State Space Reduction based on Live Variables Analysis

Marius Bozga[1], Jean-Claude Fernandez[2], and Lucian Ghirvu[1]*

[1] VERIMAG***, Centre Equation, 2 avenue de Vignate, F-38610 Gières
Marius.Bozga@imag.fr, Lucian.Ghirvu@imag.fr
[2] LSR/IMAG, BP 82, F-38402 Saint Martin d'Hères Cedex
Jean-Claude.Fernandez@imag.fr

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.
The goal of this paper is to point out that an equivalence on our model derived from the information on *live variables* is stronger than the strong bisimulation. This equivalence, further called *live bisimulation*, exploits the unused dead values stored either in variables or in queue contents and allow to simplify the state space 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.

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

1 Introduction

Formal Description Techniques such as LOTOS [16] or SDL [17] are now at the base of a technology for the specification and the validation of telecommunication systems. This is due not only to the fact that these formalisms are promoted by ITU and other international standardization bodies but also to the availability of mature commercial tools, mainly for editing, code generation and testing.

Alternatively, we have been developing for more than ten years a set of tools dedicated to the design and validation of critical systems and based on the model checking paradigm [22, 8]. One of them is the model checker ALDEBARAN [7] maintained and distributed in collaboration with the VASY team of INRIA Rhône-Alpes as part of the CADP toolset [11]. Another one is the test sequence generator TGV [13], built upon CADP and jointly developed with the PAMPA project of IRISA.

* This work was partially supported by Région Rhône Alpes
*** VERIMAG is a joint laboratory of CNRS, UJF and INPG Grenoble

The central problem arising in the context of model based validation and implicitly for the above mentioned tools is the well known *state explosion problem*. To deal with it, we begin more recently to investigate alternative program representations and more important, ways to adapt techniques issued from other *advanced* domains such as compiler design and optimization in the context of model checking. In this respect, we developed IF [5] which is an intermediate program representation based on *asynchronously communicating timed automata*. IF was designed on one hand to be able to represent significant subsets of SDL and LOTOS and on the other hand to support the application of *static analysis techniques* used in compiler optimization [2, 21]. In particular, a translation from SDL into IF is already implemented using the SDL/API interface provided by the industrial tool *Object*GEODE to its SDL compiler.

In general, model checkers and in particular ALDEBARAN and TGV are based on the central notion of *bisimulation* [20]. In fact, either in the verification process or in the test generation process there is usually a step of minimization modulo *strong bisimulation*. This lead us to consider static analysis techniques for IF programs in the context of bisimulation equivalences.

The main goal of this paper is to point out that an equivalence on our model derived from the information on *live variables* is stronger than the strong bisimulation. This equivalence, further called *live bisimulation*, exploits the unused dead values stored either in variables or in queue contents and allow to simplify the state space 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.

The idea of using static analysis to improve model checking was already being investigated in different particular contexts. For instance, in [10] was proposed a method to reduce the number of clocks in timed automata using live clocks and clocks equality analysis. In [18] was given a method which combines partial order reductions and static analysis of independent actions for SDL programs. An important work was done to find efficient representations of possible infinite queue contents and to exploit the static control structure when performing reachability analysis [4, 1]. However, at the best of our knowledge we are the first to make use of live variables to simplify the state space, including queue contents, of asynchronous systems with queue-based communication.

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. Finally, in section 6 we discuss the general utility of introduced equivalences in the context of model-checking. Some practical results obtained on a small example are given in section 7.

2 The model

2.1 Syntax

We consider systems consisting of asynchronous parallel composition of a number of *processes* that communicate through *parameterized 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 a semantics for terms we assume also the existence of a set of declarations:

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

for each name Z which occurs in terms. Note that in our algebra such declaration binds the occurrences of the variable y inside p to Z, therefore y is considered 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.

2.2 Semantics

We give the semantics of systems in terms of *labeled 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. queues 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 labeled by actions, as follows:

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

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

Note that, in this paper we restrict our attention to systems where the control graph is finite, that is if it contains 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 associates to each queue c a sequence $(s_1, v_1), ..., (s_k, v_k)$ of *messages*, that is 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 *undefined* message σ. 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 labeled 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 labeled with τ, when derived from assignments or signal inputs, either *visible* and labeled with $c!s(v)$ when derived from signal outputs:

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

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

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

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 $Use(\alpha)$ and $Def(\alpha)$ are defined as follows:

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

We consider now the set of *live* variables for some term p. Intuitively it is the smallest set of variables which might be used before they are redefined when interpreting the term. Formally, the sets $Live(p)$ are defined as the least fixpoint solution of the following equation system over sets of variables:

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

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

Lemma 1.

