αSPIN: Extending SPIN with Abstraction

Maria del Mar Gallardo, Jesus Martinez, Pedro Merino, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computacion
University of Malaga, 29071 Malaga, Spain
e-mail: {gallardo, jmcruz, pedro, ernesto}@lcc.uma.es

Abstract. Abstraction methods have become one of the most interesting topics in the automatic verification of software systems because they can reduce the state space to be explored and allow the analysis of more complex systems. One direction for abstracting a system is to transform its formal description (its model) into a simpler version specified in the same language, thus skipping the construction of a specific (model checking) tool for the abstract model. This paper presents the details of αSPIN, an XML-based tool to abstract PROMELA models in order to employ SPIN as an abstract model checking tool. αSPIN is built on the top of SPIN which allows us to take advantage of its current and future improvements for the verification of abstract models. Although the theoretical basis has been previously presented, many aspects related to correctness of the abstraction process are also included in this work to make the paper self-contained.

1 Introduction

Computer based verification methods, such as model checking [1, 4], have become realistic techniques to be used in the software development cycle. However, effective verification is only possible and fruitful if *useful formal models* of the systems are available. A useful model is an abstract representation of the real system, containing only the details necessary to ensure that satisfaction (non-satisfaction) of interesting properties in the model gives information about the behavior of the real system. Excessive model details may produce the well-known state explosion problem, which could prevent the use of current tools to fully analyze the system. This problem affects both the (classic) explicit model checking method and the more recent symbolic approach [22], and both of them employ ideas of abstract interpretation [6] to construct more abstract state spaces or models [3, 20, 7].

The use of abstract interpretation in symbolic model checking has been based mainly on predicate abstractions [14], and it usually consists in the automatic generation of an Ordered Binary Decision Diagram (BDD) that represents the abstract state space. When the verification results produce (abstract) counterexamples, some refinement method can be employed to construct a more precise abstraction [2, 23].

From our point of view, in explicit model checking the abstract state space should not be constructed prior to verification, so it must be constructed on-the-fly. Furthermore, it seems a good practice to employ the same language for the original (concrete) and the abstract models, in such a way that the same tool can be employed to verify both systems and the results can be more easily related. Many authors consider explicit model checking very suitable for software systems. For this reason, it is convenient to focus efforts in obtaining automatic methods to exploit abstraction in this context.

This paper presents an approach to extend explicit model checkers with abstraction capabilities. Although the approach can be applied to different tools, we present αSPIN, a tool to introduce abstraction in the model checker SPIN [16, 17]. αSPIN is based on using abstract interpretation to obtain more abstract models by automatic syntactic transformation of PROMELA. The inclusion of automatic abstraction in explicit model checking must improve some of the classical steps enumerated by Clarke et al. in their initial proposal [3]: a) defining one abstraction function α suitable for the temporal property to be verified, b) the construction of the abstract model (or the abstract state space) and c) relating the verification results to the behavior of the initial (concrete) model.

As regards step a), we propose the use of an abstraction library with previously defined functions that can be selected by the user depending on the property to be analyzed. That is, α should preserve the relevant information for this property and eliminate the non-relevant information [10]. Function α defines the data approximation and the abstraction of the simplest language operations' behaviour. Although α is selected by the user from a repository of well defined functions, αSPIN includes strategies to assist the users in the selection (mainly by using static analysis to find out information from the model and the property).

Our method to construct the abstract model (step b) is based on working on XML in order to be as independent as possible of the actual modelling language, so that the technique can be applied to other model checkers. Using XML also allows us to take advantage of currently available tools and APIs to process XML documents [11].

Finally, with respect to step c), we introduce the notion of non-standard verification to relate the results in the abstract and the concrete model [13]. The abstract model allows us the efficient verification of absence of deadlock and temporal properties encoded with Linear Time propositional temporal Logic (LTL) [21].

