

Abstract Matching for Software Model Checking^{*}

Pedro de la Cámara, María del Mar Gallardo, and Pedro Merino

University of Málaga, Campus de Teatinos s/n,
29071, Málaga, Spain
pedro.delacamara@gmail.com
{gallardo, pedro}@lcc.uma.es

Abstract. Current research in software model checking explores new techniques to handle the storage of visited states (usually called the heap). One approach consists in saving only parts or representations of the states in the heap. This paper presents a new technique to implement sound abstract matching of states. This kind of matching produces a reduction in the number of states and traces explored. With the aim of obtaining a useful result, it is necessary to establish some correctness conditions on the matching scheme. In this paper, we use static analysis to *automatically* construct an abstract matching function which depends on the program and the property to be verified. The soundness of the static analysis guarantees the soundness of verification. This paper describes the overall technique applied to **Spin**, the correctness issues and some examples which show its efficiency.

Keywords: State Explosion, Model Extraction, Static Analysis.

1 Introduction

Using model checking techniques for software verification usually involves the manual construction of high-level models. This construction process allows designers to exploit many abstraction techniques in order to reduce the size of the model and its complexity. However, it requires a deep understanding of both the real software and the modelling language features. Furthermore, the manual construction process is susceptible to human error due to misunderstandings or simply programming bugs. These errors are especially subtle because they may lead to false results in the process of model checking, thus failing to detect the presence of errors in the program being verified. Recently, many projects are developing automatic *model extraction* techniques, that can contribute to solving this problem with minimal human interaction (see Feaver [8], JPF1 [6] and Bandera [2]). However, extracted models are too cumbersome and have too many implementation details. Therefore, it is desirable to develop further optimization (abstraction) techniques in order to reduce the complexity of the model. This paper is devoted to a new optimization technique which reduces the explored

^{*} This work has been supported by the Spanish MEC under grant TIN2004-7943-C04.

state space in models extracted for Spin. The technique is based on the use of abstractions to implement the matching functions to discard visited states, as introduced in [7] and [10]. The main novelty of our *abstract matching* method is the ability to preserve the verification results, due to how the abstract matching function is constructed.

The method proposed in [7] consists in hiding specific C variables in such a way that they are never used to compare global states and decide whether they have been visited or not. However all the variables are always visible when making backtracking or producing new states. The mechanism used to implement this abstraction scheme in [7] is based on a new Promela extension that allows verifier to hide variables when performing the matching of states.

Our approach is an extension of the implementation mechanism in [7], that adds soundness to the verification results for a given class of abstraction functions. In particular, we employ a property-oriented static analysis to locate the set of variables that should be hidden or matched after every execution step. The analysis is a variant of dependency analysis, called *influence analysis*, that produces a set of visible variables for every statement in the model. These variables should be visible after executing the statement (after producing a new global state), in such a way that their values are considered to match the global state.

In our method, the correctness conditions and the algorithm used to carry out the static analysis can be changed depending on the properties to be preserved during verification. In the paper we describe methods for three kinds of properties: a) code reachability, b) safety properties (state properties in Spin) and c) liveness properties (sequence properties in Spin). For all these cases, static analysis is done prior to verification, during the model extraction, producing a Promela model with property-oriented abstract matching. The new model is verified as usual with Spin.

The new approach can be directly implemented for other tools that perform model extraction for Spin (like FeaVer or Bandera), however we are integrating the static analysis in our tool SOCKETMC [3]. This tool is a model extractor focused on verifying *concurrent software with well defined-APIs*. The experimental results with the new optimization are very promising.

Regarding another closely related work, the implementation of abstract matching in [10] is based on applying predicate abstraction to the global states to be compared, in such a way that explicit hiding is not used. Predicate abstraction works matching over-approximations of the states, so the method can produce unsoundness when verifying properties. For that reason, a refinement method, assisted with a theorem prover, is used in order to improve the quality of the analysis.

As far as we know, our work contains valuable contributions compared with [7] and [10]. For example,

1. The method for obtaining the abstraction function based on static analysis can be done automatically
2. Static analysis provides a sound function for each given property.
3. The soundness conditions also allow the verification of liveness properties.

The paper is organized as follows. The preliminary material in Section 2 summarizes the extraction approach in our tool SOCKETMC, which is used as our first target tool to include the new optimization. Section 3 gives an overview of the method and its application to a real example. The soundness of the method is presented in the next two sections. Section 4 explains the static analysis called influence analysis, and Section 5 contains a discussion on the correctness of this approach. Conclusions are given in Section 6.

2 Model Extraction and SocketMC

The aim of tool SOCKETMC is to verify concurrent C applications that make an extensive use of operating system facilities through system calls. We have constructed a Spin oriented model of the behavior of the operating system API. This model is used to automatically obtain a correct abstraction of the software that makes use of this API. Following [8], we have defined a mapping from the original C code to extended Promela. The tool SOCKETMC automatically transforms each API call into Promela code preserving the semantics of the calls. The new Promela model constructed can be verified with standard Spin. Figure 1 shows the main parts of SOCKETMC, the parser and the model generator. The figure also shows the relevant role of the formal semantics given to the operating system API. The semantics is used as a reference to construct a sound Promela version of each API call.

Our basic mapping scheme works as shown in Figure 2. Given a set of C processes (`main()` functions), the mapping from the original code to Promela is done replacing every process (every `main()` function) with a `proctype()` definition. Then, the body of every `proctype()` is filled using the Promela extensions for C code (`c_decl`, `c_state`, `c_expr` and `c_code`). This is done breaking the original C code in the points where a call to API appears. The final Promela