1. $Live(p) = \bigcup_{p \xrightarrow{\alpha} p'} Use(\alpha) \cup Live(p') \setminus Def(\alpha)$.
2. $p \xrightarrow{\alpha} p' \ \Rightarrow \ Use(\alpha) \subseteq Live(p)$, $Live(p') \setminus Def(\alpha) \subseteq Live(p)$.

Proof. *1.* Structural induction over term structure. *2.* Immediate from *1.*

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:

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

We consider similar equivalence relations defined for queue contexts. Intuitively two queue contexts are equivalent if and only if they enable the same sequences of transitions and if for each enabled 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:

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

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.

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, that is, the live equivalence is also a bisimulation relation over the global states.

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

Proof. By definition, $\approx^{live}$ is a bisimulation if and only if for all pairs $(\rho_1, \delta_1, p) \approx^{live} (\rho_2, \delta_2, p)$ we have:

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

$\forall(\rho_2,\delta_2,p)\xrightarrow{a}(\rho_2',\delta_2',p') \Rightarrow \exists(\rho_1,\delta_1,p)\xrightarrow{a}(\rho_1',\delta_1',p') \wedge (\rho_1',\delta_1',p) \approx^{live} (\rho_2',\delta_2',p)$
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. $\alpha = [\ b\ ]\ x := e$
   $\forall i = 1,2\ \ \rho_i(e) = v_i,\ \rho_i(b) = true\ \Rightarrow$
   $(\rho_i,\delta_i,p)\xrightarrow{\tau}(\rho_i',\delta_i,p'),\ \rho_i' = \rho_i[v_i/x]$
   $\rho_1 \sim_p^{live} \rho_2\ \Rightarrow\ \rho_1(b) = \rho_2(b),\ \rho_1(e) = \rho_2(e),\ \rho_1[v_1/x] \sim_{p'}^{live} \rho_2[v_2/x]$
   $\delta_1 \approx_p^{live} \delta_2\ \Rightarrow\ \delta_1 \approx_{p'}^{live} \delta_2$

2. $\alpha = [\ b\ ]\ c!s(e)$
   $\forall i = 1,2\ \ \delta_i(c) = w_i,\ \rho_i(e) = v_i,\ \rho_i(b) = true\ \Rightarrow$
   $(\rho_i,\delta_i,p)\xrightarrow{c!s(v_i)}(\rho_i,\delta_i',p'),\ \delta_i' = \delta_i[w_i.s(v_i)/c]$
   $\rho_1 \sim_p^{live} \rho_2\ \Rightarrow\ \rho_1(b) = \rho_2(b),\ \rho_1(e) = \rho_2(e),\ \rho_1 \sim_{p'}^{live} rho_2$
   $\delta_1 \approx_p^{live} \delta_2\ \Rightarrow\ \delta_1 \approx_{p'}^{live} \delta_2\ \Rightarrow\ \delta_1[w_1.s(v_1)/c] \approx_{p'}^{live} \delta_2[w_2.s(v_2)/c]$
   Note that the second implication comes from the more general property of queue content equivalence to be *closed* under output operations. That is, 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.

3. $\alpha = [\ b\ ]\ c?s(x)$
   $\forall i = 1,2\ \ \delta_i(c) = s(v_i).w_i,\ \rho_i(b) = true\ \Rightarrow$
   $(\rho_i,\delta_i,p)\xrightarrow{\tau}(\rho_i',\delta_i',p'),\ \rho_i' = \rho_i[v_i/x],\ \delta_i' = \delta_i[w_i/c]$
   $\delta_1 \approx_p^{live} \delta_2\ \Rightarrow\ v_1 = v_2$ if $x \in Live(p'),\ \delta_1[w_1/c] \approx_{p'}^{live} \delta_2[w_2/c]$
   $\rho_1 \sim_p^{live} \rho_2\ \Rightarrow\ \rho_1(b) = \rho_2(b),\ \rho_1[v_1/x] \sim_{p'}^{live} \rho_2[v_2/x]$

An immediate consequence of the previous theorem is the following one, stating that the live equivalence is stronger than the strong bisimulation.

