pre(C_{112}) = x \wedge \neg y \wedge \neg w \wedge \neg a$$

$$C_{112} \wedge \neg pre(C_{111}) \wedge \neg pre(C_{112}) = \mathit{false}$$

So, C_{112} is split into:

$$C_{1121} = C_{112} \wedge pre(C_{111}) \wedge \neg pre(C_{112}) = x \wedge \neg y \wedge (w \vee a)$$

$$C_{1122} = C_{112} \wedge \neg pre(C_{111}) \wedge pre(C_{112}) = x \wedge \neg y \wedge \neg w \wedge \neg a$$

and only C_{1121} is accessible. We have:

$$pre(C_{1121}) = \{(\neg x \vee \neg w) \wedge \neg y \wedge \neg a\}$$

$$pre(C_{1122}) = \{\neg x \wedge \neg y \wedge w \wedge \neg a\}$$

Step 4: When considering C_{1121}, we find it stable; all of its states lead to C_{111} and only to C_{111}. So, C_{111} becomes accessible.

Step 5: C_{111} is found stable

$$C_{111} \subseteq pre(C_{111}) \wedge pre(C_{12})$$

So, C_{12} becomes accessible.

Step 6: In turn, C_{12} is found stable, all of its states leading to C_2, which becomes accessible.

Step 7: C_2 is split into:

$$C_{21} = C_2 \wedge pre(C_{111}) = \neg x \wedge \neg y \wedge a$$

$$C_{22} = C_2 \wedge pre(C_{1121}) \wedge pre(C_{1122}) = \{\neg x \wedge \neg y \wedge \neg a \wedge w\}$$

$$C_{23} = C_2 \wedge pre(C_{1121}) \wedge \neg pre(C_{1122}) = \{\neg x \wedge \neg y \wedge \neg a \wedge \neg w\}$$

So C_{12} is removed from stable classes. We have:

$$pre(C_{21}) = pre(C_{23}) = \{(\neg x \vee \neg w) \wedge y \wedge \neg a\}$$

$$pre(C_{22}) = \mathit{false}$$

which means that C_{22} is accessible from nowhere.

Steps 8, 9, 10: C_{12} is considered again, and found stable, all of its states leading to both C_{21} and C_{23}, which become accessible. C_{21} and C_{23} are also found stable, respectively leading to C_{111} and C_{1121}.

Since all accessible classes have been considered and found stable, the algorithm stops. Figure 4 shows the resulting graph, which is exactly the one considered in the introduction (Fig. 2).

This example shows several features of our algorithm:

- In such a boolean case, the symbolic computation of classes is fully automatic. As a matter of fact, we have an implementation of the algorithm, which performs efficient boolean computations by means of "binary decision diagrams" [4, 2].
- Only 7 classes have been identified, and 5 accessible classes have been considered for splitting. However, the usual enumeration algorithm (strategy 1) builds a state graph with 10 states shown in Figure 1.
- Compared with strategy 2, notice that two classes (C_{1122} and C_{22}) have not been considered for splitting, since they are not accessible from the initial state. In this example, these classes happen to be stable; however, in the general case, building the quotient of the whole set of states, without regards to accessibility, can give rise to many irrelevant classes.

