Software Verification Based on Linear Programming*

S. Dellacherie**, S. Devulder***, and J-L. Lambert†

GREYC, CNRS UPRESA 6072, Université de Caen,
BP 5186, 14032 Caen cedex, France
dellache@info.unicaen.fr, devulder@info.unicaen.fr, jll@info.unicaen.fr

Abstract. We introduce a new software verification method based on plain linear programming. The problematic is being given a software S and a property $\mathcal{P}$, to find whether there exists a path (i.e. a test sequence) of S satisfying $\mathcal{P}$, or a proof that $\mathcal{P}$ is impossible to satisfy.
The software S is modelized as a set of communicating automata which in turn is translated into a system of linear equations in positive numbers. Property $\mathcal{P}$ is then translated as extra linear equations added to this system.
We define the extended notion of flow-path (which includes the notion of path) permitting the automata to carry flows of data rather than undividable tokens. By applying linear programming in a sophisticated way to the linear system, it is possible, in time polynomial in the size of $(S, \mathcal{P})$, either to display a flow-path of S satisfying $\mathcal{P}$ or to prove that $\mathcal{P}$ is impossible to satisfy.
The existence of a flow-path does not always imply the existence of a path, as it can be non-integer valued. Yet, on all our modelized examples, the study of the flow-path solution always permitted either to display a path satisfying $\mathcal{P}$ or to underscore a reason proving $\mathcal{P}$ to be impossible to satisfy.
The first part of this document introduces the theoretical background of our method. The second part sums up results of the use of our method on some systems of industrial size.

Keywords: software, concurrent program, distributed system, formal verification, validation, simulation, test case generation, proof, linear programming, integer programming.
URL: http://www.info.unicaen.fr/lpv

* This work was partly supported by CNET under grant n#95 5B 046, by Région Basse-Normandie and CNRS under contract CON950207DR19
** PhD Student, CNET and University of Caen
*** PhD Student, CNRS and University of Caen
† Professor, University of Caen

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1147–1165, 1999.

1 Introduction

Linear programming gathers means for optimizing a linear function subject to a set of linear constraints in real positive numbers [faq]. Due to its efficiency and the range of its applications areas (production management, networks organization, resources planification, ...), it is the cornerstone of combinatorial optimization techniques. Still at present, due to the arising of interior-point algorithms, linear programming attracts an important research power, both in theoretical and practical fields. Linear programming algorithms now routinely solve on a desktop computer problems involving hundreds of thousand rows per hundreds of thousand columns.

Linear programming is also a classical mean for tackling the much more difficult integer programming problem where all (or part of) variables need to have an integer value. A wide range of techniques are available for this particular problem.

Attempts to use linear programming in the verification domain is not new. The Petri-nets community has been using linear programming for almost fifteen years. Yet, it seems to be only used on very constrained models, not well suited for the modelization of real-world systems [ES92] or for the generation of the set of the model invariants, which is constructed blindly and whose size increases very quickly [LM89] [CHP92]. More recently, Corbett and Avrunin [CA95] studied the use of a general purpose integer programming algorithm directly on a communicating automata model. Unfortunately, the use of a general purpose integer programming algorithm destroys the efficiency of mere linear programming, and prevents the possibility of constructing proofs on the model.

Thus, to our knowledge, none of these attempts make a complete use of the powerful theoretical background bind to linear programming: the duality theory. We will overview in this document how the use of linear programming duality permits to obtain a non-trivial and efficient (polynomial time) completeness theorem on the proof of existence of a flow-path or the proof of non-existence of any flow-path.

Furthermore, as we will see in this document, our method shares very few features with other existing methods. Unlike *model-checking* techniques (see for example [McM93][Kur92] [Hol97]) our method works directly on the automata model without constructing a representation of the reachability graph, thus enabling the handling of huge models. Unlike *theorem-proving* techniques (see for example [Abr95][ORR$^+$96] [GH93]), the proofs given by our method are automatic and fastly computed directly on the automata model.

Now considering its negative points, the main drawback of the method is the fact that if a flow-path is proved to exist, it does not always induce the existence of a path (i.e. a test sequence on the automata): linear programming works with real numbers, and the proposed flow-path is not always integer-valued. When such a fractional solution occurs, one has to make the property more precise and/or slightly transform the model in order to conclude.

Yet, as we will see in this document, a flow-path is not "very far" from a path. In practice on the automata models we have studied, the careful reading of the flow-path solution has always led us to find a path satisfying the property or a proof that the property is impossible to satisfy.

The first part of this document presents the theoretical background of the method. The first section formally defines the automata model, the set of properties handled, the synchronization and flow-synchronization rules. The second section explains how the automata model and a property are translated into a system of linear equations, and introduces the main theoretical result obtained using linear programming theory.

The second part of this document summarizes the results of the use of the method on three different systems:

1. A generic telephony system. We used our method on instances using from 5 to 7 telephones communicating through fifo-channels of size 3 to 7. Such a model uses more than 800 automata and 2500 different synchronization messages. The corresponding state space is more than 10^{40} wide. Resolutions took a few tens of minutes.
2. A generic access control system. We used our method on instances using up to 20 cards, 8 doors and 4 buildings, communications being done using buffers. Such a model uses 230 automata and 2800 different synchronization messages. The corresponding state space is more than 10^{52} wide. Resolutions took a few hours.
3. A generic bus arbiter. The method was used on instances going up to 1200 cells. The state space is then about 10^{500} wide. Resolutions took a few tens of minutes.

On all three systems we have always been able for all properties checked, either to find a test-suite or a proof of impossibility.

