Generating Exact Approximations to Model Check Concurrent Systems

Mustapha Bourahla
Computer Science Department, University of Biskra, Algeria

Abstract: *In this paper, we present a method to generate abstractions for model checking concurrent systems. A program using a defined syntax and semantics, first describes the concurrent system that can be infinite. This program is abstracted using the framework of abstract interpretation where an abstract function will be given. This abstract program is demonstrated to be an accurate approximation of the original program that may contain spurious behaviours. These spurious behaviours will be identified and removed using a new defined abstraction framework based on the restrictions. The new produced abstract program is an exact approximation of the original program.*

Keywords: *Model checking, abstractions, concurrent systems.*

Received February 23, 2007; accepted June 6, 2007

1. Introduction

In the literature, there are many techniques proposed to analyse the concurrent systems [1, 2, 3, 4, 5, 8, 22]. Model Checking [5] represents one of the most useful results of almost twenty years of research in formal methods to increase the quality of software and hardware systems. A model checker works with a high level description of a system (model), and it can automatically inspect the reachable states of the system to check if a given property (expressed with some variant of temporal logic) is satisfied.

In the context of model checking, abstract interpretation [6] is used as way of dealing with the so-called state explosion problem, which occurs when realistic systems are analysed. Abstract model checking involves two activities. On the one hand, in order to reduce the state space of the original model M, we apply abstract interpretation to construct an abstract model $\widetilde{M}$ approximating M. On the other, we abstract the original temporal properties.

The final objective of the abstraction process is the "strong preservation", that is, the preservation of both the truth and the falsehood, of the temporal properties j between M and $\widetilde{M}$, in other words,

$$\widetilde{M} \models \alpha(\varphi) \Leftrightarrow M \models \varphi \quad (1)$$

where a is the property abstract function. However, the strong preservation of the temporal properties is only possible if M and $\widetilde{M}$ are bi-similar, which entails a considerable constraint when the objective is to decrease the state space.

We propose in this paper a method to construct accurate approximations of infinite concurrent systems described by programs written with a defined syntax. By accurate approximations, we mean abstract models having only the states and transitions that can be mapped to concrete states and concrete transitions respectively. The accurate approximations can have spurious behaviours that are executions with no corresponding executions in the concrete system. It is sufficient to remove these spurious behaviours to get an abstract system strongly preserving the temporal properties.

Our method is different from the other techniques [9, 19, 20, 21]. For removing the spurious behaviours in the abstract model of the concurrent system, the techniques proposed in the literature refine gradually the abstract model in many steps. These methods use the generated counter-examples to refine the abstract model. Thus, the abstract model is augmented by a behaviour depending on the generated counter-example. The new abstract model will be checked and if the property is not satisfied, the process of refinement will continue using the new generated counter-example until the property is satisfied or no refinement is possible. Our method identifies all the spurious behaviours, and as a result, it will refine this abstract model in a one step.

It is often expensive or impossible to construct $\widetilde{M}$ directly because we must have a representation of M to do the abstraction. We may not be able to obtain such a representation if M is infinite or simply too large for our system to handle. To circumvent this problem, we use a method that is based on the fact that we usually have an implicit representation of M as a program in a concurrent language. We will show how to compute an approximation to M directly from the program text. This approximation is generally accurate

enough to allow us to verify interesting properties of the program.

A program in a concurrent language can be transformed into relational expressions I and R that can be evaluated to obtain the initial states I and the transition relation R of the concurrent system M represented by the program. These relational expressions are simply formulas in first-order predicate logic that will built up from a set of primitive relations for the basic operators and constants in the concurrent language. We will manipulate I and R to obtain the approximation to M.

There will typically be types associated with the variables and relation arguments in the relational expressions that we write. A concurrent system is now represented by formulas I and R. Similar formulas $\widetilde{I}$ and $\widetilde{R}$ can be obtained representing $\widetilde{M}$.

The rest of the paper is organized as follows. Section 2 is devoted to the definition of the syntax and semantics of the language used to write concurrent systems. This language is based on the first-order predicate logic. An example is given at the end of this section to illustrate these definitions. In section 3, we present the abstraction algorithm, which is based on the abstract interpretation framework. Section 4 presents the abstract model checking. In section 5, we present the method of removing the spurious behaviours after their definition to generate exact approximations. At the end a conclusion is given.

2. Describing Concurrent Systems

A concurrent program will describe a concurrent system, which is the parallel composition of many processes, which is an infinite state-transition program. In this program, we specify the composed transition relation, the initial state and the invariant. It is possible to derive this infinite state-transition program from a high level language describing concurrent systems.

This program is composed of a finite set of variables $V=\{v_1,v_2,\ldots,v_n\}$. If each variable v_i ranges over a (non-empty) set Σ_i of possible values, then the set of all possible program states is $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n$, which we denote by Σ. We present the possible behaviours of the program with a set of transitions between states.

Syntax: An infinite state-transition program $P=(R, I,)$ consists of

- R is the predicate of the system transition relation. This predicate is a disjunctive formula of a set of conjunctive sub-formulas. Each conjunctive sub-formula representing a transition in the concurrent system which is of the form $\gamma \wedge A$, for a state predicate γ (the guard of the transition) and a set $A=\{v:=a_v\}$ of simultaneous assignments such that for all variables $v \in V$, the expression a_v is any acceptable expression.
- I is the predicate of initial state.
- g, is the program invariant, a state predicate. We require , to verify the condition, that is, for all states $s \in S$, $s \models$, .