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 reseted. Formally, we have:

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

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

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

Lemma 2. $\forall p \;\; \sim_p^{reset} = \sim_p^{live}$.

In order to define a similar $Reset^q$ function for queue contexts we start by introducing some auxiliary notations.

Let consider some fixed queue c. We define the relation $\xrightarrow{\tau_{c?}}$ over control states to include all the control transitions, except the ones labeled with inputs from the queue c. We note with $\overset{\tau^*_{c?}}{\Longrightarrow}$ its transitive and reflexive closure and with $Post_{\overset{\tau^*_{c?}}{\Longrightarrow}}$ the post image function defined over sets of control states, formally:

$$p \xrightarrow{\tau_{c?}} p' \quad \Leftrightarrow \quad \exists \alpha \neq [\ b\ ]\ c?s(x) \;\; p \xrightarrow{\alpha} p'$$

$$Post_{\overset{\tau^*_{c?}}{\Longrightarrow}}(Q) = \{\ p' \mid \exists p \in Q \;\; p \overset{\tau^*_{c?}}{\Longrightarrow} p'\ \}$$

Let consider $\alpha = [\ b\ ]\ c?s(x)$ to be some input action of signal s from queue c. We note $p \overset{\alpha}{\Longrightarrow} p'$ if it is possible to reach p' from p by a sequence ending with α and which does not contain any other input from the queue c. We define also the post image function $Post_{\overset{c?s}{\Longrightarrow}}$ over sets of control states, that is, it gives all the states which can be reached after consuming an s from the queue c. Formally, we have:

$$p \overset{[b]\ c?s(x)}{\Longrightarrow} p' \quad \Leftrightarrow \quad \exists p'' \;\; p \overset{\tau^*_{c?}}{\Longrightarrow} p'' \;\wedge\; p'' \xrightarrow{[b]\ c?s(x)} p'$$

$$Post_{\overset{c?s}{\Longrightarrow}}(Q) = \{\ p' \mid \exists p \in Q \;\; p \overset{[b]\ c?s(x)}{\Longrightarrow} p'\}$$

We define now the local *reset* function for the queue c given an initial non-empty set of control states Q. That is, given the content w of the queue c, this function basically rewrite it and *forgot* the signal parameters which statically are detected to be unused, when execution starts somewhere in Q. With σ denoting the *undefined* message, the function is recursively defined as follows:

$$reset(Q,c,\epsilon) = \begin{cases} \sigma & \text{if } \forall s \;\; Post_{\overset{c?s}{\Longrightarrow}}(Q) = \emptyset \\ \\ \epsilon & \text{otherwise} \end{cases}$$

$$reset(Q,c,s(v).w) = \begin{cases} \sigma & \text{if } \; Post_{\overset{c?s}{\Longrightarrow}}(Q) = \emptyset \\ \\ s(v).reset(Post_{\overset{c?s}{\Longrightarrow}}(Q),c,w) & \\ \quad \text{if } \exists p \in Q,\ p \overset{[b]\ c?s(x)}{\Longrightarrow} p' \;\wedge\; x \in Live(p') & \\ \\ s(\bot).reset(Post_{\overset{c?s}{\Longrightarrow}}(Q),c,w) & \\ \quad \text{if } \forall p \in Q,\ p \overset{[b]\ c?s(x)}{\Longrightarrow} p' \;\Rightarrow\; x \notin Live(p') & \end{cases}$$

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

Lemma 3. *For any Q, c, w, w_1, w_2 holds*

1. $reset(Q, c, w) = reset(Post_{\stackrel{\tau^*_{c?}}{\Longrightarrow}}(Q), c, w)$.
2. $reset(Q, c, w_1) = reset(Q, c, w_2) \;\Rightarrow$
   $\forall Q' \subseteq Q \quad reset(Q', c, w_1) = reset(Q', c, w_2)$.