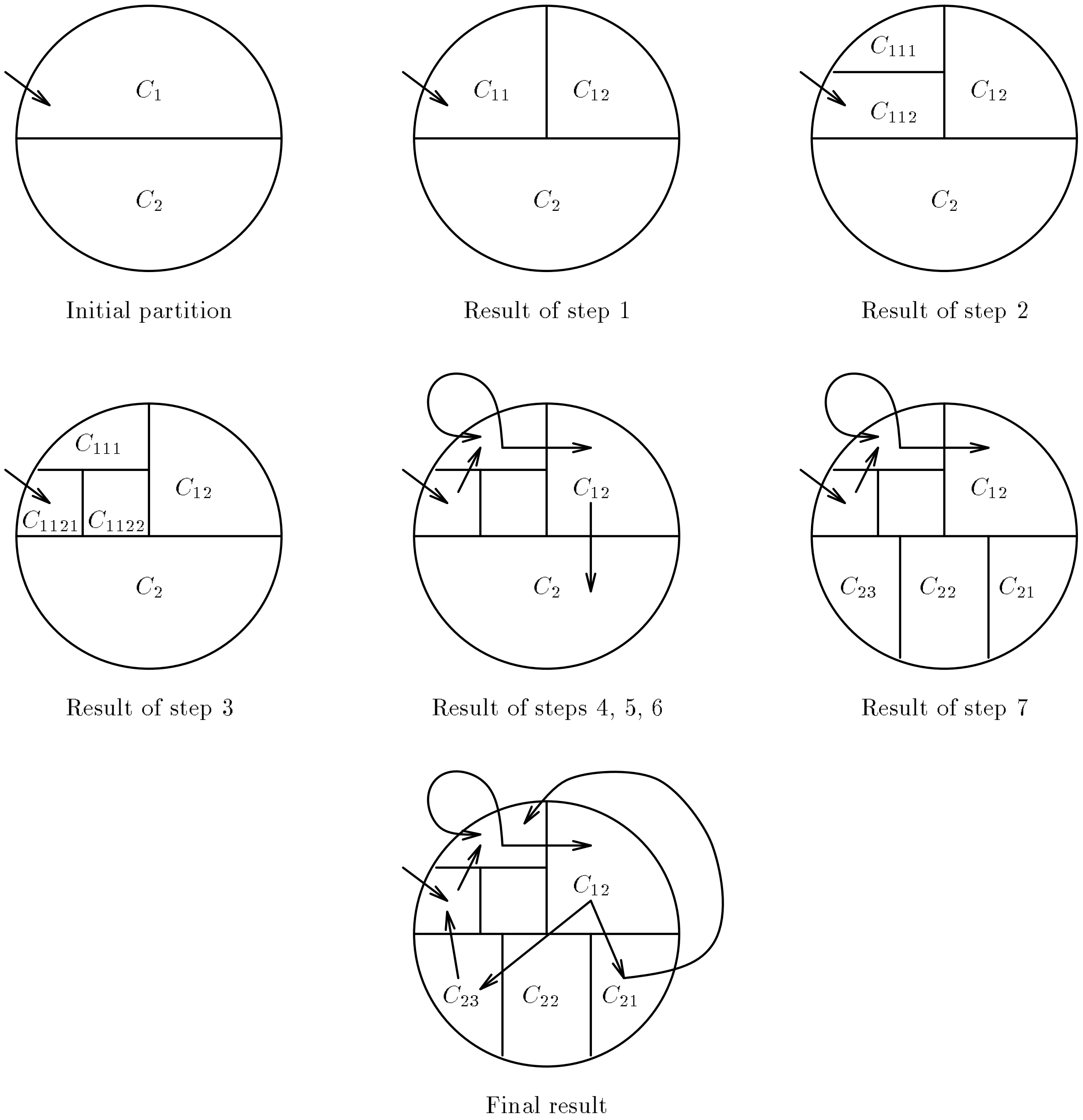

Figure 3: The successive partitions built by the algorithm

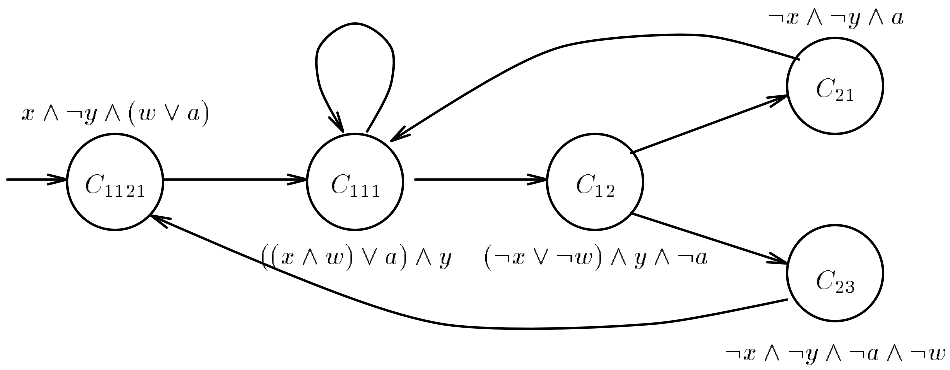

Figure 4: The reduced graph of the example

5 Conclusion

We have formally derived an algorithm combining generation and reduction methods. In our opinion, this algorithm is interesting for program verification: a state graph with several thousand (or even infinitely many) states may be reduced to one with a small number of states by considering an equivalence relation.