Note that the method described herein is subject to a patent[1] and is about to be industrialized.

2 Theoretical Principles

2.1 The Model

Our linear programming verification method works with a communicating automata formalism. Using communicating automata is quite common in the verification domain (StateCharts, part of the SDL and UML formalisms, ...). They are easy to understand and to use while being powerful enough to modelize a

[1] patent #97 15217 registered on Dec. 3rd of 1997 and owned in common by France Telecom, the CNRS, and the University of Caen

large variety of software components. They also consitute a dynamic model of a software and are thus easily implementable and executable.

The use of communicating automata means that our verification method is mainly suited to verify the control part of a software, interactions between components of a software, and interactions between the control and the data of a software. A great part of verification needs in industrial software developments are of this kind.

Synchronized Automata The automata we use communicate via rendezvous, synchronizing themselves on messages carried by the transitions. We will call them in the remaining *synchronized automata*.

We modelize every possible component of the software with synchronized automata: the data, the control-flow, the communication channels (fifo-channels, stacks, ...). Of course, all these components need to have a finite domain, furthermore not too large. Yet we will see in the second part of this document that rather huge models can be handled by our verification method. Moreover, abstraction principles such as described in [CC77] can also be used in our approach to handle larger validity domains.

Figure 1 presents a small example that we are going to use to illustrate the method all along this document.

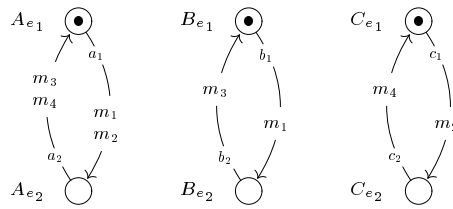


Fig. 1. A small example of synchronized automata.

Each automaton has states (the nodes, labeled A_{e_i}, B_{e_i} or C_{e_i}) and transitions (the arcs, labeled a_i, b_i or c_i). Each automaton has a single token that can move from state to state using the transitions. Transitions carry synchronization messages (m_1, m_2, m_3 and m_4). A transition may have multiple messages. For instance, arc a_1 bears two messages m_1 and m_2.

An automaton can go (i.e. move its token) from a state to another if and only if there exists a transition between those two states and all the synchronization messages present on that transition can be emitted. A message can be emitted if and only if all automata that know the message (ie. that have at least one transition carrying this message) can use simultaneously a transition carrying this message.

For example, automaton A can go from state A_{e_1} to state A_{e_2} if and only if both synchronization messages m_1 and m_2 can be emitted. This is possible if, for example, automata B and C are (ie. have their token) respectively in states B_{e_1} and C_{e_1}. In this situation, the three automata will arrive in state A_{e_2}, B_{e_2} and C_{e_2} respectively.

On the other hand, if A is in state A_{e_1} while automaton B is in state B_{e_2}, we are in a deadlock situation. Using transition a_1 requires the ability of emitting message m_1 which is impossible. Indeed B knows m_1 —carried by b_1— but is not in a state where it can emit it. So transition a_1 cannot be used and both m_1 and m_2 cannot be emitted which means that transition c_1 cannot be used and that all the automata will have to stay on the same state, thus the deadlock situation.

Let us state the formal definition of a system of automata, the definition of the synchronization rule, and then the definition of a system of synchronized automata.

Definition (automata). A *system of automata* S is composed of N sub-systems S_n, $1 \leq n \leq N$ called *automata*, and of a set $M = \{m^k, 1 \leq k \leq |M|\}$ containing the *messages* m^k of S. Every automaton S_n is described by

1. the set $E_n = \{e_n^i,\ 1 \leq i \leq |E_n|\}$ of its states;
2. the set $A_n = \{a_n^j,\ 1 \leq j \leq |A_n|\}$ of its transitions;
3. the set of messages $M_n \subset M$ carried by A_n.

To every transition a_n^j of S_n is associated a unique starting state $e_n^{j_1} \in E_n$ and a unique arriving state $e_n^{j_2} \in E_n$. Every transition a_n^j of S_n carries a set of messages $M_n^j \subset M_n$. ♢

Definition (synchronization rule). Let us call *configuration* a mapping C which associates to every automaton S_n a unique state $e_n \in S_n$ called the *activated state* of S_n, and let us call *synchronization* a subset s of M. We then define the *synchronization rule* as follows: the synchronization s has S changed from configuration C to configuration C' if and only if $\forall S_n \in S$,

1. if $s \cap M_n = \emptyset$ then $C'(S_n) = C(S_n)$
2. if $s \cap M_n \neq \emptyset$ then $\exists a_n^j = (e_n^{j_1}, e_n^{j_2}) \in A_n$ such that
   (a) $M_n^j = s \cap M_n$
   (b) $e_n^{j_1} = C(S_n)$, $e_n^{j_2} = C'(S_n)$

We then say that transition a_n^j is *fired* and that messages $m \in M_n^j$ are *activated* during the change from C to C'. ♢

Definition (synchronized automata). A *system of synchronized automata* is a system of automata endowed with the synchronization rule.
Furthermore, let C and C' be two configurations of S. The change from C to C' by synchronization s defines a *step* (C, s, C') for S. A succession of steps $(C_0, s_0, C_0^{'}), \ldots, (C_{n-1}, s_{n-1}, C_n^{'})$ such that $C_{i+1} = C_i^{'}$ defines a *path* for S. ♢

The synchronization rule can be interpreted in two ways: an automaton which aims at firing a transition can be considered forcing the other automata to follow him, or having to ask the other automata the permission to do so. The appropriate interpretation has to be given at higher level by the semantic of the system.

