State Space Reduction based on Live Variables Analysis

Jean-Claude Fernandez

LSR/IMAG, BP 82, F-38402 Saint Martin d'Hères[1]

Marius Bozga, Lucian Ghirvu[2]

Verimag, Centre Equation, 2 avenue de Vignate, F-38610 Gières[1]

Abstract

The intrinsic complexity of most protocol specifications in particular, and of asynchronous systems in general, lead us to study combinations of static analysis with classical model-checking techniques as a way to enhance the performances of automated validation tools. In this context, we found that a non-trivial equivalence over the model states can be derived from the information on *live variables*. This equivalence exploits the unused dead values stored both in process variables and in communication queues and allows to reduce the model with a rather important factor. Furthermore, this reduction comes almost for free and is always possible to directly generate the quotient model without generating the initial one.

Key words: model checking, state space reduction, bisimulation, asynchronous communication, live variables analysis

1 Introduction

The central problem arising in the context of model based validation, model-checking and test generation, is the well known *state explosion problem*. In order to tackle it, one current direction of research focuses on adapting static analysis techniques such as the ones issued from compiler design and optimization [1,2] to this specific problem. The idea is to prevent state explosion

[1] E-mail: {Marius.Bozga, Jean-Claude.Fernandez, Lucian.Ghirvu}@imag.fr
[2] Work partially supported by Région Rhône-Alpes, France

by tacking into account as most as possible of the *useful* information that can be computed statically on the program under verification.

Model-checking and test generation algorithms are based on the central notion of *bisimulation* equivalence [3]. In practice, both of them benefit from model minimisation modulo *strong bisimulation* [4]. That is, strong equivalent models do not need to be distinguished and therefore, validation can always be done on the minimal model from one equivalence class while preserving all the validation results. This fact motivates us to consider static analysis techniques in the context of bisimulation equivalences. In this paper, we point out that a *non-trivial* equivalence can be derived from the information on *live variables*(in some given control point, a variable is live if its value could be used before being redefined on some path starting at that point). This equivalence, further called *live bisimulation*, exploits the unused *dead* values stored both in variables and in communication queues. It turns out that the live bisimulation is stronger than the strong bisimulation, but still allows to simplify the models with a rather important factor.

The reduction based on live bisimulation can be obtained with almost no additional cost: live variables need to be computed once in the beginning with an algorithm operating on the control graph of the program and then are used by specialized primitives (e.g, such as *Reset* defined in section 5) which operate on model states and which have a linear complexity with respect to the size of the state (the number of variables plus the number of messages in the communication queues). In fact, in the context of exhaustive state space exploration algorithms such primitives can be viewed as having constant operation time, similar to operations like state copying or state comparison.

Live bisimulation reduction is fully *orthogonal* to other techniques applied to deal with the state-explosion problem. For instance, enumerative simulation is at the basis of most of the validation algorithms. It is always possible to directly generate the quotient model with respect to live bisimulation, thus preserving all the relevant behaviours, without constructing the initial, complete model. This can be easily done for instance by considering every time a new model state is encountered some canonical representative from its equivalence class. In this way, equivalent states will be automatically identified as equals and explored once by the generation algorithm.

On-the-fly verification techniques such as [5] rely on enumerative depth-first traversals of the model and the meantime evaluation of the property (e.g., temporal logic formula, observer automaton, test purpose). In this case too, the property can be directly evaluated on the reduced model, if it is not explicitly dependent on variables or communication queue contents but only on the observable actions of the program. Nevertheless, if such dependencies exist, they can be normally removed by introducing auxiliary observable actions and by modifying the property accordingly. In addition, live bisimulation can be safely combined with partial order techniques [6] which reduce the model by avoiding to explore interleaving of independent actions.

In the symbolic model checking context [7] the information on live variables can be used to simplify the BDDs representing the transition relation and the sets of states. In fact, dead variables for a given control point provide non-trivial *don't care* conditions which can be exploited either directly to simplify any final or intermediate result, or even better, to improve primitives like successors or predecessor computation on symbolic representations. In particular, live bisimulation reduction can also be applied at each step of the minimal model generation algorithm [8] which involves the (backward) computation of predecessor states. This must be very important as the predecessor computation usually gives a lot of spurious unreachable states and which could be equivalent w.r.t. the live equivalence.

The idea of using static analysis to improve model checking was already investigated in different situations. In particular, live variables are currently used by almost all model-checkers (such as SPIN [9] or CADP [10]) in order to simplify the models during exhaustive simulations. Nevertheless, at the best of our knowledge, the current practice is limited to define some equivalence with respect to program variables, and none of them take into account the communication queue contents, when applicable. Related work was done also by [11] who propose a method to reduce the number of clocks in timed automata using live clocks and clocks equality analysis. Furthermore, in [12] was given a method which combines partial order reductions and static analysis of independent actions for SDL programs. Another interesting direction consists to find efficient representations of possible infinite queue contents and to exploit the static control structure when performing reachability analysis [13,14].

The paper is structured as follows. Section 2 presents the underlying model which is parallel processes asynchronous communicating via queues. Section 3 briefly remember the notion of live variables and some basic properties about them. In section 4 we introduce the *live equivalence* relation on states and show that is a bisimulation. An efficient way to identify live equivalent states using a canonical form is then presented in section 5. Some practical results obtained on a small example are given in section 6.

2 The model

2.1 Syntax

We consider systems consisting of asynchronous parallel composition of a number of *processes* that communicate through *parameterised signal* passing via a set of unbounded *FIFO queues* and operate on a set of shared *variables*. Formally, a system is a tuple:

$$P \ ::= \ (S, X, C, \prod_{i=1}^{n} p_i)$$

where S is the set of signals, X is the set of variables, C is the set of queues and $\{p_i\}_{i=1,n}$ are processes. Processes perform *actions* on queues and variables. They are described by *terms* of a simplified value-passing process algebra with the following syntax:

$$p \ ::= \ nil \mid \alpha.p \mid p + p \mid Z(e)$$

This syntax contains the usual operators from process algebra: *nil* denotes the empty process, $\alpha.p$ denotes p prefixed with action α, $p_1 + p_2$ is the nondeterministic choice between p_1 and p_2, and finally $Z(e)$ stands for recursion with the value passing of the expression e. To give semantics for terms we assume also the existence of a set of declarations:

$$Z(y) \ \triangleleft_{def} \ p$$

for each name Z which occurs in terms. As usual, y is a *parameter* of the definition of Z. The actions are simple *guarded commands* defined by the following syntax:

$$\alpha \ ::= \ [\ b\ ]\ \alpha_1$$

$$\alpha_1 \ ::= \ x := e \mid c!s(e) \mid c?s(x)$$

where $[\ b\ ]$ with b a boolean expression denotes the *guard*, $x := e$ denotes an *assignment* of the variable x to the expression e, $c!s(e)$ denotes the *output* to the queue c of the signal s with parameter the expression e and finally, $c?s(x)$ denotes the *input* from the queue c of the signal s and store of its parameter in the variable x.