Applying our algorithm to program verification appears very close to what is now called "symbolic model checking" [5]. For instance, consider the problem of proving that a state property ϕ is true in any accessible state of a transition system. Applying our algorithm to this problem consists in starting from the initial partition $\{\phi, \neg\phi\}$ (verifying that the initial states belong to ϕ), then refining the class ϕ into $\{\phi \wedge pre(\neg\phi)\ ,\ \phi \wedge \neg pre(\neg\phi)\}$ (verifying that the initial states belong to $\phi \wedge \neg pre(\neg\phi)$) and so on, until getting only one stable class which characterizes all the states which cannot lead to $\neg\phi$. So, if ϕ is invariant, the resulting automaton has only one state. In that case, the algorithm is exactly the backward construction of the "greatest invariant included in ϕ".

Concerning other applications, the algorithm has been implemented in the new version of the LUSTRE compiler [10]. LUSTRE [6] is a synchronous declarative language designed for programming reactive systems. Its compiler produces efficient code by synthesising the control structure of the code as a finite automaton. Initially proposed for the ESTEREL language [1], this technique consists of an exhaustive enumeration of the states of boolean memories involved in a program. The first LUSTRE compiler performed this construction in a standard way, and often produced huge automata with many equivalent states (states with identical code). Our algorithm has been first designed in order to remedy this problem.

Of course, one must be able to compute the function *pre* and intersections of classes, and to decide the inclusion of classes. Such a symbolic computation is achievable in the boolean case, with reasonable average cost, by means of Binary Decision Diagrams [4, 8].

We have not presented complexity measures. In our opinion, a comparison of theoretical complexities wouldn't be very meaningful, for the following reason: The main cost and the main gain of our method come from symbolic computations. In the worst case, a symbolic computation on a formula is as costly as the enumeration of the set of states it characterizes. So, the main argument for our method concerns average performances, as for all symbolic methods using Binary Decision Diagrams. Let us give some experimental results about the use of the LUSTRE-V3 compiler: Considering the LUSTRE program corresponding to the simple example presented in Section 4, and a program implementing a digital watch — a very common example of reactive program — we give below

- the size of the state graph produced by the standard enumeration method, the time (on SUN4) for generating it, and the time for reducing it by means of ALDEBARAN [9].
- The size and generation time of the state graph when our method is used.

Simple example	Enumeration	Minimization	Minimal graph generation
Number of states	10	5	5
Number of transitions	20	7	7
Generation time (sec.)	0.5	0.2	0.5

Watch Program	Enumeration	Minimization	Minimal graph generation
Number of states	81	41	41
Number of transitions	1163	342	342
Generation time (sec.)	8.1	33.4	6.8

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[2] J.P. Billon and J.C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In G. Milne, editor, *IFIP WG 10.2 Int Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. North Holland.

[3] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In R. Kurshan, editor, *International Workshop on Computer Aided Verification, Rutgers*, June 1990.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, Munchen, January 1987.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[9] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.

[10] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[11] P. Kanellakis and S. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proceedings ACM Symp. on Principles of Distribued Computing*, 1983.

[12] R. Milner. *A Calculus of Communicating Systems.* LNCS 92, Springer Verlag, 1980.

[13] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.

[14] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In *IFIP WG-6.1 7th. International Conference on Protocol Specification, Testing and Verification*, Zurich, 1987. North Holland.

[15] J. Sifakis. A unified approach for studying the properties of transition systems. *TCS*, 3, 1982.