Proof. *1.* The proof is immediate from the fact that for any set of control states Q, channel c and signal s we have:

$$Q \subseteq Post_{\stackrel{\tau^*_{c?}}{\Longrightarrow}}(Q) \qquad Post_{\stackrel{c?s}{\Longrightarrow}}(Q) = Post_{\stackrel{c?s}{\Longrightarrow}}(Post_{\stackrel{\tau^*_{c?}}{\Longrightarrow}}(Q))$$

2. 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, that is, for a given control state p we consider:

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

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

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. This can be formalized as follows. For α an input action, we note $p \stackrel{\alpha}{\Longrightarrow}_* p'$ if p' is reachable from p by a sequence ending with α and which does not contain other inputs. We define now queue c to be *reset-independent* from queue c' if and only if :

$$\forall p, q, q' \;\; p \stackrel{[b']c'?s'(x')}{\Longrightarrow}_* q, \;\; q \stackrel{[b]c?s(x)}{\Longrightarrow}_* q' \;\Rightarrow$$
$$\exists p' \;\; p \stackrel{[b]c?s(x)}{\Longrightarrow}_* p', \;\; p' \stackrel{[b']c'?s'(x')}{\Longrightarrow}_* q', \;\; x \in Live(p') \Leftrightarrow x \in Live(q')$$

Lemma 4.

1. $\forall p \;\; \approx_p^{reset} \subseteq \approx_p^{live}$.
2. *if the queues are reset-independent then*
   $\forall p \;\; \approx_p^{reset} = \approx_p^{live}$.

Proof. *1.* The proof consist to check 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 the previous lemma. Then because the live equivalences are the greatest fixpoint is clear that $\approx_p^{reset} \subseteq \approx_p^{live}$ for all control states p.

2. Let $\delta_1 \approx_p^{live} \delta_2$ and the queue c fixed. We will prove inductively that $reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c))$. We distinguish the following cases:

1. $\delta_1(c) = \epsilon,\ \delta_2(c) = \epsilon$
   $\Rightarrow reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c))$
2. $\delta_1(c) = \epsilon,\ \delta_2(c) = s_2(v_2).w_2$
   (a) $\forall s \quad Post_{\stackrel{\Longrightarrow}{c?s}}(\{p\}) = \emptyset$
   $\Rightarrow reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c)) = \sigma$
   (b) $\exists s \quad Post_{\stackrel{\Longrightarrow}{c?s}}(\{p\}) \neq \emptyset$
   $\Rightarrow \exists p = p_0 \stackrel{\tau_{c?s}}{\longrightarrow} p_1 \stackrel{\tau_{c?s}}{\longrightarrow} ... \stackrel{\tau_{c?s}}{\longrightarrow} p_n \stackrel{[b]\ c?s(x)}{\longrightarrow} p'$
   and because the inputs are independent we can choose this sequence such that it doesn't contain inputs from other queues
   $\Rightarrow$ we obtain a contradiction with the definition of the live equivalence hypothesis, that is $\delta_1 \not\approx_p^{live} \delta_2$
3. $\delta_1(c) = s_1(v_1).w_1,\ \delta_2(c) = s_2(v_2).w_2$
   (a) $Post_{\stackrel{\Longrightarrow}{c?s_1}}(\{p\}) = \emptyset,\ Post_{\stackrel{\Longrightarrow}{c?s_2}}(\{p\}) = \emptyset$
   $\Rightarrow reset(\{p\}, c, \delta_1(c)) = reset(\{p\}, c, \delta_2(c)) = \sigma$
   (b) $Post_{\stackrel{\Longrightarrow}{c?s_1}}(\{p\}) \neq \emptyset,\ Post_{\stackrel{\Longrightarrow}{c?s_2}}(\{p\}) = \emptyset$
   $\Rightarrow$ contradiction with the live equivalence, as before.
   (c) $Post_{\stackrel{\Longrightarrow}{c?s_1}}(\{p\}) \neq \emptyset,\ Post_{\stackrel{\Longrightarrow}{c?s_2}}(\{p\}) \neq \emptyset$
      i. $s_1 \neq s_2$
      $\Rightarrow$ contradiction with the live equivalence, as before.
      ii. $s_1 = s_2 = s$ and $v_1 \neq v_2$ and $\exists p^{[b]} \stackrel{c?s(x)}{\Longrightarrow} p' \quad x \in Live(p')$
      $\Rightarrow$ contradiction with the live equivalence, as before.
      iii. $s_1 = s_2 = s$ and $v_1 = v_2 = v$ or $\forall p^{[b]} \stackrel{c?s(x)}{\Longrightarrow} p' \quad x \notin Live(p')$
      $\Rightarrow reset(\{p\}, c, s_1(v_1).w_1) = s(v).reset(Post_{\stackrel{\Longrightarrow}{c?s}}(\{p\}), c, w_1)$
      $reset(\{p\}, c, s_2(v_2).w_2) = s(v).reset(Post_{\stackrel{\Longrightarrow}{c?s}}(\{p\}), c, w_2)$
      and from the live equivalence hypothesis we have also that
      $\forall p' \in Post_{\stackrel{\Longrightarrow}{c?s}}(\{p\}) \quad \delta_1[w_1/c] \approx_{p'}^{live} \delta_2[w_2/c]$
      so we can inductively infer that either contradiction or
      $reset(\{p\}, c, s_1(v_1).w_1) = reset(\{p\}, c, s_2(v_2).w_2)$