Example 1 *Using the above syntax, a* counter *process which provides initialisation, consultation, increment and decrement primitives could be described as follows:*

$$
\begin{aligned}
Z_{counter}(x) \quad \lhd_{def} \quad & c?init(y).Z_{conter}(y) + \\
& c!val(x).Z_{conter}(x) + \\
& c!incr.Z_{counter}(x+1) + \\
& c!decr.Z_{counter}(x-1)
\end{aligned}
$$

2.2 Semantics

We give the semantics of systems in terms of *labelled transition systems*. We proceed in two steps: we start with the interpretation of the control part only, then we give the interpretation of control combined with data i.e, queue contents and variables.

The control semantics is described by the following rules which give a way to build the *control graph* of the system, from the parallel composition of processes. The states in this graph are tuples of terms $\prod_{i=1}^{n} p_i$, and the transitions are labelled by actions, as follows:

$$
\frac{-}{\alpha.p \xrightarrow{\alpha} p} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}
$$

$$
\frac{p[e/y] \xrightarrow{\alpha} p' \quad Z(y) \lhd_{def} p}{Z(e) \xrightarrow{\alpha} p'} \qquad \frac{p_j \xrightarrow{\alpha} p'_j \quad \forall i.\ i \neq j \quad p_i = p'_i}{\prod_{i=1}^{n} p_i \xrightarrow{\alpha} \prod_{i=1}^{n} p'_i}
$$

In this paper we restrict our attention to systems where the control graph is finite i.e, they contain only a finite number of states and transitions.

In order to interpret the data part we assume the existence of the universal domain D which contains the values of variables and signal parameters. We suppose that the boolean values $\{true, false\}$ and also the special *undefined* $\perp$ value are contained in D. We define *variable contexts* as being total mappings $\rho : X \to D$ which associate to each variable x a value v from the domain. We extend these mappings to expressions in the usual way. We define *queue contexts* as being also total mappings $\delta : C \to (S \times D)^*$ which associate to each queue c a sequence $(s_1, v_1), ..., (s_k, v_k)$ of *messages*, which are pairs (s, v) noted also by $s(v)$, where s is a signal and v is the carried parameter value. We assume also the existence of some special *unconsumable* message σ. This notion will be used in section 5 in order to undistinguish non-empty queues from which no signal could be consumed. The empty sequence is noted with ϵ.

The semantics of a system is now completed by the following rules which give a way to build a labelled transition system based on the control graph. States of this system are triples of the form (ρ, δ, p), where ρ is a variable context, δ is a queue context and p is a control state. Transitions are either *internal* and labelled with τ, when derived from assignments or signal inputs, either *visible* and labelled with $c!s(v)$ when derived from signal outputs:

$$\frac{p \xrightarrow{[b]\ x:=e} p' \quad \rho(b) = true \quad \rho(e) = v}{(\rho, \delta, p) \xrightarrow{\tau} (\rho[v/x], \delta, p')}$$

$$\frac{p \xrightarrow{[b]\ c!s(e)} p' \quad \rho(b) = true \quad \rho(e) = v \quad \delta(c) = w}{(\rho, \delta, p) \xrightarrow{c!s(v)} (\rho, \delta[w.s(v)/c], p')}$$

$$\frac{p \xrightarrow{[b]\ c?s(x)} p' \quad \rho(b) = true \quad \delta(c) = s(v).w}{(\rho, \delta, p) \xrightarrow{\tau} (\rho[v/x], \delta[w/c], p')}$$

2.3 Notations

Let us fix some set of queues $C_0 \subseteq C$.

Given control states p, p' we write $p \xrightarrow[C_0]{} p'$ if and only if p' can be reached from p in the control graph by a path which does not contain inputs from queues $c_0 \in C_0$. $\xrightarrow[C_0]{}$ is the reflexive and transitive closure of the transition relation including all the transition except those labelled with inputs from a queue of C_0. We note with $Post[\xrightarrow[C_0]{}]$ the post image function defined over sets of control states Q by the relation above i.e, $Post[\xrightarrow[C_0]{}](Q) = \{\ p' \mid \exists p.\ p \in Q\ \wedge\ p \xrightarrow[C_0]{} p'\ \}$.

Given action α and control states p, p' we write $p \xrightarrow[C_0]{\alpha} p'$ if and only if it is possible to reach p' from p in the control graph by a path ending with α and which does not contain prior inputs from queues $c_0 \in C_0$ i.e, there exists p'' such that $p \xrightarrow[C_0]{} p''$ and $p'' \xrightarrow{\alpha} p'$.

Finally, for queue $c \in C$ and signal $s \in S$ fixed, we define the post image function $Post[\xrightarrow[C_0]{c?s}]$ over sets of control states Q as follows $Post[\xrightarrow[C_0]{c?s}](Q) =$

$\{ \; p' \mid \exists p \; \exists b \; \exists x. \; p \in Q \; \wedge \; p \xrightarrow[C_0]{[b] \;\; c?s(x)} p' \}$. That is, it gives all the states which can be reached from Q by paths containing no inputs from queues $c_0 \in C_0$, and which end with the consumption of the signal s from the queue c. When C_0 is reduced to $\{c\}$, we use $p \xrightarrow[c]{} p'$ to stand for $p \xrightarrow[\{c\}]{} p'$.