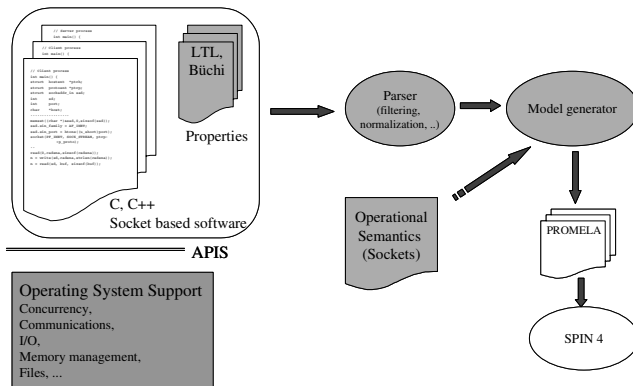


Fig. 1. Extracting models with SocketMC

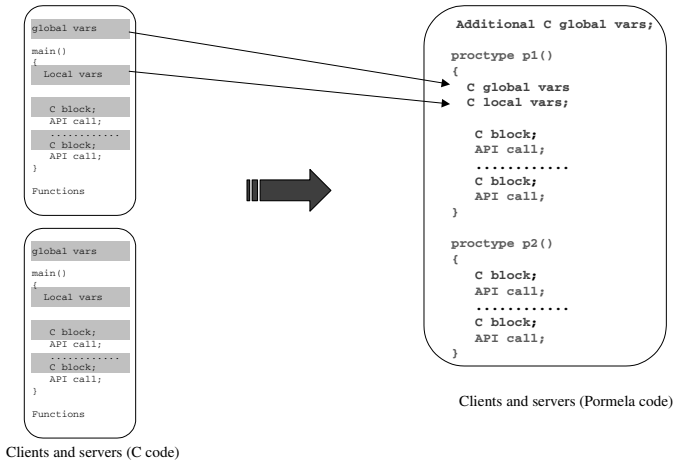


Fig. 2. Mapping scheme in SocketMC

code preserves the sequential execution of every *C block* code between two system calls. Thus, when verifying the model, *Spin* interleaves blocks and system calls as atomic instructions. The way of implementing the extraction, together with the semantic-driven API implementation, ensures the correctness of our verification model.

By default, the Promela models produced by the first version of the tool contain all the C variables in the original code. The approach presented in the following sections is oriented to automatically reduce the set of variables that should actually be managed to produce the state space. The proposals are implemented as new versions of the two components of SOCKETMC (the parser and the model generator in Figure 1), and the result is a new mapping scheme to extract the final Promela models. It is worth noting, however, that our method to construct abstract matching functions can be applied to other model extraction approaches and even to other model checking tools.

3 Sound Abstract Matching

The technique to include abstract matching in *Spin* and the problem of how to ensure the validity of abstract matching functions to preserve CTL* was originally presented by Holzmann and Joshi in [7]. The first issue, the implementation approach, is presented in the context of the nested depth-first search algorithm with abstraction described in [1]. The idea is to avoid starting a new search from a given state if an *essentially* equal state has been visited before. In summary, including abstraction when storing visited states works as follows. Given a global state s , abstraction consists in replacing the usual operation “add s to States”, that stores it as a visited state, by the new operation “add $f(s)$ to States”, where $f()$ represents the abstraction function. Function $f()$

generates the abstraction of \mathbf{s} to be matched and stored (note that in [1] and [7], operation `add` has a second argument that does not affect the abstraction process). It is worth noting that function `f()` is only used to cut the search tree, but the exploration is actually realized with the concrete state \mathbf{s} , without losing information. Observe that when we use abstraction during the model checking process as explained above, we explore a subset of the original state space. Thus, in this case, abstraction produces an *under-approximation* of the original model, in contrast to the usual applications of abstraction which produce *over-approximation*. In order to assure that the explored tree via abstract matching is equivalent to the original one, function `f()` has to satisfy some correctness conditions.

As proposed in [7], a particular version of function `f()` is implemented as a C function which is invoked within a `c_code` construct. The implementation also benefits from the `c_track` primitive to hide the values of C variables from the state-vector. Thus, the abstraction function computes abstract representations of the hidden data and copies the result onto the state-vector.

In [7], the authors do not address any particular method to generate `f()`, however they present necessary conditions to define sound abstract functions that preserve CTL properties. This is the starting point for our work. We provide implementable methods to produce abstraction functions, which are sound and oriented to the property to be checked.

Our Abstraction Approach. In our implementation scheme, abstraction functions are implemented in such a way that they can (automatically) identify the variables to be hidden from the state-vector in every global state, after the execution of every verification step. A simple case shows how it works. Let us consider the following code which can be obtained by a model extractor like the first version of SOCKETMC:

```
proctype p()
{
  c_track "&x" "sizeof(int)" "Matched"
  c_track "&y" "sizeof(int)" "Matched"
  ...
  L0: initialize();
  L1: c_code{x = 1};
  L2: c_code{y = x};
  ...
}
```

Note that in this code variables `x` and `y` are visible in the state-vector. Suppose that we extract the model assuming, by default, that C variables do not influence the verification of properties. Following this assumption, both variables `x` and `y` are declared as hidden (`UnMatched`). Consider now that we are interested in checking a particular property that needs the precise value of `x` after executing the code at L1. Then, in this case, the model extracted must keep variable `x` visible after executing the instruction at L1, as the following code shows:

```

proctype p()
{
  c_track "&x" "sizeof(int)" "UnMatched"
  c_track "&y" "sizeof(int)" "UnMatched"
  c_track "&x_" "sizeof(int)" "Matched"
  c_track "&y_" "sizeof(int)" "Matched"
  ....
  L0: atomic{initialize(); f(L0)};
  L1: c_code{x = 1; f(L1)};
  L2: c_code{y = x; f(L2)};
  ...
}
void f(int label)
{
  switch(label)
  {
    .....
    case L0:
      now.x_ = Hide()
      now.y_ = Hide()
    case L1:
      now.x_ = Show(x)
    case L2:
      now.x_ = Hide()
      now.y_ = Hide()
    .....
  }
}
}

```

This second version calls `f()` at any point where the global state should be stored. This function uses its argument to check the current execution point in the model.¹ The function updates the variables to be hidden (using `Hide()`) or updated (using `Show()`) before matching them with the current set of visited states, depending on the current label. For instance, variable `x` can be hidden until it is updated in `L1`. However, it is made visible at `L1` because it will be used to update `y`, and it is again hidden after updating `y`. The extra variables `x_` and `y_` are used to store the values of the real (hidden) variables or a representation of their values. We propose to construct `f()` using the information provided by a static analysis of the model. This construction approach for `f()` can be extended to models with multiple processes.

Example. We illustrate the use of this technique with a simple case study. In this example, we use a model of a simple server and check the property **P1** which states that “*If the process receives a message END, then it eventually leaves the main loop*”. In Figure 3 (left), it is possible to see the main loop of the server, including those variables which are visible at each control point in order to verify **P1** with abstract matching.

In the example, `READ` and `CREATERESPONSE` are actually non-deterministic selections returning a message from a limited set. `PREPROCESS` and `POSTPROCESS` are loops simulating heavy work between the reception of the message and the response.