Appendix

Characterization of *split*

We denote by $\overline{X}$ the complement of a subset X of Q : $\overline{X} = Q \setminus X$. Let us first give a characterization of *split* using a predicate transformer $\widetilde{pre}$ defined below, introduced in [15] and shown to be $\bigcap$-continuous: $(\widetilde{pre}(X) = \overline{pre(\overline{X})})$.

$$\widetilde{pre}(X) \quad = \quad \{q \in Q \mid \forall q'.q \to q' \Longrightarrow q' \in X\} \quad = \quad \overline{pre(\overline{X})}$$

Lemma 5 $split(X,\rho) = \{T \mid T = X \cap \bigcap_{Y\in\pi} pre(Y) \cap \widetilde{pre}(\overline{\bigcup_{Y\in\pi} Y})$ *and* $\pi \subseteq \rho$ *and* $T \neq \emptyset\}$

Proof: From the definition of *split*, we have:

$$\begin{aligned}
split(X,\rho) &= \prod_{Y\in\rho} split(X,Y) \\
&= \prod_{Y\in\rho} \{X \cap pre(Y)\} \cup \{X \cap \widetilde{pre}(\overline{Y})\} \\
&= \{T \mid T = X \cap \bigcap_{Y\in\pi} pre(Y) \cap \bigcap_{Z\in\rho\setminus\pi} \widetilde{pre}(\overline{Z}) \text{ and } \pi \subseteq \rho \text{ and } T \neq \emptyset\} \\
&= \{T \mid T = X \cap \bigcap_{Y\in\pi} pre(Y) \cap \widetilde{pre}(\bigcap_{Z\in\rho\setminus\pi} \overline{Z}) \text{ and } \pi \subseteq \rho \text{ and } T \neq \emptyset\} \\
&\qquad\qquad \text{(by continuity of } \widetilde{pre}\text{)} \\
&= \{T \mid T = X \cap \bigcap_{Y\in\pi} pre(Y) \cap \widetilde{pre}(\overline{\bigcup_{Z\in\rho\setminus\pi} Z}) \text{ and } \pi \subseteq \rho \text{ and } T \neq \emptyset\} \\
&= \{T \mid T = X \cap \bigcap_{Y\in\pi} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi} Z) \text{ and } \pi \subseteq \rho \text{ and } T \neq \emptyset\}
\end{aligned}$$

From this lemma, we deduce :

Corollary 1

1. $split(\pi,\rho) = \{T \mid T = X \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho \text{ and } X \in \pi \text{ and } T \neq \emptyset\}$.

2. $split(\pi_1 \cup \pi_2, \rho) = split(\pi_1,\rho) \cup split(\pi_2,\rho)$.

Proof of Proposition 1

Part 1 is obvious.

Part 2 is a direct consequence of the corollary 1.

We now prove part 3 of proposition 1. Let $X \in split(\rho_1,\rho_1)$. Then, from lemma 5 above, there exists $X_1 \in \rho_1$ and $\pi_1 \subseteq \rho_1$ such that

$$X = X_1 \cap \bigcap_{Y\in\pi_1} pre(Y) \cap \widetilde{pre}(\bigcup_{Y\in\pi_1} Y).$$

Let $X_2 \in \rho_2$ and $\pi_2 \subseteq \rho_2$ such that $X_1 \subseteq X_2$ and $\pi_2 = \{Y_2 \in \rho_2 \mid \exists Y_1 \in \pi_1 \text{ such that } Y_1 \subseteq Y_2\}$. Then

$$X_1 \cap \bigcap_{Y\in\pi_1} pre(Y) \cap \widetilde{pre}(\bigcup_{Y\in\pi_1} Y) \subseteq \overbrace{X_2 \cap \bigcap_{Y\in\pi_2} pre(Y) \cap \widetilde{pre}(\bigcup_{Y\in\pi_2} Y)}^{Z}.$$

and $Z \in split(\rho_2,\rho_2)$.

To prove part 4, we state that $\rho_1 \sqsubseteq \rho_2 \Longrightarrow Acc(\rho_1) \sqsubseteq Acc(\rho_2)$ and then apply part 3. If $[q]_{\rho_1}$ exists and $\rho_1 \sqsubseteq \rho_2$, then $[q]_{\rho_2}$ exists and $[q]_{\rho_1} \subseteq [q]_{\rho_2}$. Thus $post_{\rho_1}(X) \sqsubseteq post_{\rho_2}(X)$ and $post_{\rho_1}(\pi) \sqsubseteq post_{\rho_2}(\pi)$. On the other hand, $Acc(\rho) = \mu\pi.(\{[q_{init}]_\rho\} \cup post_\rho(\pi))$. Thus, $Acc(\rho_1) \sqsubseteq Acc(\rho_2)$.