To our knowledge this kind of synchronization rule we defined is not classical, as all automata have to get simultaneously in accordance to fire their transitions. Note however that more classical formalisms as Statecharts or the BLIF format were easily translated in our own formalism.

We now introduce a set of properties which can be verified with our method.

Accessibility Properties The kind of requests we will check on a system of synchronized automata corresponds to the classical set of accessibility (or reachability) properties. For every automaton, we give a set of states within which is the activated state at start, and a set of states within which we want the activated state to arrive. The question is then whether there exists or not a path connecting these two sets.

On our small example, such a request could be: having each automaton in its state A_{e_1}, B_{e_1} or C_{e_1} respectively, can automata A and B reach its state A_{e_2} and B_{e_2} respectively, C being in state C_{e_1} or C_{e_2}?

Such a set of requests can be formally stated as follow:

Definition (accessibility property). Let S be a system of synchronized automata. An *accessibility property* on S is a couple $\mathcal{P} = (\mathcal{C}, \mathcal{C}')$ of sets of states of S. $\diamondsuit$

Definition (path-satisfiability). An accessibility property $\mathcal{P} = (\mathcal{C}, \mathcal{C}')$ on S *has a path satisfying* $\mathcal{P}$ if and only if there exists a path in $k \in \mathbb{N}$ steps going from a configuration C_0 to a configuration $C_n^{'}$ such that $\forall S_n \in S$,

- if $E_n \cap \mathcal{C} \neq \emptyset$ then $C_0(S_n) \in \mathcal{C}$
- if $E_n \cap C' \neq \emptyset$ then $C_n^{'}(S_n) \in \mathcal{C}'$ $\diamondsuit$

The expressiveness of this set of properties is quite large if we consider the appending of "observing automata" to the system S. These observers permit to express the necessity of using a particular message before another, the necessity to avoid a particular message, to avoid a particular transition, etc. On such automata one can then state an accessibility request, and thus extend the amount of properties which can be expressed directly on S. It has been shown to have at least the ability of expressing *temporal logic safety formulae* [JPO95].

Sub-section 2.1 now introduces a different automata model based on a generalization of the synchronization rule. This new model will be useful to state the main theoretical result given later in this document.

Flow-Synchronized Automata Our method is based on mere linear programming, which means that results given by the linear programming solver won't always be integer results. This motivates the introduction (which will be fully relevant in section 2.2) of a non-integer variation of the synchronization rule and the definition of flow-synchronized automata. We begin with the introduction of various kinds of flows:

Definition (message-flow). A *message-flow* is a function f_m which associates to every message m of S a real quantity $f_m(m) \in [0,1]$. ♢

Definition (transition-flow). A *transition-flow* is a function f_a which associates to every transition a_n^j of S a real quantity $f_a(a_n^j) \in [0,1]$. ♢

Definition (state-flow). A *state-flow* is a function f_e which associates to every state e_n^i of S a real quantity $f_e(e_n^i) \in [0,1]$. ♢

We are now ready to state the definition of the non-integer variation of the synchronization rule:

Definition (flow-synchronization rule). Let

- A_n^m be the set of transitions of S_n carrying message m: $A_n^m = \{a_n^j \in A_n / m \in M_n^j\}$,
- E_n^{i+} be the set of transitions of S_n having e_n^i as starting state: $E_n^{i+} = \{a_n^j \in A_n / \exists e,\ a_n^j = (e_n^i, e)\}$,
- E_n^{i-} be the set of transitions of S_n having e_n^i as arriving state: $E_n^{i-} = \{a_n^j \in A_n / \exists e,\ a_n^j = (e, e_n^i)\}$.

Let us call

- *flow-configuration* a state-flow f_C such that $\forall S_n$, $\sum_{e_n^i \in S_n} f_C(e_n^i) = 1$ (i.e. the quantity of token on each automaton is equal to 1),
- *flow-synchronization* a pair $f_s = (f_m, f_a)$ such that $\forall m \in M$, $\forall S_n$, $A_n^m \neq \emptyset$ $\Rightarrow f_s(m) = \sum_{a_n^j \in A_n^m} f_a(a_n^j)$ (i.e. for all automata that know m, the quantity of m emitted is equal to the flow going through the transitions carrying m).

We define the *flow-synchronization rule* as follows: the flow-synchronization $f_s = (f_m, f_a)$ has S changed from f_C to $f_{C'}$ if and only if $\forall e_n^i$, the following equations hold:

1. $\sum_{a_n^j \in E_n^{i+}} f_a(a_n^j) \leq f_C(e_n^i)$ (i.e. the flow leaving e_n^i is not greater than the quantity of token which is on e_n^i),
2. $\sum_{a_n^j \in E_n^{i-}} f_a(a_n^j) \leq f_{C'}(e_n^i)$ (i.e. the flow arriving on e_n^i is not greater than the total amount of token which is on e_n^i),
3. $f_C(e_n^i) - \sum_{a_n^j \in E_n^{i+}} f_a(a_n^j) = f_{C'}(e_n^i) - \sum_{a_n^j \in E_n^{i-}} f_a(a_n^j)$ (i.e. the new quantity of token on e_n^i is the previous quantity plus the flow arriving on e_n^i and less the flow leaving e_n^i).