¹ Note that it would not be necessary to pass the label as an argument of `f()`, if Promela would allow to access the current label of process `p` with some code such as `label = now.Pp->_label`

```

do ::c_expr{enter_loop}->
  atomic {
    READ(cRead,sock2,ReadBuf, sizeof(ReadBuf));
    c_code { f(21);};};
Message_Rx:
// P1: ReadBuf and cRead are visible ----- P2: ReadBuf and cRead are visible
  atomic {
    PREPROCESS(cRead,ReadBuf);
    c_code { f(22);}; };
// P1: ReadBuf and cRead are visible ----- P2: ReadBuf and cRead are visible
  atomic {
    CREATERESPONSE(cResp,ReadBuf,WriteBuf,cRead);
    c_code { f(23);}; };
// P1: WriteBuf and cResp are visible ----- P2: ReadBuf, WriteBuf and cResp are visible
  atomic {
    POSTPROCESS(cResp, WriteBuf);
    c_code { f(24);}; };
// P1: WriteBuf and cResp are visible ----- P2: ReadBuf, WriteBuf and cResp are visible
  atomic {
    WRITE(cResp,sock2,WriteBuf,cResp);
    c_code { f(25);};
  };
// P1: cResp is visible ----- P2: ReadBuf and cResp are visible
  c_code{ enter_loop=(cResp)>0; f(26); };
// P1: enter_loop is visible ----- P2: ReadBuf and enter_loop are visible
:: else ->
  c_code { f(27);}; break;
od;

```

Fig. 3. Visibility rules for ReadBuf with properties **P1** (left) and **P2** (right)

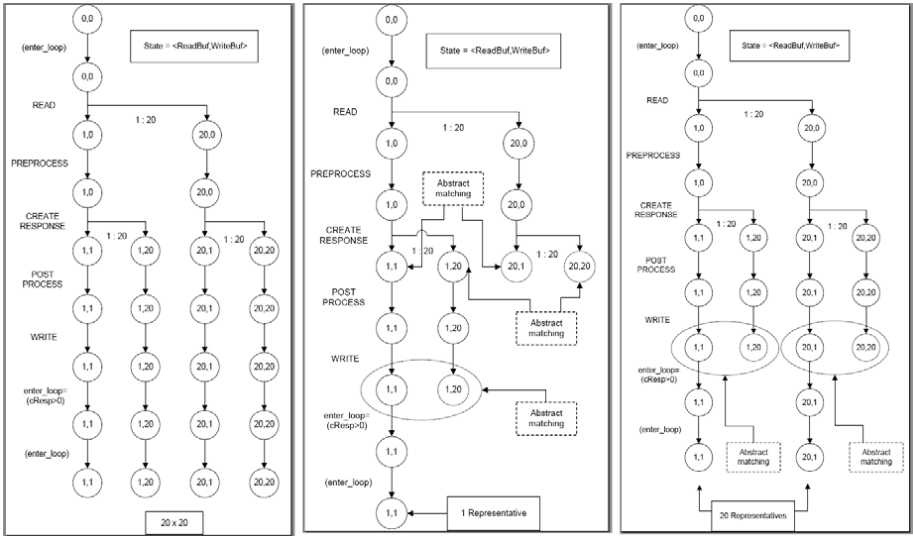
Variable `ReadBuf` is a receiving buffer that may take multiple non-deterministic values. Suppose that the static analysis to verify **P1** determines that `ReadBuf` is not significant from `CREATERESPONSE` on. Thus, if `ReadBuf` is hidden after executing `CREATERESPONSE`, we avoid multiple re-exploration of the executing paths starting at this point. It is clear that the amount of saved memory (and time) depends on the range of values `ReadBuf` may take. The static analysis decides to hide `ReadBuf` because property **P1** only checks `ReadBuf` at label `Message_Rx`.

Suppose that we modify the property in such a way that we need to check `ReadBuf` at every control point. For example, assume a new property **P2** stating that “`ReadBuf` never contains a `RETRY` message”. In order to verify **P2**, we cannot hide variable `ReadBuf` after `CREATERESPONSE`. Figure 3 (right) shows the result of the new model extracted. Note that the set of visible variables associated to the statements has changed.

Figure 4 shows the performance of our proposal for the previous example. We have assumed that variables `ReadBuf` and `WriteBuf` may take 20 different values, and that loops `PREPROCESS` and `POSTPROCESS` iterate 100 times. The table shows the state space explored in three cases: (1) without abstract matching (2) with abstract matching oriented to property **P1**, and (3) with abstract matching oriented to property **P2**.

Figure 5 explains the different reduction results obtained for the three cases described above. When we do not use abstract matching, we have 20x20 different traces to explore at the end of the loop (left column). If we hide `WriteBuf` after `WRITE`, the number of traces is divided by 20 (right column). Additionally, when

	No Abstract Matching	Abstract Matching. (ReadBuf partial hiding)	Abstract Matching (ReadBuf fully visible)
State-vector	152 bytes	28 bytes	28 bytes
Errors	26753	23	862
States stored	787705	117	1761
Total memory	90.929 MB	2.724MB	5.389 MB
Elapsed time	14:26.44	0:00.72	0:03.33

Fig. 4. Test Results

Fig. 5. Reachability trees for the case study

ReadBuf is partially hidden, we have only one representative for all the traces (center column). This phenomenon also happens with other variables and it is the main reason for state reduction.

Optimizations. The actual representation of the visible variables is a bit more complex than shown before. Instead of using duplicate variables (like x_+ and y_- , in the previous example), we employ a vector as described below.

```

void f(int label) {
    switch (label){
        ...
        case 21:    now.idVector[0]=Hide();
                  now.idVector[1]=ShowVariable(1, sizeof(cRead), &cRead);
                  now.idVector[2]=Hide();
                  ... }
                  break;
        ... }
    
```

Global variable `idVector` is in fact the abstract representation of the visible variables which are implemented as a vector of identifiers. Auxiliary function `ShowVariable()` computes the identifier associated to the current value of a given variable, while the `Hide()` function returns always a null identifier.

A lookup table allows us to map identifiers to values. This table is dynamically updated in such a way that it always keeps an entry for every reached value. In addition, the table is never included in the usual data structures of the model checker (stack and heap). It can be seen as a global data structure for all execution paths. The results shown in Figure 4 were obtained with this technique.

Experimental results show that our approach to automatically construct abstract functions is very promising, however we still have to discuss about how to ensure soundness. The following sections are devoted to this key issue.