3 Live variables analysis

In this section we briefly remember the definition of live variables and some general properties about them. We consider the sets of variables *used* and respectively *defined* by some action α. Intuitively, a variable is used either in the guards, in the right hand side of assignments or in outputs. A variable is defined either in the left-hand side of assignments or in inputs. Formally, the sets $\mathit{Use}(\alpha)$ and $\mathit{Def}(\alpha)$ are defined as follows:

α	$\mathit{Use}(\alpha)$	$\mathit{Def}(\alpha)$
$[\; b \;] \; x := e$	$\mathit{vars}(b) \cup \mathit{vars}(e)$	$\{x\}$
$[\; b \;] \; c!s(e)$	$\mathit{vars}(b) \cup \mathit{vars}(e)$	$\emptyset$
$[\; b \;] \; c?s(x)$	$\mathit{vars}(b)$	$\{x\}$

We consider now the set of *live* variables for some term p. Intuitively it is the smallest set of variables that might be used before they are redefined when interpreting the term. Formally, the sets $\mathit{Live}(p)$ are defined as the least fixpoint solution of the following equation system over sets of variables (the subscript μ indicates that we are interested in the least solution, which exists since we consider only finite-control systems):

$$\left\{ \begin{array}{rl} \mathit{Live}(\mathit{nil}) & =_\mu \emptyset \\ \mathit{Live}(\alpha.p) & =_\mu \mathit{Use}(\alpha) \cup \mathit{Live}(p) \setminus \mathit{Def}(\alpha) \\ \mathit{Live}(p+q) & =_\mu \mathit{Live}(p) \cup \mathit{Live}(q) \\ \mathit{Live}(Z(e)) & =_\mu \mathit{Live}(p[e/x]) \text{ where } Z(y) \lhd_{def} p \\ \mathit{Live}(\prod_{i=1}^{n} p_i) & =_\mu \bigcup_{i=1}^{n} \mathit{Live}(p_i) \end{array} \right.$$

An equivalent characterisation for the sets of live variables based on the control graph transitions and some basic properties are given by the following lemma.

Lemma 1

(i) $Live(p) = \bigcup_{p \xrightarrow{\alpha} p'} Use(\alpha) \cup Live(p') \setminus Def(\alpha)$.

(ii) $p \xrightarrow{\alpha} p' \quad \Rightarrow \quad Use(\alpha) \subseteq Live(p),\ \ Live(p') \setminus Def(\alpha) \subseteq Live(p)$.

Proof: (i) Structural induction over term structure. *(ii)* Immediate from *(i)*. ∎

Example 2 *If we consider the system described in figure 1, the variable x is live in state p_1 and dead in state p_2, while the variable y is dead in both of them.*

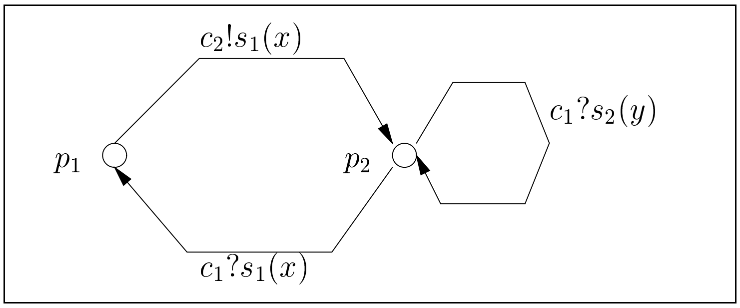


Fig. 1. A first example.

4 Live equivalence

First, we consider the variable context equivalence relation induced by the set of live variables: two variable contexts are equivalent if and only if the values assigned to each one of the live variables are pairwise identical. Formally, for each state p we define the *variable context equivalence* $\sim_p^{live}$ as follows:

$$\rho_1 \sim_p^{live} \rho_2 \quad \Leftrightarrow \quad \forall x \in Live(p).\ \rho_1(x) = \rho_2(x)$$

Example 3 *If consider the system described in figure 1 we have that contexts $[3/x, 5/y]$ and $[2/x, 7/y]$ are equivalent in state p_2, and not equivalent in state p_1 given that x is live in this state.*

We consider similar equivalence relations defined for queue contexts. Intuitively two queue contexts are equivalent if and only if they enable the same paths of transitions and if for each executed input the parameter values are identical if the receiving variable is live at the next state. Formally, we define the *queue context equivalences* $\approx_p^{live}$ to be the greatest fixpoint solution of the following equation system where for each state p we have (the subscript ν indicates that we are interested in the greatest solution):

$$\delta_1 \approx_p \delta_2 \ \Leftrightarrow_\nu \ \begin{cases} \bigwedge_{p \overset{[b]\ x:=e}{\longrightarrow} p'} \delta_1 \approx_{p'} \delta_2 \ \wedge \\ \bigwedge_{p \overset{[b]\ c!s(e)}{\longrightarrow} p'} \delta_1 \approx_{p'} \delta_2 \ \wedge \\ \bigwedge_{p \overset{[b]\ c?s(x)}{\longrightarrow} p'} \begin{cases} \delta_1(c) = \epsilon \ \wedge \ \delta_2(c) = \epsilon \ \wedge \ \delta_1 \approx_{p'} \delta_2 \\ \vee \\ \delta_1(c) = s(v_1).w_1 \ \wedge \ \delta_2(c) = s(v_2).w_2 \ \wedge \\ \quad (x \in \mathit{Live}(p') \Rightarrow v_1 = v_2) \ \wedge \ \delta_1[w_1/c] \approx_{p'} \delta_2[w_2/c] \\ \vee \\ \delta_1(c) = s_1(v_1).w_1 \ \wedge \ \delta_2(c) = s_2(v_2).w_2 \ \wedge \\ \quad s_1 \neq s \ \wedge \ s_2 \neq s \end{cases} \end{cases}$$

If the control graph is finite, the existence of the greatest fixpoint satisfying the equations above is ensured by the Tarski's fixpoint theorem given the monotonicity with respect to the relation inclusion at each state.

Example 4 *To illustrate the definition above, consider again the example from figure 1. For instance, we have that queue contexts* $[s_1(2).s_2(3)/c_1, s_1(4)/c_2]$ *and* $[s_1(2).s_2(5)/c_1, s_1(4)/c_2]$, *or* $[s_1(3)/c_1, \epsilon/c_2]$ *and* $[s_1(3)/c_1, s_1(3)/c_2]$ *are live equivalent in both states* p_1 *and* p_2. *Conversely, queue contexts* $[s_2(1).s_1(2)/c_1, \epsilon/c_2]$ *and* $[s_1(2)/c_1, \epsilon/c_2]$ *are not live equivalent neither in* p_1 *or* p_2.

Some interesting properties of the live equivalence on queue contexts are given by the following lemmas.