There are other tools that add some kind of abstraction to SPIN, but they mainly follow the *model extraction* approach. This method consists in producing a high level model from the source code (C, Java) to be analyzed with some existing model checking tool like SPIN, as implemented in Feaver [18], Bandera [5] or JPF [15] (first version). αSPIN may be considered as complementary to these tools for two reasons. First, αSPIN could be employed in the first designs of concurrent software or protocols whereas Feaver, Bandera and JPF are more suitable when the final code has been written. Secondly, given that PROMELA is the target notation in these tools, our abstraction method provides an additional

way to optimize the verification of software systems following the model extraction approach. Maybe BANDERA toolset is the closest to our proposal. However, we are mainly interested in the verification of LTL formulas. Additionally, our contribution is the use of XML to represent the whole library and the precise definition of the correctness conditions to be held by the functions before being stored in the library.

αSPIN has been implemented on top of SPIN, which allows us to take advantage of its current and future improvements for the verification of abstract models. The result is that αSPIN maintains compatibility with future versions of PROMELA and SPIN and also with current extensions such as PSPIN(for parallel verification), dSPIN(for PROMELA with dynamic structures) and XSPIN/Project (for management of verification results) which were presented in [8]. Documentation and current and future versions of αSPIN can be found at [25].

The paper is organized as follows. Section 2 contains some preliminary background on the main topics of the paper. Section 3 gives some methodological aspects about the use of the tool. Section 4 presents the theoretical basis to support correct abstraction by transformation of PROMELA. Some of the concepts were previously presented in [10]. In Section 5, we give details on the use of XML to support automatic constructions of the models. Section 6 contains some implementation details of αSPIN and an explanatory application. In Section 7, we discuss the main contributions of the work and Section 8 presents conclusions.

2 Preliminaries

In this section, we give some details about PROMELA, LTL, SPIN and XML.

2.1 PROMELA, LTL and SPIN

In the last few years, SPIN has become one of the most employed model checkers in both academic and industrial areas. It supports the verification of usual safety properties (like deadlock absence) in systems written in the modelling language PROMELA as well as the analysis of complex requirements (regarding the evolution of the system) expressed with Linear Temporal Logic (LTL). By default, given a LTL formula, SPIN translates it into an automata that represents an undesirable behavior (which is claimed to be impossible). Then, verification consists in an exhaustive exploration of the state space searching for executions that satisfy the automata. If such an execution exists, then the tool reports it as a counterexample for the property. If the model is explored and a counterexample is not found, then the model satisfies the LTL property (all possible execution branches satisfy the property). Properties that are satisfied in all executions are called *universal properties*.

PROMELA is a modelling language designed for describing systems composed of concurrent asynchronous communicating processes (such as the software for distributed systems). A PROMELA model $P = Proc_1 || \ldots || Proc_n$ consists of a finite set of concurrent processes, global and local channels, and global and

local variables. Processes communicate via message passing through channels. Communication may be asynchronous using channels as bounded buffers, and synchronous using channels with size zero. Global channels and variables determine the environment in which processes run, while local channels and variables establish the internal local state of processes.

PROMELA is a non-deterministic language that borrows some concepts and syntax elements from Dijkstra's guarded command language, Hoare's CSP language and C programming language. A PROMELA process is defined as a sequence of possibly labelled sentences preceded by the declarative part. *Basic* sentences in PROMELA are those that produce a definite effect over the model state; in other words, the assignments, the instructions for sending (receiving) messages to (from) channels and the Boolean expressions, $BExp$, that include tests over variables and contents of channels. In addition, PROMELA has other non-basic sentences like the non-deterministic *If* and *Do* sentences. Initialization of models may be defined by the *init* process.

SPIN verifies LTL formulas against PROMELA models. Well-formed LTL formulas are inductively constructed from a set of atomic propositions (in PROMELA, propositions are tests over data, channels or labels), the standard Boolean operators, and the *temporal operators*: *always* "$\Box$", *eventually* "$\Diamond$" and *until* "$\mathtt{U}$". Formulas are interpreted with respect to traces $t_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. Each trace expresses a possible model execution from state s_i. The use of *temporal operators* permits construction of formulas that depend on the current and future states of a configuration sequence. The semantics of LTL is shown in Fig. 1.