4 Static Analysis

In this section, we describe the static analysis from which we construct sound abstract matching functions. In particular, we develop the so-called *influence analysis* (IA) to annotate each program point with a set of *significant* variables needed to correctly analyze a given property. This static analysis is a refinement of the *live variables analysis* given in [9] (adapted to the case of Promela) where the properties of interest to be verified are taken into account. Our analysis is also related to cone of influence reduction, described in [4]. However, as far as we know, the cone-of-influence technique does not take into account that while a given variable could be visible for some states it then could become invisible for successor states of the same trace.

Note that in the analysis we do not distinguish between C variables and pure Promela variables, although currently we have only implemented abstract matching for C variables. In order to simplify the presentation, we only use the traditional Promela syntax for the variables and we have omitted the explicit treatment of some Promela instructions such as those dealing with channels (including rendez-vous).

The *influence analysis* is used to decide which variables should be visible at each program point during the model checking process. It determines for each program control point the variables which *influence* a given set of variables V of interest. The analysis then records the variables which are *alive* wrt a particular property. Thus, if a variable does not affect any variable in V at a given program point, we may hide it since its current value is not relevant for the verification.

Clearly, the most precise analysis is the one attaching the smallest set of variables to each program point. In the following sections, we show different versions of IA. Each extension gives us a different precision degree for the analysis and the abstract matching function induced preserves a different set of program properties. The first analysis IA_1 is the most precise one, it produces the *best* abstract matching function, the one inducing the best state space reduction.

However, IA_1 only preserves the code reachability tree of the original Promela model. In addition, since global variables must be dealt with very carefully, IA_1 assumes that the model under analysis has only local variables. The second analysis IA_2 produces bigger sets of variables, but it preserves *safety properties*. The third analysis IA_3 studies the case of models with global variables, and, finally, IA_4 is the least precise analysis, but in contrast, it preserves *liveness properties*.

4.1 Influence Analysis for Promela Models

Given a Promela program M , the goal of IA is to associate each program point in M with the least set of variables whose value is needed to analyze M .

Let \mathcal{V} be the set of program variables. Informally, given $x, y \in \mathcal{V}$, we say that variable y *influences* variable x at a given program point, if there exists an execution path in M from this point to an assignment $x = exp$ and the current value of y is used to calculate exp , that is, if the current value of y is *needed* to construct the value of x in the future. In Section 5, when we prove the correctness of the analysis, we give a formal definition of the *influence* notion.

In this section, we focus on describing the four above-mentioned IA analyses. We first define the input language and some previous notions.

Let *Inst* be the set of all valid Promela instructions including the *Basic* statements (boolean expressions, assignments, and input/output instructions over channels), *If*, *Do*, *Atomic*, *Unless* statements, etc. In the sequel, we do not distinguish between C variables and pure Promela variables. We also consider that blocks of C instructions inside `c_code` are *Basic* instructions, and that the C boolean expressions are managed as pure Promela boolean expressions. In order to simplify the analysis, we assume that *Do* instructions are implemented using *If* and *goto* statements. In addition, we assume that branches of *If* instructions always begin with a boolean expression followed by at least one statement. We use *true* and *skip* to complete the instruction when necessary (for example, see the codes of Figure 6). Finally, when an *else* branch appears, we assume that it

```

active proctype p1(int n) {
  int x = n;
  int y = 1;
  L1:if
    :: x > 0 -> L2: x = x - 1;
                L3: y = 2 * y; goto L1
    :: else -> L4: printf(y); goto End;
  fi;
End:
}

active proctype p2() {
  int x1,x2,x3,x4;
  L1:if
    :: true -> L2: x2 = 0;
    :: true -> L3: x2 = 1;
  fi;
  L4: x1 = x2;
  L5: if
    :: x3 < 2 -> L6: x1 = x1 + 1; goto L5;
    :: else -> L7: skip;
  L8: if
    :: true -> L9: x4 = 0;
    :: true -> L10: x4 = -1;
  fi;
  L11: if
    :: x4 >= 0 -> L12: assert(x1 == 2);
    :: else -> L3: skip
  fi
fi;
End:
}

```

Fig. 6. Two Promela processes

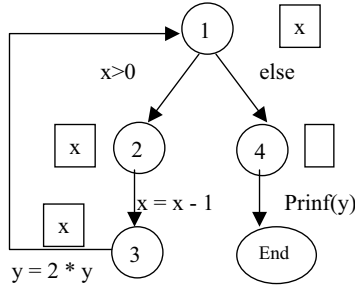


Fig. 7. Result of IA_1 for process p1

has been replaced by the corresponding boolean expression. In the sequel, *BoolExp* will denote the set of boolean expressions that can be constructed with the usual arithmetical and boolean operators and with the constant and variables of the model.

As commented above, the first approach IA_1 is focussed on preserving the code reachability tree of M , that is, the abstract matching function induced by IA_1 should preserve each possible execution path in the original model. Since the control flow is determined by boolean expressions and *If* instructions, in order to simulate all the execution paths, we need to record all possible values of the variables appearing in the guards of the control statements.

Let us define the set $Init \subseteq \mathcal{V}$ of all program variables appearing in some boolean expression in M . We perform the *influence analysis* IA_1 attaching each program counter of M with the set of program variables influencing some variable in the set $Init$.

For instance, set $Init$ is respectively defined as $\{x\}$ and $\{x3, x4\}$ for the *Promela* codes of processes $p1$ and $p2$ given in Figure 6. In addition, Figure 7 shows the intended result of IA_1 for $p1$. For this process, the static analysis associates the set $\{x\}$ with the labels $L1$, $L2$, and $L3$. The usefulness of the analysis is clear. If we are interested in knowing whether a particular label of process $p1()$ is reachable, we only have to store variable x at labels $L1$, $L2$, and $L3$. In particular, variable y may be completely hidden because its value is not relevant for this analysis.

The rest of this section is devoted to formalizing IA_1 . Let $M = P_1 || \dots || P_n$ be the *Promela* model to be analyzed, where each P_i denotes a concurrent process declared in M . We assume that all instructions of the *Promela* model M to be analyzed are *labelled*, i.e., each one has the form $L : ins$ where $L \in \mathcal{L}$ is a unique label of the instruction ins . Labels may be defined by the user or automatically assigned. *End* denotes the set of user-defined labels starting with *end*. The code of each process is finished with a label $L \in End$. Note that labels represent program counters of processes. For the sake of simplicity, we assume that labels in each *Promela* process are different.