If $f(a_n^j) > 0$ then we say that transition a_n^j is *flow-fired* and that messages $m \in M_n^j$ are *flow-activated* during the change from f_C to $f_{C'}$. ♢

This flow-synchronization-rule then defines a new kind of automata:

Definition (flow-synchronized automata). A *system of flow-synchronized automata* is a system of automata endowed with the flow-synchronization rule. Furthermore, let f_C and $f_{C'}$ be two flow-configurations of S. The change from f_C to $f_{C'}$ by flow-synchronization f_s defines a *flow-step* $(f_C, f_s, f_{C'})$ for S. A succession of flow-steps $(f_{C_0}, f_{s_0}, f_{C_0'})$, ..., $(f_{C_{n-1}}, f_{s_{n-1}}, f_{C_n'})$ such that $f_{C_i'} = f_{C_{i+1}}$ defines a *flow-path* for S. ♢

We also have by extension of path-satisfiability:

Definition (flow-path-satisfiability). A property $\mathcal{P} = (\mathcal{C}, \mathcal{C}')$ is *flow-path-satisfiable* if and only if there exists a flow-path going from $\mathcal{C}$ to $\mathcal{C}'$ in a finite number of flow-steps. ♢

A system of flow-synchronized automata is a "continuous" version of the corresponding system of synchronized automata. On synchronized automata the quantity of information can be modelized for every automaton with a token that moves from state to state following the synchronization rule. On flow-synchronized automata, the information which is in quantity still equal to one token per automaton, can this time flow through states as would do a liquid, following the flow-synchronization rule.

The synchronization rule is obviously a special occurence of a flow-synchronization rule (it is an integer-valued flow-synchronization), which implies that all notions binded to this former rule are also special occurences of the equivalent notions binded to the latter rule (step, path, satisfiability, ...).

The flow-automata model will be used to state the theoretical result of section 2.2. We yet need one more automata model to fulfill this first section. This last model, derived from the two former ones, will be the one used in practice by our method.

The Storied Extension of Automata The idea is to "unfold" through time the automata model in order to have each synchronization step (or flow-synchronization step) mapped to a given time step. Thus, during a time step, every automaton will have to use one of its transitions in accordance with the (flow-)synchronization rule, or to use a special transition, named an ε-transition, which will leave the automaton in the same state.

Figure 2 shows the storied version of the previous small example, unfolded through 3 time steps. All automata are respectively in their state $A_{e_1}(0)$, $B_{e_1}(0)$ and $C_{e_1}(0)$ before the first synchronization occurs. Suppose the first synchronization is the empty-set: all automata will use an ε-transition to stay in the same state, but ready for the second synchronization: respectively state $A_{e_1}(1)$, $B_{e_1}(1)$ and $C_{e_1}(1)$. Now if the second synchronization includes the emission of

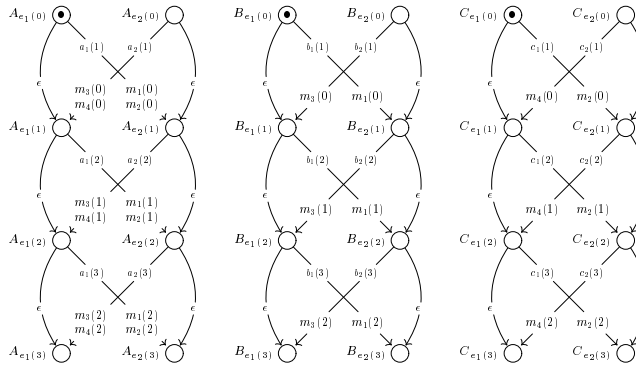


Fig. 2. storied automata of figure 1 on 3 time steps.

m_1 and m_2, this will make the automata to use respectively transitions $a_1(2)$, $b_1(2)$ and $c_1(2)$ to reach respectively $A_{e_2}(2)$, $B_{e_2}(2)$ and $C_{e_2}(2)$. And so on.

We see that there is a one to one correspondence between synchronization (or flow-synchronization) steps on the automata model and time steps on the storied extension. A solution on the latter will thus give directly a solution path (or flow-path) for the studied automata model. Section 2.2 explains how to use linear programming on this storied extension of the automata model.

Here is the formal definition of the storied extension of an automata model:

Definition (storied automata). Let S be a system of automata. We consider S on $T+1$ time steps as follow: for each automaton $S_n \in S$ we associate

1. to every value $t \in \{0, \dots, T\}$ and every state e_n^i, a state $e_n^i(t)$;
2. to every value $t \in \{1, \dots, T\}$ and every transition $a_n^j = (e_n^{j_1}, e_n^{j_2})$, a transition $a_n^j(t) = (e_n^{j_1}(t-1), e_n^{j_2}(t))$;
3. to every value $t \in \{1, \dots, T\}$ and every message $m^k \in M_{a_n^j}$, a message $m^k(t) \in M_{a_n^j(t)}$;
4. to every value $t \in \{1, \dots, T\}$ and every state e_n^i, an ϵ-transition $\epsilon_n^i(t) = (e_n^i(t-1), e_n^i(t))$.

The system thus constructed from S is called the *storied system of automata* S_T of S on T time steps. ♢

A storied system of automata S_T is clearly a special occurence of a system of automata, and both the definitions of synchronisation and flow-synchronisation rules are valid on S_T. The translation of an accessibility property from S to S_T is obvious: the starting set of configurations $\mathcal{C}$ is specified on the first time step $t = 0$, and the ending set of configurations $\mathcal{C}'$ is specified on the last time step $t = T$. Formally, it gives:

Definition. Let S be a system of automata and S_T the corresponding storied system. Let $(\mathcal{C}, \mathcal{C}')$ be an accessibility property on S. The corresponding accessibility property on S_T is given by $\mathcal{C} = \mathcal{C}(0)$ and $\mathcal{C}' = \mathcal{C}'(T)$. ♢

Furthermore we have the trivial following result binding a path in S and a path in S_T:

Proposition. Let S be a system of automata, S_T the corresponding storied system of automata. An accessibility property $(\mathcal{C}, \mathcal{C}')$ on S is (flow-)path satisfiable in n steps if and only if $(\mathcal{C}(0), \mathcal{C}'(T))$ is (flow-)path satisfiable on S_T with $T = n$. $\diamondsuit$

We are now ready to see how the storied system of automata is used in accordance with linear programming to verify accessibility properties.