$t_i \models p$ iff $s_i \models p$, p being a proposition
$t_i \models \Box p$ iff $\forall j \geq i . t_j \models p$, p being a temporal formula
$t_i \models \Diamond p$ iff $\exists j \geq i . t_j \models p$, p being a temporal formula
$t_i \models p \ \mathtt{U} \ q$ iff $\exists k \geq i . \forall j . i \leq j < k . t_j \models p, t_k \models q$, p and q being temporal formulas

Fig. 1. LTL semantics

2.2 XML and related tools

W3C introduced XML [27] as a way to format and manage formal documents. XML offers several interesting properties: it is straightforwardly usable over Internet, it supports a wide variety of applications, and there exist tools to easily write programs which process XML documents.

XML is used to create a particular set of tags used for coding specific information. The basic XML building block is known as entity. An entity contains data which may be syntactically analyzed. Analyzed data use tags for describing the storage structure of a document and its logical organization. Element tags represent the most used components. Elements start with an identifier tag <*element*>, and end with a closing tag </*element*>. Elements may contain mixed

data, consisting of character data and/or some other elements, thus composing a hierarchical tagged structure.

In relation to the XML specification, a number of tools and related standards have been created. Some of them are intended to extend the XML basic functionality, like XPath [26] (a language for addressing parts of an XML document) or XSLT (eXtensible Stylesheet Language for Transformations) [29]. There are also APIs like SAX or DOM [28] that can be used when reading and manipulating XML documents. SAX is a document-driven programming library that supports an event-handler register for the treatment of tags. DOM allows us to convert a document into a tree-shaped structure in memory, where new elements can be easily replaced and added. JDOM is a library, written in Java, which includes the functionality of SAX and DOM and which is in the process of being adopted as part of the Java core platform [19].

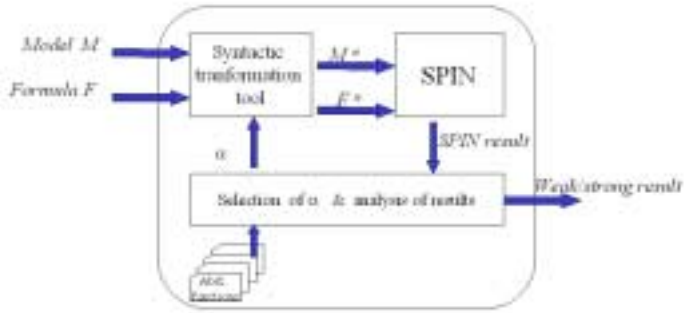

Fig. 2. αSPIN architecture

3 Methodology

In this section, we briefly present some methodological notes on the use of αSPIN. The main components of our current project are shown in Fig. 2.

The usual method for proving the correctness of a model consists of successively proving certain temporal formulas that represent the safety and liveness properties that the model must hold. In the first stages of the system analysis, when the properties to be proved are universal (i.e. they must hold over all execution traces), errors usually are quickly detected, that is, it is not necessary to analyze many states in order to find an execution trace over which the current

property does not hold. In this case, the tool provides user with a counterexample which may be used to improve the model. However, a problem arises when the tool can neither prove nor disprove a temporal property due to the state-explosion problem. At this moment, user may make use of the abstraction tool as next explained.

Our idea is to iteratively alternate between the verification and the abstraction phases as follows. We begin with the verification phase. If the system (SPIN) cannot verify the satisfaction (or the non-satisfaction) of a property F over the original model M because M is too large to be handled, then the abstraction phase is started. At this point, the user must supply an abstraction function α to guide the transformation of M into a more abstract model M^α. We assume here that α represents both the abstraction of data and the modification of the basic operations in the language. The abstraction function α may be provided from a library or newly defined.