Lemma 2 $\delta_1 \approx_p^{live} \delta_2 \ \Leftrightarrow$

$$\forall p'.\, p \xrightarrow[C]{[b]\ c?s(x)} p' \ \Rightarrow \ \begin{cases} \delta_1(c) = \epsilon \ \wedge \ \delta_2(c) = \epsilon \ \wedge \ \delta_1 \approx_{p'} \delta_2 \\ \vee \\ \delta_1(c) = s(v_1).w_1 \ \wedge \ \delta_2(c) = s(v_2).w_2 \ \wedge \\ \quad (x \in \mathit{Live}(p') \Rightarrow v_1 = v_2) \ \wedge \ \delta_1[w_1/c] \approx_{p'} \delta_2[w_2/c] \\ \vee \\ \delta_1(c) = s_1(v_1).w_1 \ \wedge \ \delta_2(c) = s_2(v_2).w_2 \ \wedge \\ \quad s_1 \neq s \ \wedge \ s_2 \neq s \end{cases}$$

Proof: The proof is immediate from the definition of the live equivalence. ∎

Furthermore, it can be proven inductively that, two equivalent queue contents at some state p, are still equivalent after adding the same message to the same queue in both of them.

Lemma 3

$$\delta_1 \approx_p^{live} \delta_2,\ \ \delta_1(c) = w_1,\ \ \delta_2(c) = w_2\ \ \Rightarrow$$

$$\forall s\ \forall v.\ \ \delta_1[w_1.s(v)/c] \approx_p^{live} \delta_2[w_2.s(v)/c].$$

Proof: Because $\delta_1 \approx_p^{live} \delta_2$ the queue contexts δ_1 and δ_2 should satisfy the $\wedge$ clauses in the lemma 2 :

1: If $w_1 = \epsilon$ and $w_2 = \epsilon$ then the contexts $w_1.s(v) = w_2.s(v) = s(v)$ will fulfill the conditions of the second $\wedge$ clause so $\delta_1[w_1.s(v)/c] \approx_p^{live} \delta_2[w_2.s(v)/c]$.

2: If w_1 and w_2 satisfy both the second or the third $\wedge$ clause then adding them $s(v)$ does not change the fact they fulfill the second or the third $\wedge$ clause so $\delta_1[w_1.s(v)/c] \approx_p^{live} \delta_2[w_2.s(v)/c]$. ∎

In general, the property above does not hold for input actions: two equivalent queue contents do not remain necessary equivalent when taking away the same message in the same queue from both of them.

Example 5 *Consider the system* $p_1 \xrightarrow{c_1?s_1(x)} p_2 \xrightarrow{c_2!s_2(x)} p_3$. *The queue context* $[s_1(2).s_1(3)/c_1, \epsilon/c_2]$ *and* $[s_1(2)/c_1, \epsilon/c_2]$ *are equivalent in* p_1*, but if we remove* $s_1(2)$ *then the resulting queue contexts are not equivalent in* p_1*.*

We define now the *live equivalence* $\approx^{live}$ relation over global states: two states are equivalent if the control states are identical and both the variable context and the queue context are equivalent:

$$(\rho_1, \delta_1, p) \approx^{live} (\rho_2, \delta_2, p) \quad \Leftrightarrow \quad \rho_1 \sim_p^{live} \rho_2 \quad \wedge \quad \delta_1 \approx_p^{live} \delta_2$$

The next theorem gives the central result of the paper: the live equivalence is also a bisimulation relation over the global states. By definition, for all pairs $(\rho_1, \delta_1, p) \approx^{live} (\rho_2, \delta_2, p)$ we have:

$$\forall a\ \forall(\rho_1', \delta_1', p').\ (\rho_1, \delta_1, p) \xrightarrow{a} (\rho_1', \delta_1', p') \Rightarrow$$

$$\exists(\rho_2', \delta_2', p').\ (\rho_2, \delta_2, p) \xrightarrow{a} (\rho_2', \delta_2', p') \wedge (\rho_1', \delta_1', p') \approx^{live} (\rho_2', \delta_2', p')$$

$$\forall a\ \forall(\rho_2', \delta_2', p').\ (\rho_2, \delta_2, p) \xrightarrow{a} (\rho_2', \delta_2', p') \Rightarrow$$

$$\exists(\rho'_1,\delta'_1,p').\ (\rho_1,\delta_1,p)\xrightarrow{a}(\rho'_1,\delta'_1,p') \wedge (\rho'_1,\delta'_1,p') \approx^{live} (\rho'_2,\delta'_2,p')$$

Theorem 1 *The live equivalence $\approx^{live}$ is a bisimulation.*

Proof:

Let $(\rho_1,\delta_1,p) \approx^{live} (\rho_2,\delta_2,p)$ and let $p\xrightarrow{\alpha}p'$. We distinguish three different cases, depending on the type of α.

(1) $\alpha = [\ b\ ]\ x := e$

From $\rho_1 \sim_p^{live} \rho_2$, we deduce that $\rho_1(b) = \rho_2(b)$ and $\rho_1(e) = \rho_2(e) = v$. Suppose that $\rho_1(b) = \rho_2(b) = true$. Thus, for each $i = 1,2$ we have $(\rho_i,\delta_i,p)\xrightarrow{\tau}(\rho'_i,\delta_i,p')$ where $\rho'_i = \rho_i[v/x]$. We have to show that $\rho_1[v/x] \sim_{p'}^{live} \rho_2[v/x]$ and $\delta_1 \approx_{p'}^{live} \delta_2$

From lemma 1, if $y \in Live(p')$ and $y \neq x$ then $y \in Live(p)$. Because $\rho_1(y) = \rho_2(y)$ we have that $\rho'_1(y) = \rho'_2(y)$. Also, we have that $\rho'_1(x) = \rho'_2(x) = v$. In conclusion, $\forall z \in Live(p')$, $\rho'_1(z) = \rho'_2(z)$ and thus $\rho'_1 \sim_{p'}^{live} \rho'_2$. Finally, by definition of live equivalence on queue contexts, we have $\delta_1 \approx_{p'}^{live} \delta_2$.

(2) $\alpha = [\ b\ ]\ c!s(e)$

Let us assume that, for each $i = 1,2$ we have $\delta_i(c) = w_i$, $\rho_i(e) = v_i$, $\rho_i(b) = true$. As before, we have that $(\rho_i,\delta_i,p)\xrightarrow{c!s(v_i)}(\rho_i,\delta'_i,p')$ where $\delta'_i = \delta_i[w_i.s(v_i)/c]$.

Because $\rho_1 \sim_p^{live} \rho_2$ we deduce respectively $\rho_1(b) = \rho_2(b)$, $\rho_1(e) = \rho_2(e)$ and $\rho_1 \sim_{p'}^{live} \rho_2$. Furthermore, from definition of $\approx_p^{live}$, $\delta_1 \approx_p^{live} \delta_2$ implies $\delta_1 \approx_{p'}^{live} \delta_2$. Now, given that live equivalence is closed under output operations (see lemma 3) it results $\delta_1[w_1.s(v_1)/c] \approx_{p'}^{live} \delta_2[w_2.s(v_2)/c]$.