2.2 The Use of Linear Programming

As stated in the introduction, linear programming is a very efficient mean for solving systems of linear equations in positive numbers when a real (i.e. not always integer) solution is searched for.

Here is a classical way of formally expressing the kind of problems treated by linear programming: the problem is to find an x^*, a vector of size n, optimal solution of

$$\begin{cases} \max c^t x \\ Ax = b \\ x \geq 0 \end{cases}$$

where A is an m rows, n columns matrix, c a vector of size n, b a vector of size m.

The optimization criteria $c^t x$ is optional, and the problem of only finding an x subject to $\{Ax = b, x \geq 0\}$ is of the same difficulty (it constitutes for example the first phase of the two-phases simplex algorithm). Our method relies mainly on finding such an x subject to a system of constraints that translates the properties of the automata system.

This implies first the necessity of constructing a system of linear constraints that catches the structural properties of the system of automata. This is the subject of the following sub-section.

Linear Constraints Drawn out of Automata The system of linear equations in positive numbers is drawn out of the storied extension S_T of the automata system S as follows:

Two kinds of equations are constructed: flow equations which translate the preservation of information on every state of every automaton of S_T, and synchronization equations which translate the synchronization (or flow-synchronization) rule for every message of S_T.

To this set of equations are added equations which translate the accessibility property, by forcing the value of some states at step $t = 0$ and at step $t = T$ to be equal to 1.

Here are the equations drawn out of the storied extension ($T = 3$) of our small example, the accessibility property being given by $\mathcal{C}(0) = \{A_{e_1}(0), B_{e_1}(0), C_{e_1}(0)\}$ and $\mathcal{C}'(3) = \{A_{e_2}(3), B_{e_3}(3), C_{e_3}(3)\}$.

The flow equations are for $i \in \{0, \ldots, 3\}$:

$$\begin{aligned} A_{e_1}(i) &= a_1(i+1) + \epsilon_{A_{e1}}(i+1) \\ A_{e_2}(i) &= a_2(i+1) + \epsilon_{A_{e2}}(i+1) \\ A_{e_1}(i+1) &= a_2(i+1) + \epsilon_{A_{e1}}(i+1) \\ A_{e_2}(i+1) &= a_1(i+1) + \epsilon_{A_{e2}}(i+1) \end{aligned}$$

$$\begin{aligned} B_{e_1}(i) &= b_1(i+1) + \epsilon_{B_{e1}}(i+1) \\ B_{e_2}(i) &= b_2(i+1) + \epsilon_{B_{e2}}(i+1) \\ B_{e_1}(i+1) &= b_2(i+1) + \epsilon_{B_{e1}}(i+1) \\ B_{e_2}(i+1) &= b_1(i+1) + \epsilon_{B_{e2}}(i+1) \end{aligned}$$

$$\begin{aligned} C_{e_1}(i) &= c_1(i+1) + \epsilon_{C_{e1}}(i+1) \\ C_{e_2}(i) &= c_2(i+1) + \epsilon_{C_{e2}}(i+1) \\ C_{e_1}(i+1) &= c_2(i+1) + \epsilon_{C_{e1}}(i+1) \\ C_{e_2}(i+1) &= c_1(i+1) + \epsilon_{C_{e2}}(i+1) \end{aligned}$$

The synchronization equations are for $i \in \{1, \ldots, 3\}$:

$$\begin{aligned} m_1(i) &= a_1(i) \quad m_2(i) = a_1(i) \\ m_1(i) &= b_1(i) \quad m_2(i) = c_1(i) \end{aligned}$$

$$\begin{aligned} m_3(i) &= a_2(i) \quad m_4(i) = a_2(i) \\ m_3(i) &= b_2(i) \quad m_4(i) = c_2(i) \end{aligned}$$

The property equations are:

$$\begin{aligned} A_{e_1}(0) = 1 \; B_{e_1}(0) = 1 \; C_{e_1}(0) = 1 \\ A_{e_2}(3) = 1 \; B_{e_2}(3) = 1 \; C_{e_2}(3) = 1 \end{aligned}$$

Let us see the formal definition of these three sets of equations:

Definition (system of equations). Let S be a system of automata, S_T the storied extension of S on T time steps, and $\mathcal{P} = (\mathcal{C}, \mathcal{C}')$ an accessibility property on S. We recall that for every automaton $S_n \in S$, E_n^{i+} is the set of transitions having e_n^i as starting state, E_n^{i-} is the set of transitions having e_n^i as arriving state, and A_n^m is the set of transitions carrying message m.