Property F may also need to be transformed into an abstract property F^α when the model is transformed. Once M^α and F^α have been constructed, we pass again to the verification phase. When transforming the property, we introduce a weak satisfaction relation which includes the loss of precision due to the abstraction process. Since the execution of M^α usually involves more behaviours than really possible, the counterexample provided by the tool, when the abstract property does not hold over the model, has to be carefully analyzed. The analysis of spurious counterexamples is one of our current work. When the abstract property holds, the user can discard very undesirable behaviours, and he gains confidence in the correctness of the model. In addition, it also possible to analyze a more precise version of the abstract property by a refinement process. The refinement consists of replacing the definitions of propositions in the formula by more accurate ones. We repeat the process with a different α until obtaining a feasible counterexample or some information about the satisfaction of the original property F against the original model M. Usually, the new α is a more precise version of the previous one. In Section 4.2, we summarize the theoretical results that support the methodology here presented.

4 Abstracting PROMELA

In this section, we give an overview of the proposal in [10, 12]. In these works, we present a theoretical framework to automatically construct more abstract models by syntactic transformation. For this purpose, we define a *generalized structured operational semantics* for PROMELA. The selection of this semantics may be understood by noting that many language aspects (like the interaction among processes and the execution of non basic sentences) are not modified during the abstraction process. Thus, it seems natural to use a semantics in which these aspects are separated from the ones which may be changed by abstraction (the data interpretation and the execution of basic sentences).

Thus if $State$ is the set of model states, and $effect : Inst \times State \to State$ and $test : BExp \times State \to \{false, true\}$ are two functions describing the ef-

fect of executing a basic sentence and a test in a given state, respectively, then the semantic function $Gen(-, \mathit{effect}, \mathit{test})$: PROMELA $\rightarrow \wp(Trace)$ associates each model M with the set of traces $Gen(M, \mathit{effect}, \mathit{test}^\alpha)$, where $Trace = \cup_{\{i \geq 0\}} State_i \bigcup State^w$, $State_i = State \times \overset{i}{\ldots} \times State$ and $State^w$ is the set of infinite sequence of states. Each trace represents a model simulation which uses functions *effect* and *test* when executing a basic sentence or a Boolean expression. Basic sentences are assignments, operations over channels and so.

The advantage of this representation is the ability to describe and relate different model behaviors. For instance, the usual/standard behavior of a model M is given by $STD(M) = Gen(M, \mathit{effect}^P, \mathit{test}^P)$, where effect^P and test^P are the functions defining the standard behavior of PROMELA basic sentences.

In accordance with these considerations, given $Gen(M, \mathit{effect}, \mathit{test})$ an interpretation of a model M, the use of the abstract interpretation technique to reduce the model size consists in defining a reduced set of states $State^\alpha$ by means of an abstraction function $\alpha : State \rightarrow State^\alpha$ and two functions $\mathit{effect}^\alpha : Basic \times State^\alpha \rightarrow State^\alpha$ and $test^\alpha : BExp \times State^\alpha \rightarrow \{false, true\}$, giving the proper meaning to the basic PROMELA sentences. Given the previous discussion, $Gen(M, \mathit{effect}^\alpha, \mathit{test}^\alpha)$ defines a non-standard (abstract) behavior of M.

```
proctype Lift(int pid){
    int Order=null;
    do
    :: SysLift_Lift[pid]?Order;
       if
       :: (Order==Up) ->
           Position[pid]=Position[pid]+1;
       :: (Order==Down) ->
           Position[pid]=Position[pid]-1;
       ...........
}
```

Fig. 3. Lift system model

From the point of view of the applicability of abstraction techniques, the generalized semantics allows us to isolate the key points (effect, test) which are affected by abstraction, independently of the complexity of language constructions. This facilitates the definition of abstractions, the analysis of correctness and preservation results, and even the implementation (applying an abstraction can be computer supported by only defining two mappings, which can be described by means of macros).