Function $I : \mathcal{L} \rightarrow Inst$ returns the *basic/If* instruction following a label. For instance, considering the code of process $p2$ of Figure 6, $I(L6) = x1 = x1 + 1$ and $I(L1) = if :: true- > L2 : x2 = 0; :: true- > L3 : x2 = 1; fi$.

Let us define function $next : \mathcal{L} \rightarrow \mathcal{L}$ which associates each label l with the label pointing to the *basic/If* instruction following $I(l)$. For example, in the process $p2$ of Figure 6, $next(L2) = L4$, and $next(L6) = L5$. Function $next$ is well defined because we always apply it to labels pointing to *basic* instructions, although it may return labels pointing to a *basic/If* statement.

Given any expression or instruction, we denote with $var(e) \subseteq \mathcal{V}$ the set of program variables appearing in e . In order to simplify the description, we first define how to apply the static analysis to a simple process P , and then, we extend it to a whole program M composed of several concurrent processes. We also assume that M contains only local variables, and then we again extend the analysis to the case of global variables.

The static analysis \mathbf{IA}_1 is formally constructed as the least fixed point of function $F1 : \wp(\mathcal{V})^{\mathcal{L}} \rightarrow \wp(\mathcal{V})^{\mathcal{L}}$ which represents a transformation function which transforms vectors of $|\mathcal{L}|$ components, where $|\mathcal{L}|$ is the number of labels in the system. Each component that corresponds to a label is a set of variables.

Given $\vec{s} = \{s_l | l \in \mathcal{L}\} \in \wp(\mathcal{V})^{\mathcal{L}}$, the l component of \vec{s} is denoted as $\vec{s}(l)$ and it corresponds to a subset of variables attached to label l at a given moment during the computation of \mathbf{IA}_1 . Similarly, we denote the l -component of $F1(\vec{s})$ as $F1(\vec{s})(l)$. $F1$ is a backward analysis, that is, it extracts information following the reverse control flow of the program. Thus, to calculate the significant variables at a given label $l \in \mathcal{L}$, we have to collect all variables which are needed by any execution path starting at this point. Recall that a variable is needed at l if its value is needed for executing the next instruction $I(l)$ or for executing any instruction following $I(l)$. Considering this, given $\vec{s} \in \wp(\mathcal{V})^{\mathcal{L}}$ we construct $F1(\vec{s})(l)$ making use of function $F1^*$, defined below, as follows:

$$\begin{aligned} F1(\vec{s})(l) &= F1^*(I(l), \vec{s}(next(l))) \text{ if } I(l) \in Basic \text{ and} \\ F1(\vec{s})(l) &= \cup_{i=1}^n F1^*(b_i, \vec{s}(l_i)) \text{ if } I(l) = if :: b_1 \rightarrow l_1 : \dots :: b_n \rightarrow l_n : \dots ; fi \end{aligned}$$

where $F1^* : Basic \times \wp(\mathcal{V}) \rightarrow \wp(\mathcal{V})$ calculates the significant variables before executing a basic instruction as:

$$\begin{aligned} F1^*(x = exp, s) &= \begin{cases} s & \text{if } x \notin s \\ s - \{x\} \cup var(exp) & \text{if } x \in s \end{cases} \\ F1^*(bool, s) &= s \cup var(bool), \text{ bool being a Boolean expression} \end{aligned}$$

That is, assignment $x = exp$ modifies set s only if it has been deduced that x influences some variable in $Init$. In that case, the effect of $x = exp$ consists of introducing in s all variables appearing in exp , excluding x because its value is changed in the assignment. In addition, all variables appearing in a boolean expression influence variables in $Init$ (in fact, they belong to $Init$).

Define $\forall l \in \mathcal{L}. s_l = \emptyset$, and consider the initial vector $\vec{s}_{init} = \{s_l\}_{l \in \mathcal{L}}$. Then, the static analysis $\mathbf{IA}_1 \in \wp(\mathcal{V})^{\mathcal{L}}$ is given by the least fixed point of the equation $\vec{s} = F1(\vec{s})$ which can be calculated as the limit of the sequence $\vec{s}_{init}, F1(\vec{s}_{init}), \dots$.

Proposition 1. *The following assertions regarding sequence $\overrightarrow{s_{init}}, F1(\overrightarrow{s_{init}}), \dots$ hold: (1) $\forall i \in \mathbb{N}, F1^i(\overrightarrow{s_{init}}) \subseteq F1^{i+1}(\overrightarrow{s_{init}})$; (2) $\exists k \geq 0, F1^k(\overrightarrow{s_{init}}) = F1^{k+1}(\overrightarrow{s_{init}})$.*

Now, consider a Promela program $M = P_1 \parallel \dots \parallel P_n$ involving the concurrent execution of several processes. Let \mathbf{IA}_1^i be the vector produced by the *Influence Analysis* for the process P_i . If we denote with \mathcal{L}_i the set of labels appearing in process P_i , then a program point of M may be represented by a tuple (l_1, \dots, l_n) with $l_i \in \mathcal{L}_i$ being the current program counter of process P_i . Considering this, we define function $\mathbf{IA}_1 : \mathcal{L}_1 \times \dots \times \mathcal{L}_n \rightarrow \wp(\mathcal{V})$ as: $\mathbf{IA}_1(l_1, \dots, l_n) = \bigcup_{i=1}^n \mathbf{IA}_1^i(l_i)$. That is, the information regarding analysis \mathbf{IA}_1 at program counter (l_1, \dots, l_n) is the union of all variables collected by \mathbf{IA}_1 for each process P_i at label l_i .

Example. The following code is a simplified version of the example shown in Fig 3. Observe that when applying analysis \mathbf{IA}_1 to this code, variables become visible/unvisible following the rules given by function $F1^*$. For instance, `cResp` is needed before evaluating the test `CResp > 0`, but before assignment `cResp = WriteBuf` is not needed because its value is rewritten. The same rule is applicable to the rest of the variables.

```
do :: enter_loop ->
    ReadBuf = any();
    // ReadBuf is visible
    WriteBuf = ReadBuf;
    // WriteBuf is visible
    cResp = WriteBuf;
    // cResp is visible
    enter_loop = (cResp > 0);
    :: else -> break;
od;
```

4.2 Extending the Influence Analysis \mathbf{IA}_1

In this section, we propose several refinements of \mathbf{IA}_1 which are able to preserve more interesting temporal properties and to take global variables into account. Observe that the construction of \mathbf{IA}_1 is based on function $F1^*$, which propagates the information about the needed variables in a bottom-up manner, and on the initial vector $\overrightarrow{s_{init}}$ which is used to start the fixed point computation. The variants of \mathbf{IA}_1 presented below are constructed by modifying function $F1^*$ and by considering different initial vectors.