A guarded transition defines a partial function from S to S. Let $g=\gamma \wedge A$ be the guarded transition and let $s \in S$ be a state. The guarded transition g is enabled in the state s if $s \models \gamma$. Any guarded transition that is enabled in s may be executed in s. The execution of g, in particular, leads to the state $s[A]$, where $s[A](v)=s(a_v)$ for all variables $v \in V$. The infinite state-transition program $P=(R, I,,)$ defines the transition relation R_P such that $(s,s') \in R_P$ if

- For some guarded transition $g=\gamma \wedge A \in R$, $s \models \gamma$ and $s'=s[A]$,
- $s \models$, and $s'=s$, .

Execution of a concurrent program may be defined by means of a transition system M. If P_M is the set of paths of the transition relation R_P, the invariant , of an infinite state-transition program P defines precisely the set S_{R_P} of states that occur on some M-path. It follows that a non-deadlock infinite transition program can be executed by starting with the initial state that satisfies the invariant and then, repeatedly, choose a guarded transition that is enabled and whose execution does not violate the invariant. The iteration of the next relation defined by a deadlock infinite transition program, on the other hand, may lead to a state from which execution cannot continue.

Example 1: consider a system composed of two processes (the dining mathematicians, example taken from [7]), which use a parallel version of the *Colatz* program for the mutually exclusive access to the critical section where they may eat. The set of system states Σ is $\{think,eat\}^2 \times N$, where N is the set of natural numbers. An element $\langle m_0,m_1,n\rangle \in S$ represents the state of each mathematician, thinking or eating, and the current value of variable n.

The initial state s_0 is $\langle think,think,50\rangle$, and the infinite state-transition program (which is the composition of the two processes) is defined as shown in Figure 1. That is, the parity of n decides which mathematician may eat. $next(s)=s$ means that the next value of state s will be equal s. This expression represents the conjunctive sub-formula composed of the assignments in that transition. The unique trace in P_M is

$$t=\langle think,think,٥٠\rangle \langle think,eat,٥٠\rangle \langle think,think,٢٥\rangle \langle eat,think,٢٥\rangle \langle think,think,٧٦\rangle \ldots$$

For example, the transition t_1 has the corresponding predicate expression (which is evaluated to be true, because it is taken).

$t_1 = (m_0 = think \wedge m_1 = think \wedge n = 50 \wedge even(50) \wedge next(m_0) = think \wedge next(m_1) = eat \wedge next(n) = 50)$

$$
\begin{aligned}
&P = \{\\
&\quad R = \{\\
&\qquad (s = \langle think, m_1, n\rangle \wedge odd(n) \wedge next(s) = \langle eat, m_1, n\rangle) \vee\\
&\qquad (s = \langle eat, m_1, n\rangle \wedge next(s) = \langle think, m_1, 3 * n + 1\rangle) \vee\\
&\qquad (s = \langle m_0, think, n\rangle \wedge even(n) \wedge next(s) = \langle m_0, eat, n\rangle) \vee\\
&\qquad (s = \langle m_0, eat, n\rangle \wedge next(s) = \langle m_0, think, n/2\rangle)\\
&\quad \}\\
&\quad I = \langle think, think, 50\rangle\\
&\quad , = true\\
&\}
\end{aligned}
$$

Figure 1. The example program.

3. Abstractions

The model checking works only on finite models. However, it is not always possible to construct finite models from the specification programs. Thus, we need to do abstractions. Abstractions [10] will be formed by letting the program variables range over (non empty) sets $\tilde{S}_i$ of abstract values. We will give mappings to specify the correspondence between unabstracted and abstracted values. Formally, we let $h_1, h_2, \ldots, h_n$ be surjection, with $h_i : S_i \to \tilde{S}_i$ for each i. These mappings induce a surjection $h : S \to \tilde{S}$ defined by

$$h((s_1, \ldots, s_n)) = (h_1(s_1), \ldots, h_n(s_n)) \quad (2)$$

Alternatively, the relation between unabstracted and abstracted values can be specified by a set of equivalence relations. In particular, each h_i corresponds to the equivalence relation $\equiv_i \subseteq S_i \times S_i$ defined by

$$d_i \equiv_i d_i' \iff h_i(d_i) = h_i(d_i') \quad (3)$$

The mapping h induces an equivalence relation $\equiv \subseteq S \times S$ in the same manner

$$(d_1, \ldots, d_n) \equiv (d_1', \ldots, d_n') \iff d_1 \equiv_1 d_1' \wedge \ldots \wedge d_n \equiv_n d_n' \quad (4)$$

Definition 1:
Let M be a concurrent system over S and $\tilde{M}$ be a concurrent system over $\tilde{S}$. We say that $\tilde{M}$ approximates M (denoted $M \hat{0}_k \tilde{M}$) when:

- $\exists \sigma (h(\sigma) = \tilde{\sigma} \wedge I(\sigma)) \Rightarrow \tilde{I}(\tilde{\sigma})$.
- $\exists \sigma_1 \exists \sigma_2 (h(\sigma_1) = \tilde{\sigma} \wedge h(\sigma_2) = \tilde{\sigma}_2 \wedge R(\sigma_1, \sigma_2)) \Rightarrow \tilde{R}(\tilde{\sigma}_1, \tilde{\sigma}_2)$

Then $\tilde{M}$ approximates M when initial states and transitions in M have corresponding initial states and transitions in $\tilde{M}$. For exact approximation, we must have a type of converse as well: if $\tilde{s}$ is an initial state of $\tilde{M}$, then all of the states s of M that map to $\tilde{s}$ should be initial as well (and similarly for transitions).