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

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

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

Theorem 3.

1. $\approx^{reset} \subseteq \approx^{live}$
2. *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 a signal input in a process does not interfere with other possible inputs in other processes.

6 Live analysis and model checking

The reductions based on live or reset equivalence can be obtained with almost *no cost* and are fully *orthogonal* to other techniques applied in the context of model-checking validation to deal with the state-explosion problem.

The weak cost of the live equivalence reduction is due to the static analysis approach: live variables need to be computed once in the beginning with an algorithm operating on the control graph and then are used by primitives such as *Reset* which operate on states content 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 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 equivalence reduction can be combined with techniques ranging from the simplest model generation, with standard on the fly verification methods and even with more sophisticated partial order or symbolic techniques.

Enumerative simulation is at the basis of most of the verification algorithms. It is always possible to directly generate the quotient graph with respect to live or reset equivalences, thus preserving all the observable behaviors, without constructing the initial one. This can be easily done for instance by considering every time a new state is generated its canonical form given by *Reset*. Reset equivalent states will be automatically identified as equals and explored once by the generation algorithm.

On the fly verification techniques such as [12] rely on enumerative depth first traversals of the model state space 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 graph, if it is not explicitly dependent on state variables or queue contents but only on the observable actions of the system (e.g, outputs). Note that, if such dependencies exist, they can be normally removed by introducing auxiliary observable actions and by modifying the property accordingly.

Partial order techniques [15] reduce the state space by avoiding to explore interleaving of independent actions. Clearly, the live equivalence reduction can be further considered when new states are encountered, as before.

In the symbolic model checking context [19] 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 state 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 equivalence reduction can also be applied at each step of the minimal model generation algorithm [14] 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 might be equivalent w.r.t. the live equivalence.

7 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 1. The first process only send arbitrarily *request* and *switch* messages. It is described by Z_{dummy} and the following equation:

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

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 looses them otherwise. The mode changes from *normal* to $fault$ (and backward) when input a *switch* signal. Formally, we consider this process described by Z_{normal} and the following equations:

$$Z_{normal} \triangleleft_{def} in?request(x).Z_{deliver} + in?switch.Z_{fault}$$

$$Z_{deliver} \triangleleft_{def} out!request(x).Z_{normal}$$

$$Z_{fault} \triangleleft_{def} in?request(x).Z_{fault} + in?switch.Z_{normal}$$

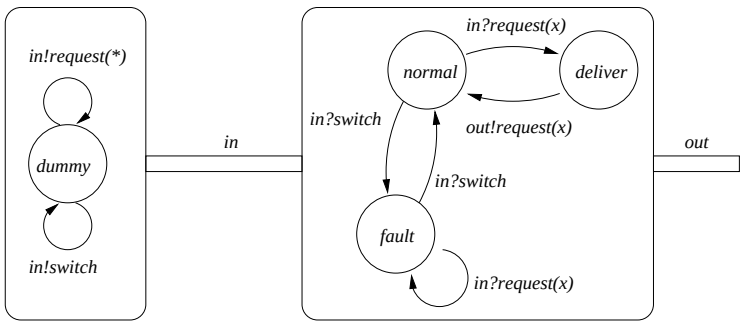

Fig. 1. The example

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 that *deliver*. Furthermore, it can be seen that when at the $fault$ state, the

parameters of the incoming *requests* are not *live* too. That is, *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 2 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 an logarithmic scale was used for the *number-of-states* axis. Also, to better illustrate the reduction obtained, note that for instance 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%.