Preserving State Properties. The first extension \mathbf{IA}_2 preserves state properties specified using the `assert` statement. For instance, assume that we want to analyze the assertion `x1 == 2` of process $p2$ in the right-hand column of Figure 6. It is easy to see that we need to store not only the variables *influencing* the boolean expressions in the code in order to completely simulate the reachability tree, but also those that influence the variables in the `assert` statement (variable `x1` in the example). Figure 8 shows the intended result of \mathbf{IA}_2 for process $p2$. Observe that variable `x1` is attached to some labels of the process, since its value is needed at label `L12`. Thus, our purpose is to extend analysis \mathbf{IA}_1 to take into account variables appearing in the assertions to be validated in the code

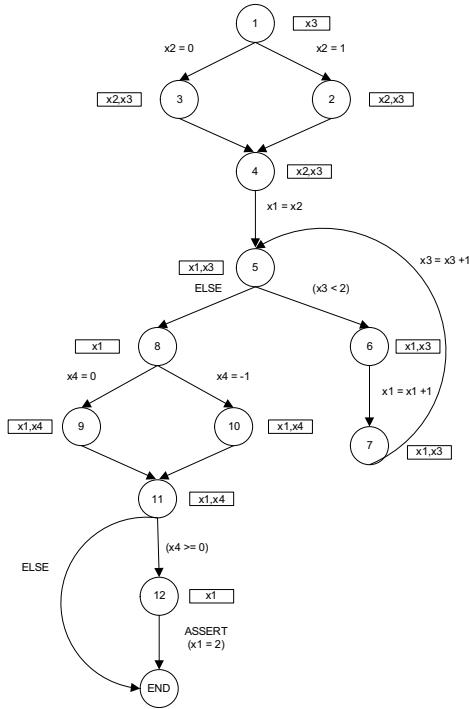


Fig. 8. Result of IA_2 for process p2

during execution. It is worth noting that at this point, we are still assuming the model only contains local variables. To extend IA_1 , it is enough to redefine $F1^*$ as function $F2^*$ defined as:

$$\begin{aligned}
 F2^*(x = exp, s) &= \begin{cases} s & \text{if } x \notin s \\ s - \{x\} \cup var(exp) & \text{if } x \in s \end{cases} \\
 F2^*(bool, s) &= s \cup var(bool), \text{ bool being a Boolean expression} \\
 F2^*(assert(b), s) &= s \cup var(b), \text{ assert}(b) \text{ being an assertion expression}
 \end{aligned}$$

Now, we construct IA_2 using $F2^*$ as IA_1 was defined from function $F1^*$, and considering the same initial vector $\overrightarrow{s_{init}}$. The resulting analysis is able to preserve the assertions as desired.

Dealing with Global Variables. As mentioned above, the previous description is only applicable to models without global variables. It is important to distinguish between global and local variables. Local variables are easier to analyze because their use is localized inside a unique process, and the static analysis follows the control flow of isolated processes. In contrast, the code regarding a global variable may be distributed through many different system processes. Thus, it is possible that some variables used to construct a given global variable in a process are erroneously hidden by the static *local* analysis. In order to solve

this problem, we consider the set $\mathcal{G}_M \subseteq \mathcal{V}$ of all global variables appearing in some boolean expression of some process. Now we modify $\overrightarrow{s_{init}}$ and use $\overrightarrow{s_{init}^g}$ defined as $\{s_l | l \in \mathcal{L}\}$ where $\forall l \in \mathcal{L}. s_l = \mathcal{G}_M$.

With this definition for the initial vector, static analysis is able to extract all variables influencing global variables which are critical for the control flow of the model. In the following, we call IA_3 to this extension. Observe that if we consider assertions as boolean expressions, this analysis is also able to preserve the state properties described above.

Preserving Trace Properties. Analysis IA_3 may be extended to IA_4 which is intended to preserve generic temporal properties. Assume that f is a LTL property. In order to preserve the evaluation of f we have to make all (global) variables of $\text{var}(f)$ *always* significant for the analysis. Thus, since variables appearing in the formula are always saved, the formula may always be correctly checked. Then, function $F4^*$ takes into account variables in the temporal formula f to be checked as follows:

$$F4^*(x = \text{exp}, s) = \begin{cases} s & \text{if } x \notin s \\ s - \{x\} \cup \text{var}(\text{exp}) & \text{if } x \in s, x \notin \text{var}(f) \\ s \cup \text{var}(\text{exp}) & \text{if } x \in s, x \in \text{var}(f) \end{cases}$$

$$F4^*(\text{bool}, s) = s \cup \text{var}(\text{bool}), \text{ bool being a Boolean expression}$$

Now, define $\overrightarrow{s'_{init}} = \{s'_l | l \in \mathcal{L}\}$ where $\forall l \in \mathcal{L}, s'_l = \mathcal{G}_M$. Note that, for practical reasons, we are assuming that all variables in f are global. Thus, analysis IA_4 is constructed following the approach presented in Section 4.1, but using function $F3^*$ and the initial vector $\overrightarrow{s'_{init}}$.

Note that, following the ideas in [11], we can improve the influence analysis by removing variables from the property f when they are not necessary. In that way we can adjust dynamically the visible variables to be taken into account.

5 Correctness Issues

In this section, we formalize the correctness of the static analysis developed in Section 4. We start establishing the semantics of a simplified version of Promela.

5.1 A Simplified Semantics for Promela

Assume that $M = P_1 || \dots || P_n$ is a Promela system constituted by the concurrent execution of processes $P_i (1 \leq i \leq n)$. If $Value$ represents the set of all possible values for the variables in \mathcal{V} , we define the set $Env = \mathcal{V} \rightarrow Value$ of all possible functions giving values to the elements of \mathcal{V} . In the sequel, we call environments to the elements of Env . Thus, given $e \in Env$ and $v \in \mathcal{V}$, $e(v)$ denotes the value given to v by the environment e . In addition, $e[n/v]$ denotes the environment that is equal to e for all variables except for v whose value is n .