Definition 2:
Let $\tilde{M}$ be a concurrent system over $\tilde{S}$. We say that $\tilde{M}$ exactly approximates M (denoted $(M \approx_h \tilde{M})$ when $M \hat{0}_k \tilde{M}$ and:

- $\tilde{I}(\tilde{\sigma}) \Rightarrow \forall \sigma (h(\sigma) = \tilde{\sigma} \Rightarrow I(\sigma))$
- $\tilde{R}(\tilde{\sigma}_1, \tilde{\sigma}_2) \Rightarrow \forall \sigma_1 \forall \sigma_2 (h(\sigma_1) = \tilde{\sigma}_1 \wedge h(\sigma_2) = \tilde{\sigma}_2 \Rightarrow R(\sigma_1, \sigma_2))$.

Thus, the concrete and abstract models exhibit identical behaviour. Exact approximations generally allow very little simplification, and hence they are not very useful for reducing the complexity of verification.

3.1. Generating Accurate Abstractions

An accurate abstract concurrent system has only initial states and transitions verifying the definition of the approximation. We call this accurate abstract concurrent system $\tilde{M}_a$.

Definition 3:
$\tilde{M}_a$ is the concurrent system over $\tilde{S}$ given by:

- $\tilde{I}_a(\tilde{\sigma}) \Leftrightarrow \exists \sigma (h(\sigma) = \tilde{\sigma} \wedge I(\sigma))$
- $\tilde{R}_a(\tilde{\sigma}_1, \tilde{\sigma}_2) \Leftrightarrow \exists \sigma_1 \exists \sigma_2 (h(\sigma_1) = \tilde{\sigma}_1 \wedge h(\sigma_2) = \tilde{\sigma}_2 \wedge R(\sigma_1, \sigma_2))$

Obviously $M \hat{0}_k \tilde{M}$. Further, for any other concurrent system $\tilde{M}$ over $\tilde{S}$, we see that $M \hat{0}_k \tilde{M}$ if and only if $\tilde{I} \supseteq \tilde{I}_a$ and $\tilde{R} \supseteq \tilde{R}_a$. Thus, $\tilde{M}_a$ is the most accurate approximation to M that is consistent with h.

For simplicity, we assume that all of the variables $V = \{v_1, \ldots, v_n\}$, range over the same domain S. We also use a set $\tilde{V} = \{\tilde{v}_1, \ldots, \tilde{v}_n\}$, of variables ranging over the abstract domain $\tilde{S}$, with $\tilde{v}_i$ representing the abstract value of v_i. We will also assume that there is only one abstraction function h mapping elements of S to elements of $\tilde{S}$. Each transition $t = s \wedge guard \wedge next(s)$, where

$$s = \bigwedge_{i=1}^{n} (v_i = d_i) \text{ and } next(s) = \bigwedge_{i=1}^{n} (next(v_i) = e_i) \quad (5)$$

in the transition relation, which is specified in the concurrent system program, will be accurately approximated as.

$$t' = AccurateApproximation(t) = (\bigcup_{i=1}^{n}(v_i = h(d_i) \mid guard \Rightarrow h(d_i))) \cup (\bigcup_{i=1}^{n}(next(v_i) = h(e_i))) \quad (6)$$

By the same way, we can compute $\widetilde{R}$ and $\widetilde{I}$.

Theorem 1: $\widetilde{R}$ and $\widetilde{I}$ are accurate approximations of *R* and *I* respectively.

Proof: The demonstration of this theorem is trivial, because by construction we are creating $\widetilde{R}$ and $\widetilde{I}$ consisting solely from transitions and initial states that are only mapped to concrete transitions and concrete initial states respectively.

Definition 4: The equivalence relation $\equiv$ is congruence with respect to a relation R over S if

$$\forall x, y \in S: x \equiv y \Rightarrow R(x) \Leftrightarrow R(y) \quad (7)$$

If the mapping function h induces a congruence equivalence relation $\equiv$ with respect to the transition relation, then the generated accurate approximation $\widetilde{M}_a$ is an exact approximation.

Example 2: Consider the abstract concurrent system, where $S = \{think, eat\} \times \{e, o\}$, where e means the number n is even and o means n is odd. If $m_0, m_1 \in \{think, eat\}$ and $n \in N$, we can define the mapping function as

$$h(\langle m_0, m_1, n\rangle) = \langle h_1(m_0), h_1(m_1), h_2(n)\rangle, \text{ where } h_1(m) = m \text{ and } h_2(n) = \begin{cases} e \text{ if } n \text{ is even} \\ o \text{ if } n \text{ is odd} \end{cases} \quad (8)$$

The abstract initial state $\widetilde{I}$ and the abstract transition relation $\widetilde{R}$ generated by the algorithm accurate approximation are presented in Figure 2.