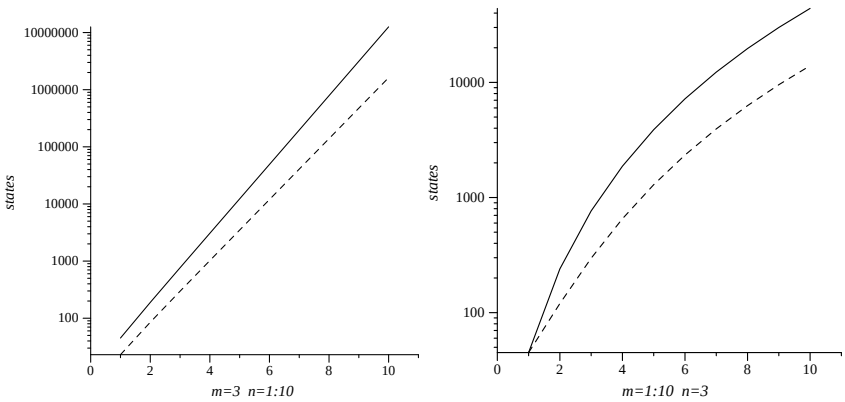

Fig. 2. Live equivalence reduction

8 Conclusion and future work

The essence of this work is to show that the live variable analysis define an equivalence stronger than bisimulation equivalence. This allow to simplify the state space exploiting the information on unused dead values stored either in the variables or in the queue contents. This idea was already experimented to the industrial case study SSCOP [6] where we have obtained impressive reductions of the state space up to 200 times.

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. We are currently developing IF [5], a toolbox for a high level representation of programs, especially for telecommunication specification languages. 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 [9]).

We envisage to experiment more sophisticated analysis, such as constraints propagation in the context of symbolic test generation. We also want to exploit a connection with the tool INVEST [3], which computes abstractions and invariants on a set of guarded commands.

References

1. P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In *Proceedings of CAV'98, Vancouver, Canada*, volume 1427 of *LNCS*, 1998.
2. A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Readings, MA, 1986.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, 1998.
4. B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. In *Proceedings of CAV'96, New Brunswick, USA*, volume 1102 of *LNCS*, 1996.
5. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. In *Proceedings of SDL-FORUM'99, Montreal, Canada*, 1999.
6. M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and Test Generation for the SSCOP Protocol. *SCP*, 1998. to appear.
7. M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the Aldebaran Toolset. *Springer International Journal on Software Tools for Technology Transfer*, 1(1+2):166–183, December 1997.
8. E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of 10th ACM Symposium on Programming Languages*, 1983.
9. C. Colby, P. Godefroid, and L.J. Jagadeesan. Automatically Closing Open Reactive Systems. In *Proceedings of ACM SIGPLAN on PLDI*, June 1998.

10. C. Daws and S. Yovine. Reducing the Number of Clock Variables of Timed Automata. In *Proceedings of RTSS'96*, 1996.
11. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *Proceedings of CAV'96, New Brunswick, USA*, volume 1102 of *LNCS*, 1996.
12. J.-C. Fernandez, C. Jard, T. Jéron, and L. Mounier. "On the Fly" Verification of Finite Transition Systems. *Formal Methods in System Design*, 1992.
13. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *SCP*, 29, 1997.
14. J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic Equivalence Checking. In *Proceedings of CAV'93, Heraklion, Greece*, volume 697 of *LNCS*, 1993.
15. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State Explosion Problem. volume 1032 of *LNCS*, 1996.
16. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, 1988.
17. ITU-T. *Recommendation Z-100. Specification and Description Language (SDL).* 1994.
18. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction. In *Proceedings of TACAS'98, Lisbon, Portugal*, volume 1384 of *LNCS*, 1998.
19. K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem.* Kluwer Academic Publisher, 1993.
20. R. Milner. A Calculus of Communication Systems. In *LNCS*, number 92. 1980.
21. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.
22. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Programs in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, 1982.