Given a process P , we define the set of process states $State = \mathcal{L} \times Env$ as the set of pairs $\langle l, e \rangle$ where $l \in \mathcal{L}$ is the program counter of P and $e : \mathcal{V} \rightarrow Value \cup \perp$

BoolExp	$\frac{I(l) \in \text{BoolExp}, \text{eval}(I(l), e) = \text{true}}{\langle l, e \rangle \mapsto_P \langle \text{next}(l), e \rangle}$	Assign	$\frac{I(l) = x = \text{exp}, \text{eval}(\text{exp}, e) = n}{\langle l, e \rangle \mapsto_P \langle \text{next}(l), e[n/x] \rangle}$
Non-det	$\frac{I(l) = \text{if}:: b_1 \rightarrow l_1 \dots b_n \rightarrow l_n \dots ; fi, \text{eval}(b_i, e) = \text{true}}{\langle l, e \rangle \mapsto_P \langle l_i, e \rangle}$		
IntLeaving	$\frac{\langle l_i, e \rangle \mapsto_P \langle l'_i, e' \rangle}{\langle l_1, \dots, l_i, \dots, l_n, e \rangle \mapsto_S \langle l_1, \dots, l'_i, \dots, l_n, e' \rangle}$		

Fig. 9. Process-Level and System-Level Rules

is an environment restricted to the variables accessed by P . Thus, $v \in \mathcal{V}$ is given the special value \perp if P cannot access to it (v is local to a different process).

Figure 9 shows a simplified version of the complete semantics for Promela processes. We have included only the most relevant statements for the sake of simplicity. The whole semantics may be found in [5]. The *simplified* transition relation for the process level is defined as $\mapsto_P \subseteq \text{State} \times \text{State}$. In the figure, we use the function $\text{eval} : \text{Expr} \times \text{Env} \rightarrow \text{Value}$ to evaluate expressions, where Expr represents the set of all Boolean and arithmetical expressions that may be constructed with the usual operators and the constants and variables of M .

Given a Promela system $M = P_1 \parallel \dots \parallel P_n$, we define the set $\text{Conf} = \mathcal{L}^n \times \text{Env}$ of all global states of M . Thus, a configuration consists of the current program counter of each process in M and the global environment giving the current value to all model variables. Figure 9 also shows the system-level transition relation $\mapsto_S \subseteq \text{Conf} \times \text{Conf}$. This rule realizes the interleaving of the system processes in execution.

5.2 Correctness Results for IA₁

In this section, we prove the results showing the usefulness of our proposal. We start by giving some necessary definitions.

Definition 1. We say that variable $x \in \mathcal{V}$ is redefined at label $l \in \mathcal{L}$, and write it as $\text{redef}(x, l)$, iff $I(l) = x = \text{exp}$, that is, if x is given a new value at l .

Definition 2. Given $x \in \mathcal{V}$ and $l_{1i} \in \mathcal{L}$, we say that x is needed at l_{1i} for IA₁ and write it as $\text{needed}_1(x, l_{1i})$ iff there exists a finite path $\langle \dots, l_{1i}, \dots, e_1 \rangle \mapsto_S \dots \mapsto_S \langle \dots, l_{ki}, \dots, e_k \rangle$ such that $\forall 1 \leq j \leq k. \neg \text{redef}(x, l_{ji})$ and it holds that

1. $I(l_{ki}) \in \text{BoolExp}$ and $x \in \text{var}(I(l_{ki}))$
2. $I(l_{ki}) = y = \text{exp}$, $x \in \text{var}(\text{exp})$ and $\text{needed}_1(y, l_{ki})$

That is, variable x is needed by the analysis IA₁ at a given program counter if its current value is used in some boolean expression (case 1) or it is used to calculate some variable further needed by the static analysis (case 2).

Proposition 2 proves that IA₁ attaches each label with all the variables needed at this point.

Proposition 2. *Given $x \in \mathcal{V}$ and $l \in \mathcal{L}$, if $\text{needed}_1(x, l)$ then $x \in \text{IA}_1(l)$.*

Once we have defined the notion of variable needed at a program location, we may formalize the variables that should be stored at program labels.

Definition 3. *Consider $l \in \mathcal{L}$ such that $I(l)$ is a basic instruction (a Boolean expression or an assignment) or a non-deterministic selection. Then, we define the set $\text{nvar}(l) = \{x \in \mathcal{V} \mid \text{needed}_1(x, l)\}$.*

The following proposition proves that the *Influence Analysis* associates each process label with the variables needed (wrt the previous definition) to execute the following instruction.

Proposition 3. *Let P be a Promela process and consider the static analysis for the code reachability tree IA_1^P given in Section 4.1. Let $l \in \mathcal{L}$ be a label of P then $\text{nvar}(l) \subseteq \text{IA}_1^P(l)$.*

We may extend Proposition 3 to Promela systems as follows:

Corollary 1. *Let $M = P_1 \parallel \dots \parallel P_n$ be a Promela system and consider the static analysis for the code reachability tree IA_1 given in Section 4.1. Let $(l_1, \dots, l_n) \in \mathcal{L}^n$ be a program counter of M then $\cup_{i=1}^n \text{nvar}(I(l_i)) \subseteq \text{IA}_1(l_1, \dots, l_n)$.*

Given $V \subset \mathcal{V}$, we define the *equivalence relation* $\sim_V \subseteq \text{Env} \times \text{Env}$ as follows: $e_1 \sim_V e_2 \iff \forall v \in V. e_1(v) = e_2(v)$

Proposition 4. *Consider two environments $e_1, e'_1 \in \text{Env}$ and two labels $l, l' \in \mathcal{L}$ such $\langle l, e_1 \rangle \mapsto_P \langle l', e'_1 \rangle$. Then if $e_1 \sim_{\text{IA}_1(l)} e_2$, there exists $e'_2 \in \text{Env}$ such that $\langle l, e_2 \rangle \mapsto_P \langle l', e'_2 \rangle$ and $e'_1 \sim_{\text{IA}_1(l')} e'_2$.*

We may extend the previous proposition to system configurations as follows.

Corollary 2. *Consider two environments $e_1, e'_1 \in \text{Env}$ and two labels $l_i, l'_i \in \mathcal{L}$ such $\langle l_1, \dots, l_i, \dots, l_n, e_1 \rangle \mapsto_S \langle l_1, \dots, l'_i, \dots, l_n, e'_1 \rangle$. Define $\vec{l} = (l_1, \dots, l_i, \dots, l_n)$ and $\vec{l}' = (l_1, \dots, l'_i, \dots, l_n)$. Then if $e_1 \sim_{\text{IA}_1(\vec{l})} e_2$, there exists $e'_2 \in \text{Env}$ such that $\langle l_1, \dots, l_i, \dots, l_n, e_2 \rangle \mapsto_S \langle l_1, \dots, l'_i, \dots, l_n, e'_2 \rangle$ and $e'_1 \sim_{\text{IA}_1(\vec{l}')} e'_2$.*

The following theorem gives us the desired correctness result for analysis IA_1 .