The abstract action of $n := 2 \times n + 1$ is $\varnothing even(n)$. But the abstract action of $n := n/2$ is indeterminate which produces imprecise values. Observe that in this example the imprecision of the action $n := n/2$ is solved by means of a non-deterministic selection between the last four transitions. We should remark also that there are no reachable states in the abstract model from the specified initial state. The abstract trace approximating $t \in P_M$ is

$$t' = h(t) = \langle think, think, e\rangle \langle think, eat, e\rangle \langle think, think, o\rangle \langle eat, think, o\rangle \langle think, think, e\rangle$$

Note that $\widetilde{M}$ contains spurious traces that are caused by the presence of non-deterministic transitions.

$$\widetilde{P} = \{$$
$$\widetilde{R} = \{$$
$$(\widetilde{a} = \langle think, m_1, o\rangle \wedge next(\widetilde{a}) = \langle eat, m_1, o\rangle) \vee$$
$$(\widetilde{a} = \langle eat, m_1, o\rangle \wedge next(\widetilde{a}) = \langle think, m_1, e\rangle) \vee$$
$$(\widetilde{a} = \langle eat, m_1, e\rangle \wedge next(\widetilde{a}) = \langle think, m_1, o\rangle) \vee$$
$$(\widetilde{a} = \langle m_0, think, e\rangle \wedge next(\widetilde{a}) = \langle m0, eat, e\rangle) \vee$$
$$(\widetilde{a} = \langle m0, eat, e\rangle \wedge next(\widetilde{a}) = \langle m_0, think, e\rangle) \vee$$
$$(\widetilde{a} = \langle m_0, eat, e\rangle \wedge next(\widetilde{a}) = \langle m_0, think, o\rangle) \vee$$
$$(\widetilde{a} = \langle m_0, eat, o\rangle \wedge next(\widetilde{a}) = \langle m_0, think, o\rangle) \vee$$
$$(\widetilde{a} = \langle m_0, eat, o\rangle \wedge next(\widetilde{a}) = \langle m_0, think, e\rangle)$$
$$\}$$
$$\widetilde{I} = \langle think, think, e\rangle$$
$$= true$$
$$\}$$

Figure 2. The abstract program.

4. Model Checking

After the description of the abstraction process of concurrent systems, we will present the process of the abstraction of their properties and we will show how the abstract system can preserve these temporal properties.

4.1. Temporal Logic

CTL* (Computation Tree Logic, * stands for universal logic) [11] is a powerful temporal logic that can express both branching time and linear time properties. If $v_i \in V$ is a program variable and $d_i \in S_i$, then $v_i = d_i$ and $v_i \neq d_i$ are atomic state formulas. *true* and *false* are also atomic state formulas. We denote the set of atomic formulas by A.

Syntax and Semantics of CTL*: The grammar given below defines two entities, state formulas (denoted by *sf*) and path formulas (denoted by *pf*). The logic CTL* is formally defined as the set of state formulas obtained by the grammar:

$$\begin{aligned} sf &::= p \in A \mid \neg sf \mid sf \wedge sf \mid \forall pf \mid \exists pf \\ pf &::= p \in A \mid \neg pf \mid pf \wedge pf \mid Fsf \mid Gsf \mid Xsf \mid \\ &\quad sf \cup sf \mid Fpf \mid Gpf \mid Xpf \mid pf \cup pf \end{aligned} \tag{9}$$

This grammar is not given in its most succinct form and there exist equivalence rules to express the same formula with different operators. In practice, by using this equivalence rules, a formula can be written such that the negation appears only at the level of atomic propositions. Such a form of a formula is known as Positive Normal Form (henceforth PNF form) [12].

When specifying abstract concurrent systems, the atomic state formulas will take the form $\tilde{v}_i = \tilde{d}_i$ instead of $v_i = d_i$. CTL [11] is a restricted subset of CTL* in which the $\forall$ and $\exists$ path quantifiers may only precede a restricted set of path formulas. CTL is of interest because there is a very efficient model-checking algorithm for it [13]. $\forall CTL^*$ and $\forall CTL$ [14] are restricted subsets of CTL* and CTL respectively in which the only path quantifier allowed is $\forall$. These two logics are sufficient to express many of the properties that arise when verifying programs. As we will see, these logics will also be used when the conditions needed for exactness do not hold.

A path in M is an infinite sequence of states $p = s_0 s_1 s_2 \ldots$ such that for every $i \in \mathbb{N}$, $R(s_i, s_{i+1})$. The notation p^n will denote the suffix of p which begins at s_n. If $p = s_0 s_1 s_2 \ldots$ is a sequence of states from S, we denote the sequence $h(s_0)h(s_1)h(s_2)\ldots$ by $h(p)$. Satisfaction of a state formula f by a state s ($s \models f$) and of a path formula γ by a path p ($p \models \gamma$) is defined inductively as follows.