The system of linear equations $L(S_T, \mathcal{P})$ drawn out of S_T and $\mathcal{P}$ is given by the three following sets of equations:

– *flow equations:* $\forall S_n \in S$, $\forall t \in \{1, \dots, T\}$, $\forall e_n^i \in S_n$, we have

$$e_n^i(t-1) = \sum_{j_1 \in E_n^{i+}} a_n^{j_1}(t) + \epsilon_n^i(t)$$
$$e_n^i(t) = \sum_{j_2 \in E_n^{i-}} a_n^{j_2}(t) + \epsilon_n^i(t)$$

– *synchronization equations:* $\forall S_n \in S$, $\forall t \in \{1, \dots, T\}$, $\forall m \in M_n$, we have

$$m(t) = \sum_{j_3 \in A_n^m} a_n^{j_3}(t)$$

– *property equations:* $\forall S_n \in S$, we have $\sum_{e_n^i \in E_n} e_n^i(0) = 1$

$$\text{if } C \cap E_n \neq \emptyset \text{ then } \sum_{e_n^i \in \mathcal{C} \cap E_n} e_n^i(0) = 1$$
$$\text{if } C' \cap E_n \neq \emptyset \text{ then } \sum_{e_n^i \in \mathcal{C}' \cap E_n} e_n^i(T) = 1$$

♢

We clearly have a one to one correspondence between the use of a transition, a state or a message of the storied system of automata at a given time step and the value of the corresponding variable of the system of equations, whether the synchronization rule or the flow-synchronization rule is used.

If we use the synchronisation rule on S, we need to add to $L(S_T, \mathcal{P})$ positivity and integrality constraints on all its variables. If we use the flow-synchronisation rule on S, we need to add to $L(S_T, \mathcal{P})$ only the positivity constraints.

Proposition. Let S_T be a storied system of automata on T time steps, and $(\mathcal{C}(0), \mathcal{C}'(T))$ an accessibility property on S_T.
If the synchronization rule is used on S_T, then $(\mathcal{C}(0), \mathcal{C}'(T))$ is satisfiable on S_T if and only if $L(S_T, \mathcal{P})$ has a solution with all variables being positive and integer valued.
If the flow-synchronization rule is used on S_T, then $(\mathcal{C}(0), \mathcal{C}'(T))$ is satisfiable on S_T if and only if $L(S_T, \mathcal{P})$ has a solution with all variables being positive. ♢

Linear programming can handle systems of positive variables without integrality constraints. Thus we are only able to use linear programming on a system of automata using the flow-synchronization rule.

The solving of $L(S_T, \mathcal{P})$ gives either a flow-path or a proof (via the classical duality theory of linear programming) of the inexistence of any flow-path on a model of T stories. Subsection 2.2 will show that this completeness result on the existence or inexistence of flow-paths is independent of the number of steps T.

The Completeness Theorem Being given $L(S_T, \mathcal{P})$, the idea is to eliminate in an iterative way transitions, messages and states that, whatever the value of T is, can never be used if one wants to satisfy property $\mathcal{P}$. The fundamental point is that this iterative mechanism, called the *proof system*, works independently of the number of time steps T. This proof system permits to establish the following completeness theorem:

Theorem (completeness). Let S be a system of automata and $\mathcal{P}$ an accessibility property on S. The proof system establishes, in time polynomial in the size of $(S, \mathcal{P})$, the following alternative:

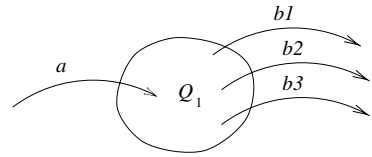
- it proves the existence of a *flow-path* on S satisfying $\mathcal{P}$ and gives an upper bound on the number of flow-steps;
- it proves the non-existence of any *flow-path* of S satisfying $\mathcal{P}$, and thus of any *path* of S satisfying $\mathcal{P}$. ♢

The details of the proof of this theorem are technically difficult and too long to be given in this document (the proof system is fully developed in [Dev99]). We yet can give an idea of how the proof system works.

Let us suppose $\mathcal{P} = (C_0, C_f)$ (we ask the system to go from a configuration C_0 to a configuration C_f). The proof system is made of three kind of inferences. Each inference can deduce from the conclusions of the previous one that some data (transitions, states or messages) of the system are useless and can be eliminated safely, or that the accessibility property is impossible to satisfy.

First kind of inference:

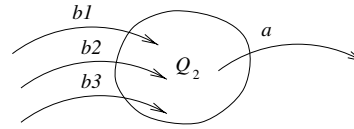
If one can find a set of states Q_1 such that any entering transition (say a) must flow-synchronize with another one (say b_1) that goes out of this set,



then we infer: if $C_0 \cap Q_1 = \emptyset$ then $C_f \cap Q_1 = \emptyset$, which implies that all the in-going and out-going transitions of Q_1 (here a, b_1, b_2, b_3) cannot be used. These transitions are proved impossible to use and are thus eliminated.

Second kind of inference:

Similarly, if one can find a set of states Q_2 such that any transition that goes out of it must flow-synchronize with another one that goes inside this set,



then we infer: if $C_f \cap Q_2 = \emptyset$ then $C_0 \cap Q_2 = \emptyset$, which implies that all the in-going and out-going transitions of Q_2 (here a, b_1, b_2, b_3) cannot be used. These transitions are proved impossible to use and are eliminated.

Third kind of inference:

If one can find a vector Y such that for any configuration C_n, we have $Y^tC_0 \leq Y^tC_n$, then we infer: if C_f satisfies $Y^tC_0 > Y^tC_f$, C_f cannot be accessed and the request is proved infeasible, or else we must have $Y^tC_0 = Y^tC_f$ and any transition that strictly increases Y^tC_n is forbidden and eliminated.

All inferences are computed using linear programming, which is a numerical algorithm. All data eliminated by an inference are yet proved impossible to use. The key idea of the proof is given by the well known following linear programming result:

Lemma (Farkas). Let A be a m rows, n columns matrix, and b a vector of size m. Then one and only one of the following statements is true:

1. there exists a vector $x \geq 0$ such that $Ax = b$;
2. there exists a vector y such that $y^tA \geq 0$, $y^tb < 0$. ♢

Based on this lemma, for all inferences linear programming gives a certificate (the y vector) justifying the elimination of the selected set of data.