(3) $\alpha = [\ b\ ]\ c?s(x)$

Again, let us consider that for each $i = 1,2$ we have $\delta_i(c) = s(v_i).w_i$ and $\rho_i(b) = true$. Then, $(\rho_i,\delta_i,p)\xrightarrow{\tau}(\rho'_i,\delta'_i,p')$ where $\rho'_i = \rho_i[v_i/x]$ and $\delta'_i = \delta_i[w_i/c]$

As before, because $\rho_1 \sim_p^{live} \rho_2$ we have that $\rho_1(b) = \rho_2(b)$ and $\rho_1[v_1/x] \sim_{p'}^{live} \rho_2[v_2/x]$. By definition $\delta_1 \approx_p^{live} \delta_2$ implies $v_1 = v_2$ if $x \in Live(p')$ and $\delta_1[w_1/c] \approx_{p'}^{live} \delta_2[w_2/c]$. ∎

An immediate consequence of the previous theorem is the following one, stating that the live equivalence is stronger than the strong bisimulation. Clearly, we do not have equality because strong bisimulation might identify global states as bisimilar even if they have distinct control states.

Theorem 2 $\approx^{live} \subseteq \approx^{strong}$.

5 Reset equivalence

We would like to have a more efficient way to check that two contexts are live equivalent than to directly check the definition. Here we investigate the possibility to transform contexts into some *canonical form* preserving the live equivalence. One way to do this is using the family of *Reset* functions, defined below.

The $Reset^v$ function on variable contexts ρ basically sets the value of the dead variables to the fixed *undefined* $\perp$ value, depending on the current control state p. A *reset equivalence* can be defined over variable contexts as follows: two contexts are reset equivalent if they give the same result when reset. Formally, we have:

$$Reset^v(p, \rho) = \rho^* \qquad \rho^*(x) = \begin{cases} \rho(x) \text{ if } & x \in Live(p) \\ \perp & \text{otherwise} \end{cases}$$

$$\rho_1 \sim_p^{reset} \rho_2 \quad \Leftrightarrow \quad Reset^v(p, \rho_1) = Reset^v(p, \rho_2)$$

It is straightforward to prove that the live equivalence and the reset equivalence on variable contexts are identical, i.e, the following lemma holds.

Lemma 4 $\forall p.\ \sim_p^{reset} = \sim_p^{live}$.

In order to define a similar $Reset^q$ function for queue contexts we start by introducing some auxiliary definitions. We define the local *reset* function for the queue c given an initial non-empty set of control states Q. Given the content w of the queue c, this function basically rewrites it and *forgets* the signal parameters which are detected statically to be unused, when execution starts somewhere in Q. With σ denoting the *unconsumable* message, the function is recursively defined as follows:

$$
reset(Q, c, \epsilon) = \begin{cases} \sigma & \text{if } \forall s \ \ Post[\xrightarrow[c]{c?s}](Q) = \emptyset \\ & \\ \epsilon & \text{otherwise} \end{cases}
$$

$$
reset(Q, c, s(v).w) = \begin{cases} \sigma \quad \text{if } \ Post[\xrightarrow[c]{c?s}](Q) = \emptyset \\ \\ s(v).reset(Post[\xrightarrow[c]{c?s}](Q), c, w) \\ \qquad \text{if } \ \exists p \in Q.\ p \xrightarrow[c]{[b]\ \ c?s(x)} p' \ \wedge \ x \in Live(p') \\ \\ s(\perp).reset(Post[\xrightarrow[c]{c?s}](Q), c, w) \\ \qquad \text{if } \ \forall p \in Q.\ p \xrightarrow[c]{[b]\ \ c?s(x)} p' \ \Rightarrow \ x \notin Live(p') \end{cases}
$$

Some interesting properties about the local *reset* function are given by the following lemma.

Lemma 5 *For any Q, Q_1, Q_2, c, w, w_1, w_2 the following statements hold :*

(i) $reset(Q, c, w) = reset(Post[\xrightarrow[c]{}](Q), c, w)$.

(ii) $reset(Q, c, w_1) = reset(Q, c, w_2) \ \Rightarrow$

$\qquad \forall Q' \subseteq Q \ \ \ reset(Q', c, w_1) = reset(Q', c, w_2)$.

(iii) $reset(Q_1, c, w_1) = reset(Q_1, c, w_2), reset(Q_2, c, w_1) = reset(Q_2, c, w_2) \ \Rightarrow$

$\qquad reset(Q_1 \cup Q_2, c, w_1) = reset(Q_1 \cup Q_2, c, w_2)$

Proof: (i) The proof is immediate from the fact that for any set of control states Q, channel c and signal s we have $Q \subseteq Post[\xrightarrow[c]{}](Q)$ and $Post[\xrightarrow[c]{c?s}](Q) = Post[\xrightarrow[c]{c?s}](Post[\xrightarrow[c]{}](Q))$. *(ii), (iii)* The proof can be done by induction on the maximal size of the sequences w_1 and w_2. ∎

We define the $Reset^q$ function and the *reset equivalence* on queue contexts using local *reset* functions defined before. For a given control state p we consider:

$$
Reset^q(p, \delta) = \delta^* \qquad \delta^*(c) = reset(\{p\}, c, \delta(c))
$$

$$
\delta_1 \approx_p^{reset} \delta_2 \ \ \Leftrightarrow \ \ Reset^q(p, \delta_1) = Reset^q(p, \delta_2)
$$

Example 6 *For the example given in figure 1 we can check that queue contexts* $\delta_1 = [s_1(2).s_2(3)/c_1, s_1(4)/c_2]$ *and* $\delta_2 = [s_1(2).s_2(5)/c_1, s_1(4)/c_2]$*, which are live equivalent, are also reset equivalent in both states* p_1 *and* p_2*. We have for both* $i = 1, 2$ *that* $Reset^q(p_i, \delta_1) = Reset^q(p_i, \delta_2) = [s_1(2).s_2(\bot), \sigma]$.

The following lemma gives the relation between the live equivalence and the reset equivalence on queue contexts. In general, the reset equivalence is stronger than the live equivalence. This is explained by the fact that local resets are based on a conservative assumption about other queues: *the inputs from other queues are always considered enabled.* However, in the special case when the input actions from a queue do not enable input actions from other queues, the live equivalence and the reset equivalence become identical.