- $s \models true$ and $s \not\models false$
- if $s = (e_1, \ldots, e_n)$, $s \models v_i = d_i \Leftrightarrow e_i = d_i$
- $s \models v_i \neq d_i \Leftrightarrow s \not\models v_i = d_i$
- $s \models f \wedge g \Leftrightarrow s \models f \wedge s \models g$
- $s \models \forall \gamma \Leftrightarrow \forall p' s_0 = s \wedge p \models \gamma$
- $s \models \exists \gamma \Leftrightarrow \exists p' s_0 = s \wedge p \models \gamma$
- $p \models f \Leftrightarrow s_0 \models f$
- $p \models f \wedge g \Leftrightarrow p \models f \wedge p \models g$
- $p \models X f \Leftrightarrow p^1 \models f$
- $p \models f \cup g \Leftrightarrow \exists n \in \mathbb{N} \mid p^n \models g$ and $\forall i < n, p^i \models f$

The notation $M \models f$ indicates that every initial state of M satisfies the formula f.

4.2. Abstract Model Checking

In the case of an abstract concurrent system $\tilde{M}$, we define satisfaction in exactly the same way except the atomic formula $\tilde{v}_i = \tilde{d}_i$ is true at state $(\tilde{e}_1, \ldots, \tilde{e}_n)$ if and only if $\tilde{e}_i = \tilde{d}_i$.

We now define a translation α between formulas describing M and formulas describing the abstract concurrent system $\tilde{M}$. Our goal is to be able to check a formula $\alpha(f)$ on $\tilde{M}$ and infer that the corresponding formula f holds for M.

- $\alpha(true) = true$, $\alpha(false) = false$
- $\alpha((v_i = d_i)) = (\tilde{v}_i = \tilde{d}_i) \mid \tilde{d}_i = h_i(d_i)$
- $\alpha(v_i \neq d_i) = \neg\alpha(v_i = d_i)$
- $\alpha(sf_1 \wedge sf_2) = \alpha(sf_1) \wedge \alpha(sf_2)$
- $\alpha(sf_1 \vee sf_2) = \alpha(sf_1) \vee \alpha(sf_2)$
- $\alpha(\forall pf) = \forall \alpha(pf)$
- $\alpha(\exists pf) = \exists \alpha(pf)$
- $\alpha(pf_1 \wedge pf_2) = \alpha(pf_1) \wedge \alpha(pf_2)$
- $\alpha(pf_1 \vee pf_2) = \alpha(pf_1) \vee \alpha(pf_2)$
- $\alpha(Xpf) = X\alpha(pf)$
- $\alpha(pf_1 \cup pf_2) = \alpha(pf_1) \cup \alpha(pf_2)$

Thus, a temporal property f is abstracted by this translation α, which guarantees that every model of the abstract property corresponds to a model of the concrete one. This property approximation is exact.

Example 2: the following property expresses the mutual exclusion (at any time only one mathematician is eating)

$$\begin{aligned} \varphi_1 &= \forall G(\neg(m_0 = eat \wedge m_1 = eat)) \\ \tilde{\varphi}_1 &= \alpha(\varphi_1) = \forall G(\neg(\tilde{m}_0 = eat \wedge \tilde{m}_1 = eat)) \end{aligned} \tag{10}$$

The property to express that the mathematician m_0 eventually eats is

$$\begin{aligned} \varphi_0 &= \forall F(m_0 = eat) \\ \tilde{\varphi}_0 &= \alpha(\varphi_0) = \forall F(\tilde{m}_0 = eat) \end{aligned} \tag{11}$$

Lemma 1: Assume $M \hat{\sqsubseteq}_k \tilde{M}$. If p is a path in M, then $h(p)$ is a path in $\tilde{M}$.

Proof: The relation $M \hat{\sqsubseteq}_k \tilde{M}$ means $\tilde{M}$ simulates M. Thus, each execution (path) in M has its corresponding abstract execution in $\tilde{M}$.

Using this observation, we present the main preservation theorem: formulas that hold at the abstract level also hold for the concrete system.

Theorem 2: Assume $M \hat{\sqsubseteq}_k \tilde{M}$, and let f be a $\forall CTL^*$ formula describing $\tilde{M}$. Then $\tilde{M} \models \alpha(\varphi) \Rightarrow M \models \varphi$.

Proof: The relation $\tilde{M} \models \alpha(\varphi)$ means that there is not a path in $\tilde{M}$ which falsifies the property $\alpha(f)$ ($\alpha(f)$ is a $\forall CTL^*$ formula). By Lemma 1, as all the

paths in M have their corresponding in $\widetilde{M}$, then there is no path in M falsifying j. Thus, $M \square j$.

Note that this result only talks about preserving the truth of formulas. These formulas describe behaviour that should hold on all paths from a state. Since the abstraction process adds extra behaviours to the model, properties describing the existence of a path may not be preserved in the same manner. Thus, verifying something like absence of deadlock at the abstract level requires proving a stronger progress property.

In the case where $\widetilde{M}$ exactly approximates M, we also have the converse result: satisfaction at the concrete level implies satisfaction at the abstract level. We note that paths at the abstract level and at the concrete level exactly coincide.

Lemma 2: Assume $M \approx_h \widetilde{M}$, and let p be an infinite sequence of states from S (the set of states of M). Then p is a path in M if and only if $h(p)$ is a path in $\widetilde{M}$. Then we have the analogue of Theorem 2, except now going both ways.