For instance, Fig. 3 shows an excerpt of a PROMELA model that represents the behavior of a lift (extracted from [9]). In order to simplify the exposition, we assume that system states are given by the value of the variable `Position[pid]` that is an integer number between the values $0..\mathit{nb\_floor} - 1$. Variable `Position[pid]` always stores the current floor for the lift identified by `pid`. To reduce the model size, consider the poset $(\texttt{FLOORS}, \leq^\alpha)$ illustrated in Fig. 4 and the abstraction function $\alpha : [0..\mathit{nb\_floor} - 1] \rightarrow \texttt{FLOORS}$ defined as

$\alpha(0) = lower, \alpha(nb\_floor - 1) = upper$ and $\forall 0 < j < nb\_floor - 1, \alpha(j) = middle$. The use of the partial order $\leq^\alpha$ allows us to include the notion of approximation in the abstract domain `FLOORS`: the abstract value *noUpper* approximates any floor different from the *upper* one, thus *noUpper* is an abstract value less precise than both *lower* and *middle*. Value *unknown* is the least precise abstract data since it represents any floor.

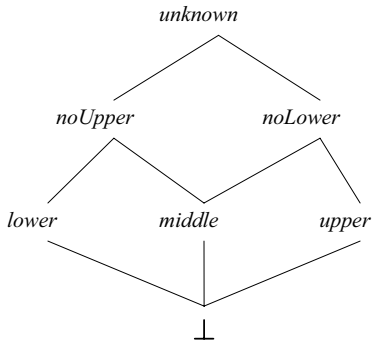

Fig. 4. The lattice for `FLOORS`

The redefinition of states involves the redefinition of the effect of basic sentences. The table in Fig. 5 shows a possible definition of the abstract effect of the instruction `i = i + 1` in the abstract state s^α, the abstract state *Lower* being the state in which variable i has the value *lower* and so on. Fig. 5 also presents a possible definition of the abstract test $i == 0$.

s^α	$effect^\alpha(i = i + 1, s^\alpha)$	$test^\alpha(i == 0, s^\alpha)$
Lower	*Middle*	*true*
Middle	*NoLower*	*false*
Upper	$\bot$	*false*
NoLower	*NoLower*	*false*
NoUpper	*NoLower*	*true*
Unknown	*Unknown*	*true*

Fig. 5. Part of the abstract effect and test

4.1 Correctness

Given α an abstraction function, it is clear that functions effect^α and test^α can be arbitrarily defined. However the interest of the approach is in preserving some correction properties between $STD(M) = Gen(M, \mathit{effect}^P, \mathit{test}^P)$ and $STD^\alpha(M) = Gen(M, \mathit{effect}^\alpha, \mathit{test}^\alpha)$. In [10, 12] there is an exhaustive study of the correctness conditions that test^α and effect^α must verify for $STD^\alpha(M)$ to be a correct approximation of $STD(M)$. Partial definitions of functions effect^α and test^α given in Fig. 5 verify these conditions.

Correctness conditions guarantee that the reduced/abstract model correctly simulates the original one in the sense that for each non-deadlocked trace $t = s_0 \rightarrow s_1 \rightarrow \ldots$ in $STD(M)$ there exists a non-deadlocked abstract trace $t^\alpha = s_0^\alpha \rightarrow s_1^\alpha \rightarrow \ldots$ in $STD^\alpha(M)$ that approximates it, which is denoted by $\alpha(t) \leq^\alpha t^\alpha$. Trace simulation means that for all i, state s_i^α approximates s_i, i. e., $\alpha(s_i) \leq^\alpha s_i^\alpha$. Note that we explicitly exclude deadlocked traces because the abstraction process may modify this safety property of the system.

4.2 Practical results

The previous semantic framework allows us to relate the original and the abstract models to obtain certain practical results concerning the satisfaction of universal temporal formulas and the absence of deadlock.

In order to analyze deadlock absence preservation, we must impose some conditions, called *executability* conditions, on the definition of test^α. In PROMELA, Boolean conditions are guards in the sense that the system deadlocks when a control point, in which all conditions are false, is reached. Executability conditions assure that abstraction does not modify the evaluation of the Boolean expressions which can provoke the system deadlock. Under executability conditions, the following theorem holds.