Recall that C is the set of all queues defined in the system (see section 2). We define queue c to be *reset-independent* from queue c' if and only if :

$$\forall p\ \forall q\ \forall q'.\ \ p \xrightarrow[C]{[b']c'?s'(x')} q,\ q \xrightarrow[C]{[b]c?s(x)} q' \ \Rightarrow$$

$$\exists p'.\ \ p \xrightarrow[C]{[b]c?s(x)} p',\ p' \xrightarrow[C]{[b']c'?s'(x')} q',\ x \in Live(p') \Leftrightarrow x \in Live(q')$$

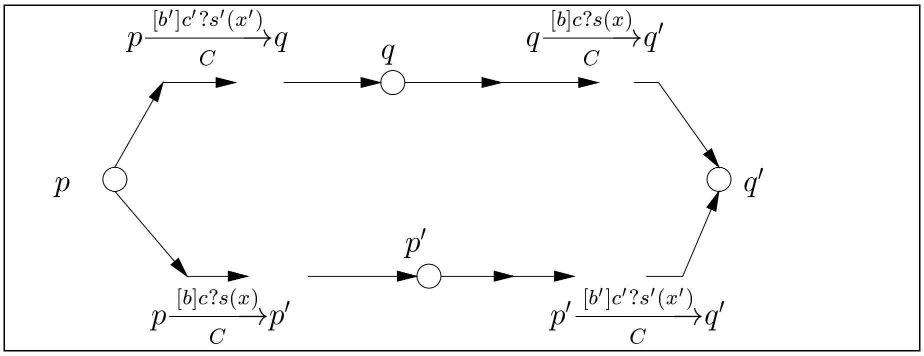


Fig. 2. Reset-independent queues.

Lemma 6

(i) $\forall p.\ \ \approx_p^{reset} \subseteq \approx_p^{live}$.

(ii) if the queues are reset-independent then $\forall p.\ \ \approx_p^{reset} = \approx_p^{live}$.

Proof: (i) The proof consists of checking that the reset equivalences $\approx_p^{reset}$ satisfies the fixpoint equations defining the live equivalence. This can be easily done given the results established by lemma 5. Then because the live equivalences are the greatest fixpoint it is clear that $\approx_p^{reset} \subseteq \approx_p^{live}$ for all control states p. *(ii)* The proof is done by induction on the maximal size of queue contexts. The initial step is straightforward: empty queue contexts are both live and reset equivalent. The induction step is proven by contradiction. Let consider $\delta_1 \approx_p^{live} \delta_2$, some queue c fixed and let assume that $reset(\{p\}, c, \delta_1(c)) \neq reset(\{p\}, c, \delta_2(c))$. We distinguish the following cases:

(1) $\delta_1(c) = \epsilon,\ \delta_2(c) = \epsilon \ \Rightarrow\ reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c))$ which is a contradiction with the reset hypothesis.

(2) $\delta_1(c) = \epsilon,\ \delta_2(c) = s_2(v_2).w_2$

(2.1) $\forall s.\ \ Post[\xrightarrow[c]{c?s}](\{p\}) = \emptyset \ \Rightarrow\ reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c)) = \sigma$ which is a contradiction with the reset hypothesis.

(2.2) $\exists s.\ \ Post[\xrightarrow[c]{c?s}](\{p\}) \neq \emptyset \ \Rightarrow\ \ Post[\xrightarrow[C]{c?s}](\{p\}) \neq \emptyset$

Because the queues are reset-independent we can choose the path ending with an input of the form $c?s$ such that it does not contain inputs from other queues. But, in such case we have a contradiction with the live equivalence hypothesis $\delta_1 \approx_p^{live} \delta_2$.

(3) $\delta_1(c) = s_1(v_1).w_1,\ \delta_2(c) = s_2(v_2).w_2$

(3.1) $Post[\xrightarrow[c]{c?s_1}](\{p\}) = \emptyset,\ Post[\xrightarrow[c]{c?s_2}](\{p\}) = \emptyset \ \Rightarrow reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c)) = \sigma$ which is a contradiction with the reset hypothesis.

(3.2) $Post[\xrightarrow[c]{c?s_1}](\{p\}) \neq \emptyset,\ Post[\xrightarrow[c]{c?s_2}](\{p\}) = \emptyset$ leads to a contradiction with the live equivalence hypothesis, as in the case *(2.2)*.

(3.3) $Post[\xrightarrow[c]{c?s_1}](\{p\}) \neq \emptyset,\ Post[\xrightarrow[c]{c?s_2}](\{p\}) \neq \emptyset$

(3.3.1) $s_1 \neq s_2$ leads to a contradiction with the live equivalence hypothesis, as before.

(3.3.2) $s_1 = s_2 = s$ and $v_1 \neq v_2$ and $\exists p \xrightarrow[c]{[b]\ \ c?s(x)} p' \ \ x \in Live(p')$ leads to a contradiction with the live equivalence hypothesis, as before.

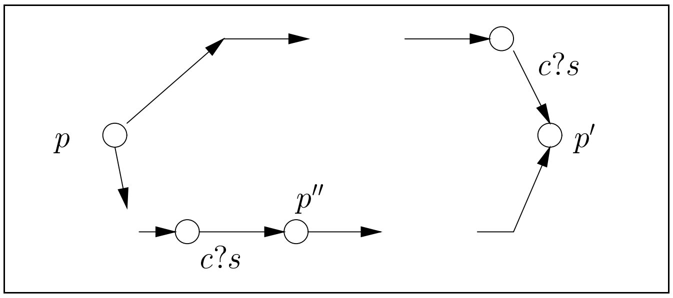
(3.3.3) $s_1 = s_2 = s$ and $v_1 = v_2 = v$ or $\forall p \xrightarrow[c]{[b]\ \ c?s(x)} p' \ \ x \notin Live(p') \ \Rightarrow$

$$reset(\{p\}, c, s_1(v_1).w_1) = s(v).reset(Post[\xrightarrow[c]{c?s}](\{p\}), c, w_1) \text{ and}$$

$$reset(\{p\}, c, s_2(v_2).w_2) = s(v).reset(Post[\xrightarrow[c]{c?s}](\{p\}), c, w_2)$$

From the reset hypothesis we must have that $reset(Post[\xrightarrow[c]{c?s}](\{p\}), c, w_1) \neq reset(Post[\xrightarrow[c]{c?s}](\{p\}), c, w_2)$. Moreover, from lemma 5 we can infer that there exists $p' \in Post[\xrightarrow[c]{c?s}](\{p\})$ such that $reset(\{p'\}, c, w_1) \neq reset(\{p'\}, c, w_2)$.