Proof: By construction $\widetilde{M}$ simulates M. Thus, it is sufficient to prove now that M simulates $\widetilde{M}$. To show this, it is sufficient to prove that for each pair of abstract states $\%_1$ and $\%_2$, if $\%_2$ is a successor of $\%_1$ by t in the abstract system, then, for every pair s_1 and s_2 of states in the concretisation of $\%_1$ and $\%_2$, s_2 is the successor of s_1 by t in the original system. Every concrete state s_1 in the concretisation of $\%_1$ satisfies the guard of t, and every successor s_2 of s_1 is in the concretisation of $\%_2$. Thus, M simulates $\widetilde{M}$.

Theorem 3: Assume $M \approx_h \widetilde{M}$, and let j be a CTL* formula describing $\widetilde{M}$. Then $\widetilde{M} \models \alpha(\varphi) \Rightarrow M \models \varphi$.

Proof: The proof is trivial because $M \not\approx_h \widetilde{M}$ means $M \hat{\square}_h \widetilde{M}$ and $\widetilde{M} \hat{0}_h M$.

The strong preservation result allows us to avoid false negative results (which can be produced by spurious behaviours) by mapping abstract error traces to concrete executions violating the property. However, the condition for strong preservation requires that $\widetilde{M}_a$ be deterministic. This is usually not the case. However, in the next section we will identify the spurious behaviours and we will show how to remove them. The result is a deterministic abstract system, which is an accurate approximation, and then it is exact.

5. Removing Spurious Behaviours

In this section, we will propose a method for abstracting the abstract concurrent system $\widetilde{M}$ to remove the spurious behaviours. The new abstract concurrent system ($\widetilde{M}$) should contain all the behaviours that are in the original concurrent system. With this method, we will reduce the size of $\widetilde{M}$ by restricting its behaviour instead of its structure (states and transitions).

5.1. Defining Spurious Behaviours

A non-deterministic transition can yield an ordinary non-deterministic path (we call it Form A, Figure 3) or a non-deterministic loop (Form B, Figure 4) [15, 16, 17].

The form A means that at its states, it is possible to have non deterministic abstract variables and the model-checking system can take the values (or states) to produce the path following the appropriate transitions (coloured states) to give a counterexample by which it demonstrates the non satisfaction of the property. This abstract error trace represents a possible behaviour in the concrete system, which is an equivalent valid concrete error trace, because during construction it was not possible to give deterministic values to those abstract variables.

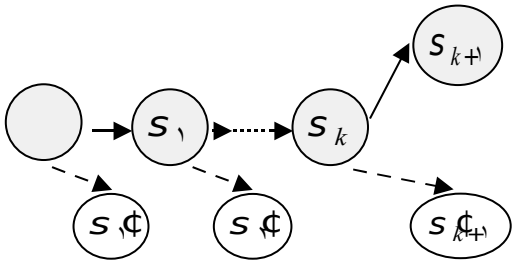


Figure 3. Non deterministic path.

In contrast, the form B (Figure 3) means that there is a possible spurious loop from the state s_i until the state s_k then it returns back to the state s_i. In the abstract system this loop is an infinite loop because of the non-deterministic aspect, but it can be a finite loop in the concrete system, which means that after a finite number of iterations the behaviour of the concrete system will take a transition to one of the uncoloured states inside the loop if there is one.

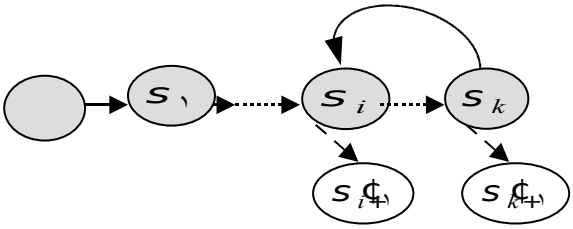


Figure 4. Non-deterministic loop.

Theorem 4: Any infinite loop in the abstract system is a spurious infinite loop if one of its transitions contains a non-deterministic abstract variable.

Proof: We assume that the abstract system contains an infinite loop. An infinite loop contains a finite set of transitions from the abstract system to be executed forever. Then the transition from one state to another is deterministic to make this infinite loop. So, all the guards of transitions are deterministic which means

that there are not non-deterministic variables. This is a contradiction with what it was supposed.

5.2. **Abstraction for Removing Spurious Behaviours**

After the identification of the spurious behaviours and their causes that are non-deterministic transitions, we present the method of removing these spurious behaviours. This method adds components to be synchronized with the original abstract concurrent system.

For each non-deterministic transition we parallel compose a component to be synchronized with the original abstract concurrent system. With this parallel composition we will restrict the abstract concurrent system to not execute the spurious behaviours (spurious loops).

$$\overline{P} = \tilde{P} \Big\| \frac{k}{i=1} \tilde{P}_i \tag{12}$$

where, $\tilde{P}$ is the original abstract program with possibly spurious behaviours, k is the number of non-deterministic transitions in $\tilde{P}$ and $\tilde{P}_i$ is the corresponding component to the non-deterministic transition to be parallel composed with $\tilde{P}$.

$\tilde{P}_i =\{$
$\tilde{R} =\{$
$(\tilde{\sigma}=B_i \wedge next(\tilde{\sigma}) =\neg B_i) \vee$
$(\tilde{\sigma}=\neg B_i \wedge next(\tilde{\sigma}) =B_i)$
$\}$
$\tilde{I} =(\tilde{\sigma}=B_i)$
$=true$
$\}$

where B_i is a new Boolean variable used to remove the spurious loop.