Theorem 1. *$STD^\alpha(M)$ has no execution trace which deadlocks if, and only if, $STD(M)$ has no deadlock either.*

Atomic propositions in LTL formulas regarding PROMELA models are Boolean expressions; thus in accordance with the previous discussion we can consider different notions of formula satisfiability, depending on the underlying model interpretation. Given $I(M) = Gen(M, \mathit{effect}, \mathit{test})$ an interpretation of the model M, we say that a state $s \in \mathit{State}$ verifies a proposition p under I, and write $s \models_I p$ iff $\mathit{test}(p, s)$ holds. This definition may be easily extended to universal temporal formulas and traces, and so we can say that model M verifies the universal temporal formula F under the interpretation I and write $I(M) \models_I F$, iff for all $t \in I(M).t \models_I F$. Given $STD(M)$ the standard interpretation of a model M and $STD^\alpha(M) = Gen(M, \mathit{effect}^\alpha, \mathit{test}^\alpha)$ an abstract interpretation that is correct with respect to $STD(M)$, we say that $t \in STD(M)$ verifies the universal temporal formula F under STD^α and write $STD(M) \models_\alpha F$, iff for all $t \in STD(M)$ there exists $t^\alpha \in STD^\alpha(M)$ such that $\alpha(t) \leq^\alpha t^\alpha$ and $t^\alpha \models_\alpha F$. We write $\models$ and $\models_\alpha$ for $\models_{STD}$ and $\models_{STD^\alpha}$, respectively. With these definitions, the

following theorem that relates satisfaction of universal temporal formulas over different model interpretations holds.

Theorem 2. *Assume that $STD(M)$ is a deadlock-free interpretation of a model M and that $STD^{\alpha}(M)$ is an abstract interpretation correct wrt $STD(M)$. If F is a universal property and $STD^{\alpha}(M) \models_{\alpha} F$, then $STD(M) \models_{\alpha} F$.*

The first result can be directly employed to discard deadlock in the original model by analyzing the abstract (simpler) model. The second result says that if the abstract model satisfies the abstract interpretation of the formula F, then original models also satisfies the abstract interpretation of F. We call this result "weak preservation" because the abstraction process may involve some loss of information when analyzing temporal formulas. The next theorem studies when temporal formulas are "strongly preserved".

Theorem 3. *Assume that $STD(M)$ is a deadlock-free interpretation of a model M and that $STD^{\alpha}(M)$ is an abstract interpretation correct wrt $STD(M)$. If F is a universal property such that $\gamma(\alpha(F)) = F$ and $STD^{\alpha}(M) \models_{\alpha} F$, then $STD(M) \models F$.*

Informally, condition $\gamma(\alpha(F)) = F$ means that the abstraction process has not modified the meaning of the formula to be analyzed. In [12, 13], we study when abstractions satisfy such a condition and also how to obtain different precision degrees when analyzing abstract formulas by means of a process of refinement.

```
inline INC(v){
 if
 ::(v==LOWER)->    v=MIDDLE;
 ::(v==MIDDLE)->   v=NO_LOWER;
 ::(v==NO_UPPER)-> v=NO_LOWER;
 ::(v==NO_LOWER)-> v=NO_LOWER;
 ::(v==UPPER || v==ILLEGAL)-> v=ILLEGAL;
                              assert(0);
 fi
}
```

```
proctype Lift(int pid) {
   int Order=null;
   do
   :: SysLift_Lift[pid]?Order;
      if
      :: (Order==Up) ->
         INC(Position[pid]);

          ...}
```

Fig. 6. Partial implementation of $effect^{\alpha}$

4.3 Implementation approach

We have addressed the implementation problem by means of source-to-source transformation. The method is based on replacing each instruction in M by a standard PROMELA code that implements $test^{\alpha}$ and $effect^{\alpha}$ in order to obtain a new model M^{α}. Then the verifier is produced for M^{α} and the verification is carried out by only executing standard instructions. This approach corresponds to implementing a verifier for $Gen(M^{\alpha}, effect^{P}, test^{P})$.