Theorem 1. *Assume that $\langle l_{11}, \dots, l_{n1}, e_1 \rangle \mapsto_S \dots \mapsto_S \langle l_{1k}, \dots, l_{nk}, e_k \rangle$ is a finite path. For all $i \leq k$, denote $\vec{l}_i = (l_{1i}, \dots, l_{ni})$. Then, if $e'_1 \in \text{Env}$ satisfies that $e_1 \sim_{\text{IA}_1(\vec{l}_1)} e'_1$, then there exists a finite path $\langle l_{11}, \dots, l_{n1}, e'_1 \rangle \mapsto_S \dots \mapsto_S \langle l_{1k}, \dots, l_{nk}, e'_k \rangle$ such that $\forall 1 < j \leq k. e_j \sim_{\text{IA}_1(\vec{l}_j)} e'_j$.*

Observe that following Theorem 1, the reachability tree may be pruned. Every new state with visible variables matching one state previously stored is considered as a visited state. The proof of the preservation of properties between the original and the reduced state spaces should take into account the search algorithm, for instance, the algorithm in [1].

5.3 Correctness Results for \mathbf{IA}_2 , \mathbf{IA}_3 and \mathbf{IA}_4

In this section, we establish the main results proving the correctness of analysis \mathbf{IA}_2 , \mathbf{IA}_3 and \mathbf{IA}_4 . The proofs of the corresponding theorems are similar to the ones given for \mathbf{IA}_1 in the previous section. Observe that the correctness result for \mathbf{IA}_3 does not appear because it is similar to Theorem 1.

Theorem 2 (Correctness for \mathbf{IA}_2). *Assume that $\langle l_{11}, \dots, l_{n1}, e_1 \rangle \mapsto_S \dots \mapsto_S \langle l_{1k}, \dots, l_{nk}, e_k \rangle$ is a finite path. In addition, assume that there exists an index j such that $I(l_{jk}) = \text{assert}(b)$ for some boolean expression b . For all $i \leq k$, denote $\vec{l}_i = (l_{1i}, \dots, l_{ni})$. Then, if $e'_1 \in \text{Env}$ satisfies that $e_1 \sim_{\mathbf{IA}_2(\vec{l}_1)} e'_1$, then there exists a finite path $\langle l_{11}, \dots, l_{n1}, e'_1 \rangle \mapsto_S \dots \mapsto_S \langle l_{1k}, \dots, l_{nk}, e'_k \rangle$ such that $\forall 1 < j \leq k. e_j \sim_{\mathbf{IA}_2(\vec{l}_j)} e'_j$ and $\text{eval}(b, e_k) = \text{eval}(b, e'_k)$.*

Theorem 3 (Correctness for \mathbf{IA}_4). *Let f be an LTL formula. Assume that $\sigma = \langle l_{11}, \dots, l_{n1}, e_1 \rangle \mapsto_S \langle l_{12}, \dots, l_{n2}, e_2 \rangle \mapsto_S \dots$ is an infinite path from $\langle l_{11}, \dots, l_{n1}, e_1 \rangle$. For all $i \geq 1$, denote $\vec{l}_i = (l_{1i}, \dots, l_{ni})$. Then, if $e'_1 \in \text{Env}$ satisfies that $e_1 \sim_{\mathbf{IA}_4(\vec{l}_1)} e'_1$, then there exists a path $\sigma' = \langle l_{11}, \dots, l_{n1}, e'_1 \rangle \mapsto_S \langle l_{12}, \dots, l_{n2}, e'_2 \rangle \mapsto_S \dots$ such that $\forall 1 < j. e_j \sim_{\mathbf{IA}_4(\vec{l}_j)} e'_j$ and $\sigma \models f \iff \sigma' \models f$, \models being the standard satisfiability relation defined for evaluating LTL formulas on execution traces.*

6 Conclusions and Future Work

State space explosion in explicit model checking can be partially solved with techniques which change the usual algorithm to identify visited states. Instead of comparing every new global state with the states in the heap, abstract matching is able to compare only parts of the new state. In that way, it is possible to cut some execution paths and reduce the time and memory required to check a particular property. However, the results are only reliable when the abstraction method has been proved to be sound.

In this paper, we have presented the theoretical framework to ensure that static analysis can provide enough information to construct sound abstract functions for a given property. Furthermore, we provide evidence that, in the context of model extraction for **Spin**, these functions can be automatically produced and included in the final model.

We have obtained the experimental results with the tool **SOCKETMC**, although static analysis was still done by hand. At the moment, we are implementing these static analysis algorithms as an extension to **SOCKETMC**. Future work is oriented to integrate the new version of **SOCKETMC** with our tool for data abstraction αSpin , in such a way that we can make more efficient model checking of C programs.

References

1. D. Bosnacki. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Eindhoven Univ. of Technology, 2001.
2. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Int. conference on Software engineering*, pages 439–448, 2000. ACM Press.
3. P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model Checking Software with Well-Defined APIs: the Socket Case. In *FMICS '05: Proc. of the 10th int. workshop on Formal methods for industrial critical systems*, pages 17–26, 2005. ACM Press.
4. E.M. Clarke, H. Grumberg, and D. Peled. *Model Checking*. 2000.
5. M.M Gallardo, P. Merino, and E. Pimentel. A Generalized Semantics of Promela for Abstract Model Checking. *Formal Aspects of Computing*, 16:166–193, 2004.
6. K. Havelund and T. Pressburger. Model Checking Java Programs using Java Pathfinder. *International Journal of Software Tools for Technology Transfer*, 2(4):366–381, 2000.
7. G. J. Holzmann and R. Joshi. Model-Driven Software Verification. In *SPIN*, pages 76–91, 2004.
8. G. J. Holzmann and M. H. Smith. Software Model Checking: Extracting Verification Models from Source Code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.
9. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. 1998.
10. C. S. Pasareanu, R. Pelánek, and W. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *CAV*, pages 52–66, 2005.
11. D. Peled, A. Valmari, and I. Kokkarinen. Relaxed Visibility Enhances Partial Order Reduction. *Formal Methods in System Design*, 19(3):275–289, 2001.