Example 3: The abstract program in Example 2 contains two non-deterministic transitions. The abstraction of this program gives the following abstract program.

5.3. Experimental Results

Symbolic Model Verifier (SMV) [18] is a model-checking tool; it has an automaton-based language to specify systems. SMV uses an automatic decision procedure to verify the system specification against CTL properties. The equivalent SMV module of the abstract program that is mentioned in Example 2 is:

$\tilde{P}: \{$

$\tilde{R}: \{$

$(s =\langle think, m_1, o, B_1, B_2 \rangle \wedge next(s) =\langle eat, m_1, o, B_1, B_2 \rangle) \vee$
$(s =\langle eat, m_1, o, B_1, B_2 \rangle \wedge next(s) =\langle think, m_1, e, B_1, B_2 \rangle) \vee$
$(s =\langle eat, m_1, e, B_1, B_2 \rangle \wedge next(s) =\langle think, m_1, o, B_1, B_2 \rangle) \vee$
$(s =\langle m_2, think, e, B_1, B_2 \rangle \wedge next(s) =\langle m_2, eat, e, B_1, B_2 \rangle) \vee$
$(s =\langle m_2, eat, e, B_1, B_2 \rangle \wedge next(s) =\langle m_2, think, e, \neg B_1, B_2 \rangle) \vee$
$(s =\langle m_2, eat, e, \neg B_1, B_2 \rangle \wedge next(s) =\langle m_2, think, o, B_1, B_2 \rangle) \vee$
$(s =\langle m_2, eat, o, B_1, B_2 \rangle \wedge next(s) =\langle m_2, think, o, B_1, \neg B_2 \rangle) \vee$
$(s =\langle m_2, eat, o, B_1, \neg B_2 \rangle \wedge next(s) =\langle m_2, think, e, B_1, B_2 \rangle)$
$\}$
$=\langle think, think, e, B_1, B_2 \rangle$
$, =true$
$\}$

```
MODULE main
VAR
  n : {e, o} ;
  m0, m1 : {think, eat} ;
ASSIGN
  init(n) := e ;
  init(m0) := think ;
  init(m1) := think ;
TRANS
  (m0 = think & n = o & next(m0) = eat &
   next(n) = o & m1 = next(m1)) |
  (m0 = eat & n = o & next(m0) = think &
   next(n) = e & m1 = next(m1)) |
  (m0 = eat & n = e & next(m0) = think &
   next(n) = o & m1 = next(m1)) |
  (m1 = think & n = e & next(m1) = eat &
   next(n) = e & m0 = next(m0)) |
  (m1 = eat & n = e & next(m1) = think &
   next(n) = e & m0 = next(m0)) |
  (m1 = eat & n = e & next(m1) = think &
   next(n) = o & m0 = next(m0)) |
  (m1 = eat & n = o & next(m1) = think &
   next(n) = e & m0 = next(m0)) |
  (m1 = eat & n = o & next(m1) = think &
   next(n) = o & m0 = next(m0))
INVAR
  1
SPEC
  AG(!(m0 = eat & m1 = eat))
SPEC
  AF(m0 = eat)
```

Figure 5. SMV program for example 2.

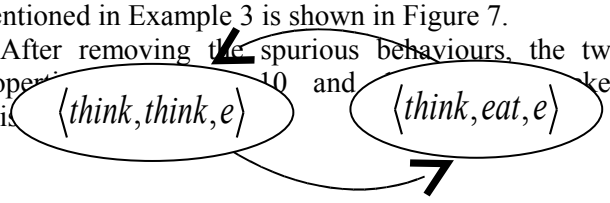
The model checking results, of the constructed model from this abstract program using the SMV model checker are:

- The first property of mutual exclusion (at any time only one mathematician is eating) expressed in equation 10 is checked to be satisfied.
- But the property (the mathematician m_0 eventually eats) expressed in equation 11 is checked to not be verified. The generated counter example is as follows:

Figure 6. Spurious loop.

This is a spurious loop. The equivalent SMV program after removing the spurious behaviours, as mentioned in Example 3 is shown in Figure 7.

After removing the spurious behaviours, the two properties (equation 10 and equation 11) are checked satisfied.




```
MODULE main
 VAR
  n : {e, o} ;
  m0, m1 : {think, eat} ;
  B1, B2 : boolean ;
 ASSIGN
  init(n) := e ;
  init(m0) := think ;
  init(m1) := think ;
 TRANS
  (m0 = think & n = o & next(m0) = eat &
   next(n) = o & m1 = next(m1) & next(B1)
   = B1 & next(B2) = B2) |
  (m0 = eat & n = o & next(m0) = think &
   next(n) = e & m1 = next(m1) & next(B1)
   = B1 & next(B2) = B2) |
  (m0 = eat & n = e & next(m0) = think &
   next(n) = o & m1 = next(m1) & next(B1)
   = B1 & next(B2) = B2) |
  (m1 = think & n = e & next(m1) = eat &
   next(n) = e & m0 = next(m0) & next(B1)
   = B1 & next(B2) = B2) |
  (m1 = eat & n = e & B1 & next(m1) =
   think & next(n) = e & m0 = next(m0) &
   next(B1) = !B1 & next(B2) = B2) |
  (m1 = eat & n = e & !B1 & next(m1) =
   think & next(n) = o & m0 = next(m0) &
   next(B1) = !B1 & next(B2) = B2) |
  (m1 = eat & n = o & B2 & next(m1) =
   think & next(n) = e & m0 = next(m0) &
   next(B1) = B1 & next(B2) = !B2) |
  (m1 = eat & n = o & !B2 & next(m1) =
   think & next(n) = o & m0 = next(m0) &
   next(B1) = B1 & next(B2) = !B2)
 INVAR
  1
 SPEC
  AG(!(m0 = eat & m1 = eat))
 SPEC
  AF(m0 = eat)
```

Figure 7. SMV program for example 3.