For instance, Fig. 6 shows `INC`, a possible implementation of abstract increment `i = i + 1` defined in Fig. 5. The code in the second column of Fig. 6 illustrates the transformation of the original code into the abstract one.

Similarly, temporal formulas are transformed using the abstract version $test^{\alpha}$ of function *test*.

Although the transformation method is clearly motivated in [10], in this previous paper, the problem of implementation was not completely solved. The rest of the paper presents one implementation of this method using XML.

5 XML descriptions

We consider XML as the unique internal representation to perform the abstraction by transformation as shown in Fig. 7. We think that the intermediate XML structure may be used to apply our abstraction method to other modelling languages. The actual modelling language can be translated into this representation by a front-end module (steps 1 and 2 in Fig. 7) and the final abstracted model for the model checker can be produced by a specific back-end module (steps 6 and 7). Furthermore, if we use the same internal notation for both models and abstraction functions, we can concentrate efforts in developing reusable techniques and uniform tools for transformation based abstraction.

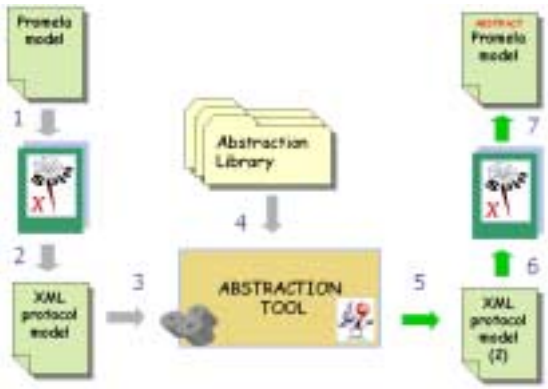

Fig. 7. Architecture of XML based abstraction

In addition to the practical reasons like the use of browsers and other user-friendly presentation tools, the development of XML oriented tools is supported by a number of more technical reasons. As every model checker uses a particular input, from the point of view of the modelling language, each one has a specific parser and additional support to convert the model specification into a suitable internal data structure for the model checking phase. Unfortunately, it is not a common practice to have access to this internal representation, because model

checking tools are source-closed or not flexible enough for implementing data transformation or manipulation via a set of APIs, as required in abstraction. Even in cases of open source projects like SPIN, most of the work to perform abstraction cannot be directly reused for other model checkers. In addition, the XML representation of the model facilitates traditional tasks in abstraction tools, such as finding relationships among variables, locating the points where a particular variable is employed, etc.

As regards abstraction functions, XML is a powerful means to represent the mapping between concrete and abstract data and abstract operations, including details such as the type of the operands, precedence rules, etc. The language also offers the advantage of being a suitable notation to define the whole abstraction library to contain all abstraction functions as a repository.

Fig. 7 shows the whole process of applying an abstraction function to a variable (or a group of them) declared in a PROMELA model. The first step is carried out by transforming the original model into an XML document suitable for external manipulation. Then, the abstraction tool produces another XML document using the abstraction function in the library to conduct the transformation. Finally, the abstract PROMELA version is obtained with the back-end module. Most of the work in this process consists in defining abstraction functions and transforming the XML documents, and it can be reused to construct specific abstraction tools for other model checkers and modelling languages.