Proof of Lemma 1

Let us first point out that if ρ is a bisimulation, then $X \in Acc(\rho)$ if and only if $X \sqcap Acc_0 \neq \emptyset$.

Let $f = \lambda X.[X \cap Acc_0]_{\rho_2}$.

- f is a bijection from $Acc(\rho_1)$ to $Acc(\rho_2)$:
  Let $X \in Acc(\rho_1)$. Then $X \cap Acc_0 \in \rho_1 \sqcap \{Acc_0\}$, so $X \cap Acc_0 \in \rho_2 \sqcap \{Acc_0\}$ and there exists one and only one $Y \in \rho_2$ such that $X \cap Acc_0 \subseteq Y$.
  By symmetric reasoning, for all $Y \in Acc(\rho_2)$, there exists one and only one $X \in \rho_1$ such that $[X \cap Acc_0]_{\rho_2} = Y$.

- Obviously $f([q_{init}]_{\rho_1}) = [q_{init}]_{\rho_2}$.

- We show that $f \circ post_{\rho_1} \subseteq post_{\rho_2} \circ f$, the reverse inclusion being obtained by symmetry. Let $X \in Acc(\rho_1)$ and $X' \in post_{\rho_1}(X)$. Let us show that $f(X') \in post_{\rho_2}(f(X))$. Since $X \in Acc(\rho_1)$ and $X' \in post\rho_1(X)$, there exists $q \in X \cap Acc_0$ and $q' \in X'$ such that $q \to q'$. Since q is accessible, so is q', and $q' \in X' \cap Acc_0$. So, $q' \in f(X')$. On the other hand, since $q \in X \cap Acc_0$, $q \in f(X)$ and there exists $Y \in post_{\rho_2}(f(X))$ such that $q' \in Y$. So, $q' \in Y \cap f(X')$, and $f(X') = Y \in post_{\rho_2}(f(X))$.

Proof of Lemma 2

$$
\begin{aligned}
B &= \{Acc_0\} \sqcap \nu\rho.G_2(\rho) \\
&= \{Acc_0\} \sqcap \prod_{i\geq 0} G_2^i(\{Q\}) \\
&= \prod_{i\geq 0}(\{Acc_0\} \sqcap G_2^i(\{Q\})) \\
&= \prod_{i\geq 0} G_2^i(\{Acc_0\} \sqcap \{Q\}) \quad \text{since, from lemma 3} \\
&\qquad \begin{aligned} \{Acc_0\} \sqcap G_2(\rho) &= \rho_{init} \sqcap split(\{Acc_0\} \sqcap \rho, \{Acc_0\} \sqcap \rho) \\ &= G_2(\{Acc_0\} \sqcap \rho) \end{aligned} \\
&= \nu\rho.G_2(\{Acc_0\} \sqcap \rho) \\
&= \nu\rho.\rho_{init} \sqcap split(\{Acc_0\} \sqcap \rho, \{Acc_0\} \sqcap \rho) \\
&= \nu\rho.G_1(\{Acc_0\}, \rho) \\
&= \{Acc_0\} \sqcap \nu\rho.G_1(\{Acc_0\}, \rho) \;=\; A
\end{aligned}
$$

Proof of Lemma 3

The proof is based on the corollary 1.

$$
\begin{aligned}
\{Acc_0\} \sqcap split(\pi, \rho) &= \{Acc_0\} \sqcap \{X \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho \textit{ and } X \in \pi\} \\
&= \{Acc_0 \cap X \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho \textit{ and } X \in \pi\} \\
&= \{X' \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho \text{ and } X' \in \pi \sqcap \{Acc_0\}\} \\
&= split(\{Acc_0\} \sqcap \pi, \rho)
\end{aligned}
$$

which proves part *1*. For *2*, we have

$$
split(\{Acc_0\} \sqcap \pi, \rho) = \{Acc_0 \cap X \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho \textit{ and } X \in \pi\}
$$