6. Conclusion

We have presented a method to generate abstractions without spurious behaviours of concurrent systems. The abstraction process takes as input a program based on relational expressions of the transition relation and the initial states. Then, it generates an accurate abstraction using the abstract interpretation framework. The abstract system is then abstracted using the method of restrictions to remove the spurious behaviours.

We have used the model checker [18] to verify our method on the examples: the mathematician dining (presented in this paper), the bakery protocol, and the greatest common divisor.

Our future work is to develop a tool implementing this approach and testing large concurrent systems. On the other hand investigating special cases for the framework of abstract interpretation.

References

[1] Dwyer M., Clarke L., Cobleigh J., and Naumovich G., "Flow Analysis for Verifying Properties of Concurrent Software Systems," *ACM Transactions on Software Engineering and Methodology*, vol. 13, pp. 359-430, 2004.

[2] Siegel S., Mironova A., Avrunin G., and Clarke L., "Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs," *in Proceedings of the International Symposium on Software Testing and Analysis*, pp. 157-168, 2006.

[3] Peled D. and Qu H., "Enforcing Concurrent Temporal Behaviors," *International Journal of Foundations of Computer Science*, vol. 17, pp. 743-762, 2006.

[4] Ostrovsky K., *On Modelling and Analysing Concurrent Systems*, PhD Dissertation, Computer Science and Electronic Engineering, Chalmers University of Technology, Sweden, 2005.

[5] Clarke E., Grumberg O., and Peled D., *Model Checking*, The MIT Press, 2000.

[6] Cousot P., "Abstract Interpretation," *ACM Computing Surveys*, vol. 28, pp. 205-212, 1996.

[7] Dams D., Gerth R., and Grumberg O., "Abstract Interpretation of Reactive Systems," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 115-132, 1997.

[8] Chouali S., Julliand J., Masson P., and Bellegarde F., "PLTL Partitionned Model-Checking for Reactive Systems Under Fairness Assumptions," *ACM Transactions on Embedded*

Computing Systems, vol. 4, no. 2, pp. 267-301, 2005.

[9] Clarke E., Grumberg O., Jha S., Lu Y., and Veith H., "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752-794, 2003.

[10] Clarke E., Grumberg O., and Long D., "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 81-112, 1994.

[11] Clarke E., Emerson E., and Sistla A., "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 54-66, 1986.

[12] Emerson E. and Lei C., "Efficient Model Checking in Fragments of the Prepositional μ-Calculus," *in Proceedings of the First Annual Symposium of Logic in Computer Science*, 1986.

[13] Bryant R., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 78-105, 1986.

[14] Grumberg O. and Long D., "Model Checking and Modular Verification," *Lecture Notes in Computer Science* (LNCS), Springer, 1991.

[15] Bourahla M. and Benmohamed M., "Predicate Abstraction and Refinement for Model Checking VHDL State Machines," *Electronic Notes in Theoretical Computer Science*, Elsevier Science, vol. 66, pp. 3-17, 2002.

[16] Bourahla M. and Benmohamed M., "Abstract Model Checking Infinite State Systems," *in ACS/IEEE International Conference on Computer Systems and Applications*, pp. 83, 2003.

[17] Bourahla M. and Benmohamed M., "Verification of Real-Time Systems by Abstraction of Time Constraints," *in Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'2003),* IEEE Computer Society, 2003.

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

[19] Tzoref R. and Grumberg O., "Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation," *in Proceedings of the International Conference on Computer Aided Verification (CAV'06)*, 2006.

[20] Grumberg O., Lerda F., Strichman O., and Theobald M., "Underapproximation-Widening for Multi-Process Systems," *in Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, 2005.

[21] Shoham S. and Grumberg O., "Monotonic Abstraction-Refinement for CTL," *in Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, 2004.

[22] Rabinovitz I. and Grumberg O., "Bounded Model Checking of Concurrent Programs," *in Proceedings of the International Conference on Computer Aided Verification (CAV'05)*, 2005.

Mustapha Bourahla has a PhD degree in computer science from the University of Biskra, Algeria in 2007 and has the Master degree in computer science from the university of Montreal, Canada in 1989. He was a member of the VHDL group at Bell-Northern Research, Ottawa, Canada during 1989-1993. He worked for Bell Canada for one year. Currently, he is teacher-researcher at the University of Biskra, Algeria. He has publications in the domains of VLSI and formal methods. His current research interests are in formal methods, especially model checking critical systems. He is a member of a research group working in the domains of VLSI and formal methods at the University of Biskra, Algeria.