So, there exists a path starting in p and ending in p' by an input of the form $c?s$ and which does not contain any prior input from the queue c (see figure below). Now, given the reset-independence hypothesis we can find another path from p leading to p', containing exactly the same inputs, reordered such that the one of form $c?s$ is the now the first in the path. Let p'' be the state after executing the $c?s$ input in this new path.



In state p'', given the live hypothesis we have $\delta_1[w_1/c] \approx^{live}_{p''} \delta_2[w_2/c]$. Also, from $reset(\{p'\}, c, w_1) \neq reset(\{p'\}, c, w_2)$ and because there is a path from p'' to p' which does not contain inputs from the queue c we have also that $reset(\{p''\}, c, w_1) \neq reset(\{p''\}, c, w_2)$ (see also lemma5). So we end up with the induction hypotheses at state p'', now for contexts smaller with one message. The proof is complete. ■

Example 7 *Let consider the example given in figure 3. Following the definition, queues c_1 and c_2 are not reset-independent. Thus, there exist queue contents such as $\delta_1 = [s_2(1)/c_1, s_2(2)/c_2]$ and $\delta_2 = [s_2(3)/c_1, s_2(7)/c_2]$ which are live equivalent in state p_1 but not reset equivalent at that state. That is, $Reset^q(p_1, \delta_1) = [\sigma/c_1, s_2(2).\sigma/c_2] \neq [\sigma/c_1, s_2(7).\sigma/c_2] = Reset^q(p_1, \delta_2)$.*

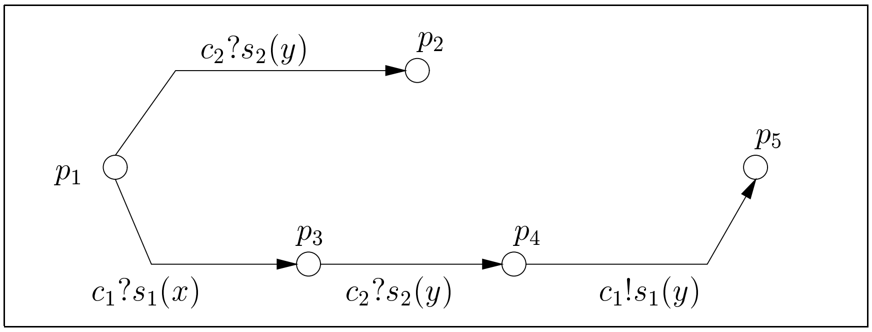


Fig. 3. A second example

Finally, we define the *reset equivalence* over global states as follows:

$$(\rho_1, \delta_1, p) \approx^{reset} (\rho_2, \delta_2, p) \quad \Leftrightarrow \quad \rho_1 \sim^{reset}_p \rho_2 \quad \wedge \quad \delta_1 \approx^{reset}_p \delta_2$$

The link between the live and reset equivalence over global states is established by the following theorem.

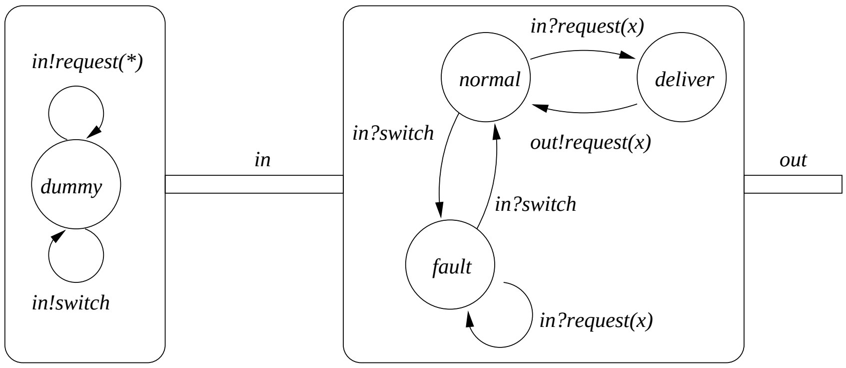


Fig. 4. The client/server example.

Theorem 3

(i) $\approx^{reset} \subseteq \approx^{live}$

(ii) if the queues are reset-independent then $\approx^{reset} = \approx^{live}$

Finally, note that SDL systems satisfy the independence hypothesis, as they are composed from parallel processes, each one having its own unique input queue. Thus, performing signal input in a process does not interfere with other possible inputs in other processes.

6 Example

We illustrate the potential reduction obtained using the live equivalence on a small example. We consider two simple processes communicating through a queue as shown in figure 4. The first process only send arbitrarily *request* and *switch* messages. It is described by Z_{dummy} and the following equation:

$$Z_{dummy} \lhd_{def} (in!switch + in!request(*)).Z_{dummy}$$

The second process is more interesting. It has two functioning modes, a *normal* one and a *fault* one. Basically, it inputs *requests* from the *in* queue and delivers them, when normal, and loses them otherwise. The mode changes from *normal* to *fault* (and backward) when the input is a *switch* signal. Formally, we consider this process described by Z_{normal} and the following equations:

$$\begin{aligned} Z_{normal}(y) &\lhd_{def} in?request(x).Z_{deliver}(x) + in?switch.Z_{fault}(y) \\ Z_{deliver}(x) &\lhd_{def} out!request(x).Z_{normal}(x) \\ Z_{fault}(y) &\lhd_{def} in?request(x).Z_{fault}(x) + in?switch.Z_{normal}(y) \end{aligned}$$

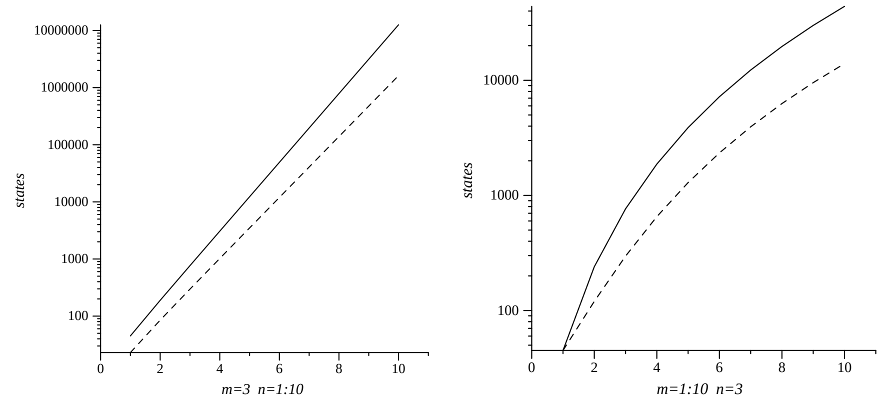


Fig. 5. Live equivalence reduction

We can easily check that the variable x is live only at the *deliver* state. Thus, the value of x must not be used to distinguish between states when somewhere else than the state *deliver*. Furthermore, it can be seen that when at the $fault$ state, the parameters of the incoming $requests$ are not *live* too. $request$ signals are only consumed, without using the carried values. It can be seen also that signals of the *out* queue are never consumed so we don't distinguish states which differs on the *out* content. Such cases are all captured by the live equivalence as defined in section 4.