The remaining question is whenever the existence of a flow-path is proved, how the existence of a true path can be proved and constructed.

The Effective Search of a Solution Path The proof of non-existence of any flow-path implies the non-existence of any path, which implies the accessibility property to be impossible to satisfy. In this case, the proof system is sufficient to conclude. On the contrary, whenever the proof of existence of a flow-path is given, it does not imply the existence of a path (as it is not always integer-valued), and it is thus not directly possible to know whether the accessibility property can be satisfied or not.

This constitutes the non-polynomial part of our method, and illustrates the intrinsic difficulty of software verification as the gap between linear programming and integer programming. Yet several remarks make this gap not so overwhelmingly difficult in our case. The remainder of this sub-section will give only pragmatic and rather subjective arguments to understand why it seems to work in practice.

The first thing to note is that the obtaining of a flow-path is very easy. It suffices to use a linear programming solver on the storied extension with enough time steps: the resulting solution is directly interpretable on the flow-synchronized automata model.

Having the flow-path solution, a general way of finding a path is to force some of the data to have integer values and to relaunch the linear programming solver. Iterating this technique permits eventually to find an all integer-valued solution, which is a path. A general strategy of this kind is yet heavily combinatorial, with a branching on every forced variable (0 or 1 value), and also combinatorial on the number of time steps.

However one has to notice that both the automata model and the flow-path have a semantic meaning. Taking into account this meaning in order to choose which data (transition, message or state) to force to a particular value, it reduces drastically the number of branching necessary to conclude. This guided branching strategy seems very efficient in practice when a path does exist: on all our examples, a few forcing steps were sufficient to find a true path.

When no paths exist, it gets more intricated and a guided branching strategy is not always sufficient. Indeed, the property has to be proved impossible whatever the number of time steps is. Yet, again guided by the semantic of the model and of the flow-path solution, the idea is then to find on the automata model the main reason permitting a non integer-valued solution to exist, and to slightly modify the automata model in order to eliminate this non integer solution (the theoretical meaning relying behind these slight modifications is given in [Del99b]).

Using this final technique together with some guided branching, we have always been able to conclude on all the examples we considered.

3 Some Case Studies

We summarize now some experiments we have done with our verification method on three different systems modelized with our automata-like formalism: a telephony system, an access control system and a bus arbiter.

All computations were done on a 168MHz UltraSparc2 with 256Mb of memory. The linear programming software used was CPLEX V4.0.

3.1 A Telephony System

This system modelizes mainly a connection/deconnection protocol between entities — telephones — which communicate between them using fifo channels. The system is not centralized, which means that any telephone can communicate directly with any other telephone by sending messages to its fifo channel. Some more details on this system are given at the end of article [DDL99].

The complexity of this telephony system is due to the increasing amount of different messages which can travel through the fifo channels. The connection/deconnection protocol uses 12 different messages (6 in emission and 6 in reception) for

every possible pair of telephones, these messages being possibly stored on any place of the corresponding fifo channel.

A telephone is made of two automata (one of 16 states, and one of 3 states). A fifo channel of size k is made of a write automaton ($k+1$ states), a read automaton (one state) and k memory-cell automata ($6 * n(n-1)+1$ states, n being the number of telephones).

The experiments were done with $n = 7$ telephones using fifo channels of size $k = 3$, or $n = 5$ telephones using fifo channels of size $k = 7$. The resulting systems is made of more than 800 automata and uses more than 2500 different synchronization messages. The state space is more than 10^{40} wide.

Here are some samples of properties for which a test-suite was found:

can phone#5 send a ABANDON to phone#6 ?
can phone#6 read a BUSY_LINE from phone#3 ?
can phone#3 send a STOPPING to phone#2 ?
can phone#4 be in its state S13_3.4 ?
can phone#1 and phone#4 be in conversation ?

Here are some samples of properties for which a proof of impossibility was found:

can phone#3 ring while offhooked ?
can phone#5 send BUSY_LINE to phone#1 while onhook ?
can phone#2 ring while nobody ever called it ?
can phone#7 be alone in communication ?
can phones #1, #4 and #6 be in circular communication ?

We also proved that with 5 telephones, fifo channels of size 6 were sufficient, by finding a test suite that fills a fifo of size 6 and by finding a proof that a fifo of size 7 can never be entirely filled.

In all cases, the computations took always less than an hour and used less than 200Mb of memory.

3.2 An Access Control System

The purpose of the system is to check in and out-goings of people through doors of some buildings. All doors have a reading-card device and communicate through buffers with a centralized controlling device. This centralized device controls the validity of the request (activated and authorized card for the given building), manages the opening-closing protocol of the door and records the entrance or exit of the card-bearer. An emergency circuit is also specified in order to open all doors of a given building in first priority, as well as a reset

protocol to end the emergency and put the doors of the building in a ready-to-work state. The full technical details on this case study are in [Del99a].

The size of the system grows cubiquely with the number of cards i, the number of doors j and the number of buildings k. The complexity of this system relies in the amount of data which has to be maintained while processing the opening-closing and the emergency-reset protocols (for example the centralized controller knows for every card in which building it is).

A door is made of 6 automata (of respectively 7, 3, 2, 2, 2, and 2 states). The centralized controller is made of many automata which represent mainly data. A door's buffer has 6 states, and the centralized controller's buffer has $2ij + 3j + 1$ states.