Let us show

$$
Acc_0 \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) = Acc_0 \cap \bigcap_{Y\in\pi_\rho\sqcap\{Acc_0\}} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho\sqcap\{Acc_0\}} Z)
$$

which will provide the result. This is a consequence of the two following properties:

1 $Acc_0 \cap pre(Y) = Acc_0 \cap pre(Y \cap Acc_0)$
$\supseteq$: follows by monotonicity of *pre*.
$\subseteq$: Let $q \in Acc_0 \cap pre(Y)$. Then there exists $q' \in Y$ such that $q \to q'$. Now, since q is accessible, so is q', and $q' \in Y \cap Acc_0$. Thus, $q \in Acc_0 \cap pre(Y \cap Acc_0)$.

2 $Acc_0 \cap \widetilde{pre}(Z) = Acc_0 \cap \widetilde{pre}(Z \cap Acc_0)$
$\supseteq$: follows by monotonicity of $\widetilde{pre}$.
$\subseteq$: Let $q \in Acc_0 \cap \widetilde{pre}(Z)$. Then for all q' such that $q \to q'$, $q' \in Z$. Now, since q is accessible, so is q', and $q' \in Z \cap Acc_0$. Thus, $q \in Acc_0 \cap \widetilde{pre}(Z \cap Acc_0)$.

Then

$$
\begin{aligned}
Acc_0 \cap \bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) &= Acc_0 \cap \bigcap_{Y\in\pi_\rho} pre(Y \cap Acc_0) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z \cap Acc_0) \\
&= Acc_0 \cap \bigcap_{Y\in\pi_\rho\sqcap\{Acc_0\}} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho\sqcap\{Acc_0\}} Z)
\end{aligned}
$$

Proof of proposition 3

From the definition of the bisimulation as a greatest fixpoint, we have $\overline{\rho} = \sqcap\{\rho_n\}$, where $\rho_0 = \rho_{init}$ and $\rho_{n+1} = \rho_{init} \sqcap split(\rho_n, \rho_n) = \rho_n \sqcap split(\rho_n, \rho_n) = split(\rho_n, \rho_n)$. If $Q/\overline{\rho}$ is finite, the sequence $\{\rho_n\}_{n\geq 0}$ converges after a finite number of steps.

Let us introduce the sequence $\{\pi_n = Acc(\rho_n)\}$, which converges to $Acc(\overline{\rho})$. Now, $\rho' = \sqcap\{\rho'_n\}$, where $\rho'_0 = \rho_{init}$, $\rho'_{n+1} = \rho'_n \sqcap split(\pi'_n, \pi'_n)$ and $\pi'_n = Acc(\rho'_n)$. Let us show that, for any $n \geq 0$, $\pi'_n = \pi_n$. It is true for $n = 0$. Assume $\pi'_n = \pi_n$, i.e., $Acc(\rho_n) = Acc(\rho'_n)$. Then,

$$\begin{array}{rcll} Acc(\rho_{n+1}) & = & Acc(\rho_{init} \sqcap split(\rho_n, \rho_n)) & \\ & = & Acc(\rho_{init} \sqcap split(Acc(\rho_n), \rho_n)) & \text{from lemma 4 (part } \mathit{2}\text{) below} \\ & = & Acc(\rho_{init} \sqcap split(Acc(\rho_n), Acc(\rho_n))) & \text{from lemma 4 (part } \mathit{1}\text{)} \\ & = & Acc(\rho_{init} \sqcap split(Acc(\rho'_n), Acc(\rho'_n))) & \\ & = & Acc(\rho'_{n+1}) & \end{array}$$

We have $Acc(\rho'_n) \subseteq \rho'_n$. Moreover, if $\rho'_{n+1} = \rho'_n$, then $\rho'_n = split(Acc(\rho'_n), Acc(\rho'_n)) \sqsubseteq Acc(\rho'_n)$, by proposition 1, part 1. Thus, $\rho'_n = Acc(\rho'_n)$. We have also $\rho'_{n+2} = \rho'_{n+1} = \rho'_n = \rho'$. Thus, $\rho' = Acc(\rho')$.