Figure 5 shows the reduction obtained for this example with respect to two parameters: m, the size of the domain of x and n, the maximal allowed size of queue *in*. The continuous lines give the real number of states and the dotted lines give the number of states obtained after considering live equivalence reduction and please notice that a logarithmic scale was used for the *number-of-states* axis. Also, to better illustrate the reduction obtained, when $m = 6$ and $n = 5$ the initial number of states is 352944 and it is reduced at 89824, so with a factor up to 75%. Furthermore, this reduction depends on both variables and queue contents equivalences. For example, when each one is applied in isolation the number of states generated is respectively 156864 when considering only the variable context equivalence, and 164304 when consider only the queue context equivalence.

7 Conclusion and future work

In the context of model-based validation, the main interest of static analysis is to reduce the state space, which is crucial in practice to deal with complex specifications. Model checking and automatic test generation for conformance testing are based on the same algorithmic techniques: given a specification and a property, they compute, in the first case the validity of the property and, in the second case, the test case with respect to either the specification and the property.

The essence of this work is to show that using the live variable analysis ones can define a *live bisimulation* equivalence stronger than the strong bisimulation. The live bisimulation allows to simplify the models exploiting unused dead values stored both in process variables and in communication queues. This idea was already experimented for the industrial case studies SSCOP [15] where we have obtained impressive reductions of the state space up to 200 times and MASCARA [16] where the reductions not only have been important (by 96%), but the live variables analysis was also essential in order to generate the model.

We are currently developing IF [17], a toolbox for a high level representation of asynchronous systems, especially for real-time specification languages such as SDL and UML. In this toolbox, static analysis is intensively used in the earlier stages of validation in order to reduce the memory costs of the next ones. This lead us to consider two classes of analysis: *property independent* analysis: (such as live variable analysis or constant propagation) without regarding any particular property, the analysis is implemented by a module which takes as input and output a intermediate program, *property dependent* analysis: the analysis takes into account some information extracted from the property or from the environment and propagate them over the static control structure of the program (such as *uncontrollable variables* abstraction [18] or the *slicing* [19] in the context of test case generation [20]). We envisage to experiment more sophisticated analysis, such as general constraint propagation. We also want to exploit a connection with the tool InVeSt [21], which computes abstractions and invariants on communicating processes.

References

[1] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Readings, MA, 1986.

[2] S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann Publishers, 1997.

[3] R. Milner, A Calculus of Communication Systems, Vol. 92 of LNCS, Springer, 1980.

[4] D. Park, Concurrency and Automata on Infinite Sequences, Theoretical Computer Science 104 (1981) 167–183.

[5] J. Fernandez, C. Jard, T. Jéron, L. Mounier, "On the Fly" Verification of Finite Transition Systems, Formal Methods in System Design .

[6] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State Explosion Problem, Vol. 1032 of LNCS, Springer, 1996, iSBN 3-540-60761-7.

[7] K. McMillan, Symbolic Model Checking: an Approach to the State Explosion Problem, Kluwer Academic Publisher, 1993.

[8] J. Fernandez, A. Kerbrat, L. Mounier, Symbolic Equivalence Checking, in: C. Courcoubetis (Ed.), Proceedings of CAV'93 (Heraklion, Greece), Vol. 697 of LNCS, Springer, 1993, pp. 85–96.

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

[10] J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu, CADP: A Protocol Validation and Verification Toolbox, in: R. Alur, T. Henzinger (Eds.), Proceedings of CAV'96 (New Brunswick, USA), Vol. 1102 of LNCS, Springer, 1996, pp. 437–440.

[11] C. Daws, S. Yovine, Reducing the Number of Clock Variables of Timed Automata, in: Proceedings of RTSS'96 (Washington, DC, USA), IEEE Computer Society Press, 1996.

[12] R. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, Static Partial Order Reduction, in: B. Steffen (Ed.), Proceedings of TACAS'98 (Lisbon, Portugal), Vol. 1384 of LNCS, Springer, 1998, pp. 345–357.

[13] B. Boigelot, P. Godefroid, Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs, in: R. Alur, T. Henzinger (Eds.), Proceedings of CAV'96 (New Brunswick, USA), Vol. 1102 of LNCS, Springer, 1996, pp. 1–12.

[14] P. Abdulla, A. Annichini, A. Bouajjani, Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol, in: R. Cleaveland (Ed.), Proceedings of TACAS'99 (Amsterdam, The Netherlands), Vol. 1579 of LNCS, Springer, 1999, pp. 208–222.

[15] M. Bozga, J. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, L. Mounier, Verification and Test Generation for the SSCOP Protocol, Journal of Science of Computer Programming, Special Isssue on Formal Methods in Industry 36 (1) (2000) 27–52.

[16] S. Graf, G. Jia, Verification Experiments on the Mascara Protocol, in: Proceedings of the SPIN'01 Workshop, 2001, to appear.

[17] M. Bozga, J. Fernandez, L. Ghirvu, S. Graf, J. Krimm, L. Mounier, IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems, in: J. Wing, J. Woodcock, J. Davies (Eds.), Proceedings of FM'99 (Toulouse, France), Vol. 1708 of LNCS, Springer, 1999, pp. 307–327.

[18] C. Colby, P. Godefroid, L. Jagadeesan, Automatically Closing Open Reactive Systems, in: Proceedings of ACM SIGPLAN on Programming Language Design and Implementation (Montreal, Canada), 1998, pp. 345–357.

[19] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering SE-10 (4).

[20] M. Bozga, J.-C. Fernandez, L. Ghirvu, Using Static Analysis To Improve Automatic Test Generation, in: S. Graf, M. Schwartzbach (Eds.), Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1785 of Lecture Notes in Computer Science, Springer, Berlin, DE, 2000, pp. 235–250.

[21] S. Bensalem, Y. Lakhnech, S. Owre, Computing Abstractions of Infinite State Systems Compositionally and Automatically, in: A. Hu, M. Vardi (Eds.), Proceedings of CAV'98 (Vancouver, Canada), Vol. 1427 of LNCS, Springer, 1998, pp. 319–331.