The experiments were done on a small instance made of 5 cards, 4 doors and 2 buildings, and then checked again on a huge instance made of 20 cards, 8 doors and 4 buildings. On this last instance the resulting system is made of 230 automata and uses more than 2800 different synchronization messages. The state space is more than 10^{52} wide.

Here are some samples of properties for which a test-suite was found:

can card#1 go in and out of building#2 with all doors locked behind him?
can card#1 go in-out of building#1 and then get in-out of building#2?
can door#3 be open with all its data in its expected state?
can card#4, being deactivated, get in building#2 after being reactivated?
can all doors of a building on emergency be opened without using a card?

Here are some samples of properties for which a proof of impossibility was found:

can card#1 be in building#2 but registered out of it?
can card#1, who entered building#1, enter building#2 without getting first out of building#1?
can door#3 be open with one of its data in an unexpected state?
can card#4, which is deactivated, get in building#2 without being reactivated?
can a door of a building under emergency stay locked?
can a door being on a building not on emergency get an emergency message?

The computations on the small instance took always a few minutes and used a few tens of Mb of memory; on the huge instance it took from 4 to 15 hours and used less than 200Mb of memory.

3.3 A Bus Arbiter

The bus arbiter is a hardware circuit whose purpose is to give to a single client an access to a resource shared by several different clients. This is a well known

example which has been treated with several techniques. The complete details on this case study are given in [Dev98].

The bus arbiter used for our experiments was a direct translation from the one given in the Xeve/Estrel package. It is made of several local cells which decide whether or not to allow the access to the resource for a client. The client claims the access by activating a cell which in turn activates a signal if it can access the resource. Cells are connected one to another to form a ring, allowing information to propagate. A token goes from one cell to the next on the ring. The cell who owns the token can access the bus if it wants to. If not, it tells the next cell that it can access the bus it it wants to. If this next cell doesn't want to use the bus, it tells the next cell and so on.

An arbiter with n cells has at least 2^n input configurations. We made experiments for systems with up to 1200 cells. The state space is then at least 10^{500} wide, and the computation took around one hour. The property checked was to know whether a client could access the bus at the same time as client#1. A proof of impossibility was found for all instances checked.

4 Conclusion

We have presented a new method for software verification which can be used either to exhibit test suites satisfying a property or proofs that a property is unsatisfiable.

In comparison with existing verification techniques, the main advantage of our method is its ability to handle huge models of automata without doing any abstraction, and to give at worse an answer which has always a semantic meaning helpful for further study. On the examples we modelized, the study of these answers has always permitted to conclude.

The modelization used (communicating automata) is dynamic and thus not far from an implementation. This means that the method is fitted for test suite generation. However, the relative poor computation time compared to probabilistic techniques indicates mostly an interest in finding probalistically hard test suites. The resolution time, though, is fast enough (both for path finding and proof finding) to allow interactivity to the model designer.

The resolution times given here can be improved easily. Our linear programming solver is not state-of-the-art and current solvers are now about 10 to 30 times faster while using less memory. The computer we use has a Specfp95 equal to 10, which is quite poor. Furthermore, we didn't work on optimizing the problem formulation given to the solver. The work involved is surely important but could decrease greatly the resolution time.

To finish, both test suites and (under some conditions) proofs can be found on a small instance of a generic specification and directly checked on a bigger instance. This is of great importance if the goal is to validate the specification and not only an instance of it.

References

[Abr95] J-R. Abrial. *The B-book*. Cambridge University Press, 1995.

[CA95] James C. Corbett and Georges S. Avrunin. Using integer programming to verify general safety and liveness properties. Technical report, University of Hawaii at Manoa, 1995.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CHP92] J.M. Couvreur, S. Haddad, and J.F. Peyre. parametrized resolution of families of linear systems. *RAIRO Recherche Operationnelle*, 26:183–206, 1992.

[DDL99] S. Dellacherie, S. Devulder, and J-L. Lambert. (technical version) software verification based on linear programming. Technical report, GREYC, universit de Caen, 1999.

[Del99a] S. Dellacherie. A case study: specification and verification of an access control system using the lpv technology. Technical report, GREYC, Universit de Caen, 1999.

[Del99b] S. Dellacherie. *Vrification logicielle base sur la programmation linaire*. PhD thesis, Universit de Caen, 1999. To appear.

[Dev98] S. Devulder. A comparison of lpv with other validation methods. Technical report, GREYC, Universit de Caen, 1998.

[Dev99] S. Devulder. *Un modle de preuve de logiciels fond sur la programmation linaire*. PhD thesis, Universit de Caen, 1999. To appear.

[ES92] J. Esparza and M. Silva. A polynomial-time algorithm to decide liveness of bounded free choice nets. *Theoretical Computer Science*, 102:185–205, 1992.

[faq] www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html.

[GH93] J.V. Guttag and J.J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag, 1993.

[Hol97] G.J. Holtzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[JPO95] Laeta Jategaonkar Jagadeesan, Carlos Puchol, and James E. Von Olnhausen. Safety porperty verification of esterel programs and applications to telecommunications software. In *Seventh Conference on Computer-aided verification*, 1995.

[Kur92] R. P. Kurshan. Automata-theoretic verification of coordinating processes. Technical report, ATT Bell Laboratories, 1992.

[LM89] J.B. Lasserre and F. Mahey. Using linear programming in petri net analysis. *RAIRO Recherche Operationnelle*, 23:43–50, 1989.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. Pvs: Combining specification, proof checking, and model checking. In *LNCS*, volume 1102, pages 411–414. Springer Verlag, 1996.