Proof of Lemma 4

The proof of part *1* is very similar to the one of Lemma 3 (part *2*). For part *2*, let $\rho_1 = \rho_{init} \sqcap split(\rho, \rho)$, and let

$$\pi = \{\bigcap_{Y\in\pi_\rho} pre(Y) \cap \widetilde{pre}(\bigcup_{Z\in\pi_\rho} Z) \mid \pi_\rho \subseteq \rho\}$$

Now, $X \in Acc(\rho_1)$ implies that there exist $X_1 \in \rho_{init}$, $X_2 \in \rho$, and $X_3 \in \pi$ such that $X = X_1 \cap X_2 \cap X_3$. Since $\rho_1 \sqsubseteq \rho$, then $Acc(\rho_1) \sqsubseteq Acc(\rho)$. Moreover, $X \subseteq X_2$, $X_2 \in \rho$ and $X \in Acc(\rho_1)$. Thus, $X_2 \in Acc(\rho)$. So,

$$\begin{array}{rcl} Acc(\rho_1) & = & Acc(\{X_1 \cap X_2 \cap X_3 \mid X_1 \in \rho_{init}, X_2 \in \rho, X_3 \in \pi\}) \\ & = & Acc(\{X_1 \cap X_2 \cap X_3 \mid X_1 \in \rho_{init}, X_2 \in Acc(\rho), X_3 \in \pi\}) \\ & = & Acc(\rho_{init} \sqcap split(Acc(\rho), \rho)) \end{array}$$

Proof of proposition 4

The proof is based on the following properties:

1. $\pi''_n \subseteq Acc(\rho''_n)$.
2. $\rho'_n \sqsubseteq \rho''_n$.
3. $\pi''_n = \sigma_n \Longrightarrow \rho''_n = \rho''_{n+1}$ and $\pi''_{n+1} = \pi_n = \sigma_{n+1}$.
4. $\pi''_n = \sigma_n \Longrightarrow \pi''_n = Acc(\rho''_n) = \sigma_n$.

Let ρ'' be the limit of $\{\rho''_n\}$. From properties 3 & 4 and definition of ρ''_n, we can deduce that $Acc(\rho'')$ is a fixpoint of G_3. Since ρ' is the greatest fixpoint of G_3, we have:

$$Acc(\rho') \sqsubseteq Acc(\rho'') \sqsubseteq \rho'.$$

On the other hand, $Acc(\rho') = \rho'$, by proposition 3. Thus, $\rho' = Acc(\rho'')$. Let us now prove properties 2-4. The first property is a consequence of the definitions of sequences $\{\rho''_n\}$, $\{\pi''_n\}$ and $\{\sigma_n\}$.

property 2

By induction on n: First, we have $\rho'_0 = \rho''_0$. Let us recall the definitions of ρ'_n and ρ''_n.

$$\begin{array}{rcl} \rho'_{n+1} & = & \rho'_n \sqcap split(Acc(\rho'_n), Acc(\rho'_n)) \\ \rho''_{n+1} & = & \rho''_n \sqcap split(\pi''_n \setminus \sigma_n, \rho''_n) \cup \sigma_n \cup (\rho''_n \setminus \pi''_n) \end{array}$$

Assume that $\rho'_n \sqsubseteq \rho''_n$. We show that $\rho'_{n+1} \sqsubseteq \rho''_{n+1}$. We have:

$$\begin{array}{rcl} \rho'_{n+1} & \sqsubseteq & \rho''_n \sqcap split(Acc(\rho''_n), Acc(\rho''_n)) \quad \text{(by monotonicity of } Acc \text{ and } split) \\ & = & \rho''_n \sqcap split(Acc(\rho''_n), \rho''_n) \quad \text{(by lemma 4 part 1)} \\ & = & \rho''_n \sqcap split(\pi''_n, \rho''_n) \cup \rho''_n \sqcap split(Acc(\rho''_n) \setminus \pi''_n, \rho''_n) \quad \text{(by corollary 1, part 2)} \\ & \sqsubseteq & \rho''_n \sqcap split(\pi''_n, \rho''_n) \cup (\rho''_n \setminus \pi''_n) \quad \text{since,} \\ & & \qquad split(Acc(\rho''_n) \setminus \pi''_n, \rho''_n) \subseteq split(Acc(\rho''_n), \rho''_n) \setminus \pi''_n \textit{ and} \\ & & \qquad\qquad split(Acc(\rho''_n), \rho''_n) \sqsubseteq \rho''_n. \\ & = & \rho''_{n+1} \quad \text{since} \\ & & \qquad split(\pi''_n, \rho''_n) = split(\pi''_n \setminus \sigma_n, \rho''_n) \cup split(\sigma_n, \rho''_n) \\ & & \qquad\qquad = split(\pi''_n \setminus \sigma_n, \rho''_n) \cup \sigma_n \end{array}$$

property 3

We have, $\rho''_{n+1} = \sigma_n \cup (\rho''_n \setminus \pi''_n) = \rho''_n$. To prove the second part, we perform an induction on k: if $\rho''_{n+1} = \rho''_n$ then

$$\begin{array}{rcl} \pi^k_{n+1} & = & \pi^k_n \\ \sigma^k_{n+1} & = & \sigma^k_n \end{array}$$

property 4

If $X \in \pi''_n$, then $X \in \sigma_n$ and $post_{\rho''_n}(X) \subseteq \pi''_n = \sigma_n$

Proof of the algorithm

Partial correctness

Let ρ'''_p, π'''_p and σ'''_p be sequences obtained after each step of iteration. We have:

$\rho'''_0 = \rho_{init} \qquad \pi'''_0 = \{[q_{init}]_{\rho'''_0}\}; \sigma'''_0 = \emptyset$

$\rho'''_{p+1} = \rho'''_p \setminus \{X\} \bigcup split(X, \rho'''_p)$

$\sigma'''_{p+1} =$ if $X = split(X, \rho'''_p)$ then $\sigma'''_p \bigcup \{X\}$ else $\sigma'''_p \setminus pre_{\rho'''_p}(X)$

$\pi'''_{p+1} =$ if $X = split(X, \rho'''_p)$ then $\pi'''_p \bigcup post_{\rho'''_p}(X)$ else $\pi'''_p \setminus \{X\} \bigcup post_{\rho'''_p}(q_{init})$

Several sequences could be considered, corresponding to the choice of X (line (3) of the second algorithm). But we show they have the same limit, $Acc(\rho'')$. Let be ρ''' be the limit of $\{\rho'''_n\}$. The proof is based on the following properties:

$\forall \{\rho'''_i\}.\forall n.\exists k$ *such that* $\rho''_k \sqsubseteq \rho'''_n$,

We perform an induction on n. We have $\rho''_0 = \rho'''_0$. Let $\{\rho'''_i\}$ be a sequence and let n, k such that $\rho''_k \sqsubseteq \rho'''_n$. We perform a refinement step of the algorithm. Let $X \in \pi'''_n \setminus \sigma_n$. If $X = split(X, \rho'''_n)$, then $\rho''_k \sqsubseteq \rho'''_{n+1}$ else $\rho''_{k+1} \sqsubseteq \rho'''_{n+1}$. Finally, we can conclude that $Acc(\rho'') \sqsubseteq Acc(\rho''')$.

$\pi_n''' = \sigma_n''' \Longrightarrow Acc(\rho_n''') = \pi_n'''$,

Let X in π_n'''. Then $X \in \sigma_n$ and $post_{\rho_p'''}(X) \subseteq \sigma_n$. Thus, all the accessible states are stable.

ρ''' is a fixpoint of $\lambda\rho.G_3(Acc(\rho), Acc(\rho))$,

It is a direct consequence of the previous property. Thus, $Acc(\rho''') \sqsubseteq Acc(\rho'')$, by proposition 3.

Then, $Acc(\rho''') = Acc(\rho'')$.

Termination

From the assumption that $Q/\overline{\rho}$ is finite, it is obvious that ρ cannot be indefinitely refined. The termination follows.