5.1 DTD for Promela

The structure of a valid XML document is supplied by a DTD (Document Type Definition) file. A DTD specifies the kind of tags that can be included in an XML document, and the valid arrangements of these tags. Although it is not necessary, a DTD file may help to verify that an XML document is well formed, i.e., that it respects the full PROMELA grammar, which represents a prerequisite for the subsequent abstraction process. Thus, a major task is defining a tagged language based on XML that can represent the PROMELA models. For this purpose, a vocabulary of tags and a DTD have been modelled.

```
<!ELEMENT model (declaration | typedef | process | init | never)+>
<!ELEMENT process (parameters?, priority?, precondition?, body)>
<!ATTLIST process instances CDATA #IMPLIED
                  name CDATA #REQUIRED>
<!ELEMENT priority (#PCDATA)>
<!ELEMENT parameters (declaration+)>
<!ELEMENT declaration (var+ | enumtype)>
<!ATTLIST declaration visibility (HIDDEN | SHOW | ISLOCAL) #IMPLIED
                      type (BIT | BOOL | BYTE | SHORT | INT | CHANNEL | UNSIGNED) #REQUIRED>
<!ELEMENT var (messages)?>
<!ATTLIST var name CDATA #REQUIRED
              value CDATA #IMPLIED
              array_elements CDATA #IMPLIED>
```

Fig. 8. Part of PROMELA DTD

Fig. 8 shows part of the file *promela.dtd* used as a template for building valid XML models. A DTD is built following EBNF notation: each element (!ELEMENT) has a correspondence with an XML tag. Attributes (!ATTLIST) refine information for a tag. In the figure, the first tag encountered is <*model*>, that may contain other nested tags, representing processes, declarations of global variables, new type definitions, startup scripts, etc. Fig. 8 also shows elements as <*declaration*>, <*process*> and <*var*>. A declaration may contain information about `visibility` of the variables included within. For the validation phase, a HIDDEN variable is not referenced in the state vector. For a simulation, SHOW visibility means that the variable can be checked for statistics. This attribute is optional (#IMPLIED). The type of the variables is a mandatory attribute for a declaration (#REQUIRED). A name must be provided for <*var*>. Optionally, a variable may include an initialization value and a number of cells if it is going to be used as array. More detailed information about DTD may be found at [25].

```
<abstractionRepository>
   <template name="FLOORS" concreteDomain="INT">
       <map>
           <abstractValue name="LOWER" infoLevel="1">  ... </abstractValue>
           <abstractValue name="MIDDLE" infoLevel="1"> ... </abstractValue>
           <abstractValue name="UPPER" infoLevel="1">  ... </abstractValue>
           <abstractValue name="NOUPPER" infoLevel="2" moreImpreciseThan="LOWER.MIDDLE" />
           <abstractValue name="NOLOWER" infoLevel="2" moreImpreciseThan="MIDDLE.UPPER" />
           <abstractValue name="UNKNOWN" infoLevel="3" moreImpreciseThan="NOUPPER.NOLOWER" />
           <abstractValue name="ILLEGAL" />
       </map>
       <operation id="INC">
           <option><test left="LOWER" />
                   <effect return="MIDDLE" />
           </option>
           <option><test left="MIDDLE" />
                   <effect return="NOLOWER" />
           </option>
           <option><test left="NOUPPER" />
                   <effect return="UNKNOWN" />
           </option>
           <option><test left="NOLOWER" />
                   <effect return="NOLOWER" />
           </option>
           <option><test left="UPPER" />
                   <effect return="ILLEGAL" />
           </option>
           <default return="ILLEGAL" />
       </operation>
   </template>
</abstractionRepository>
```

Fig. 9. An Abstraction Repository with part of the `FLOORS` abstraction

5.2 Abstraction Library

As was commented earlier, abstraction functions are stored as templates in XML format, which avoids modelling-language dependencies. For this purpose, we have defined a new metalanguage for the repository. In this section, we present some

of its features using the example in Fig. 9 which shows part of the structure for the FLOORS abstraction. The root element is named <*abstractionRepository*>, and it must include at least one <*template*> element. A template specifies a name and the concrete domain of the variables to be abstracted. Every template must also define the abstraction of every concrete value, that is, the abstraction function α. This information is stored within a unique <*map*> element. A map transforms each concrete data <*item*> into its corresponding abstract data. This transformation is represented by a sequence of tags (one tag for each abstract data) <*abstractValue*>. Each tag selects the concrete values approximated by the corresponding abstract value, making use of a Boolean expression. Such Boolean expressions are also written in XML as <*expression*> elements. This is not shown in the figure for simplicity.

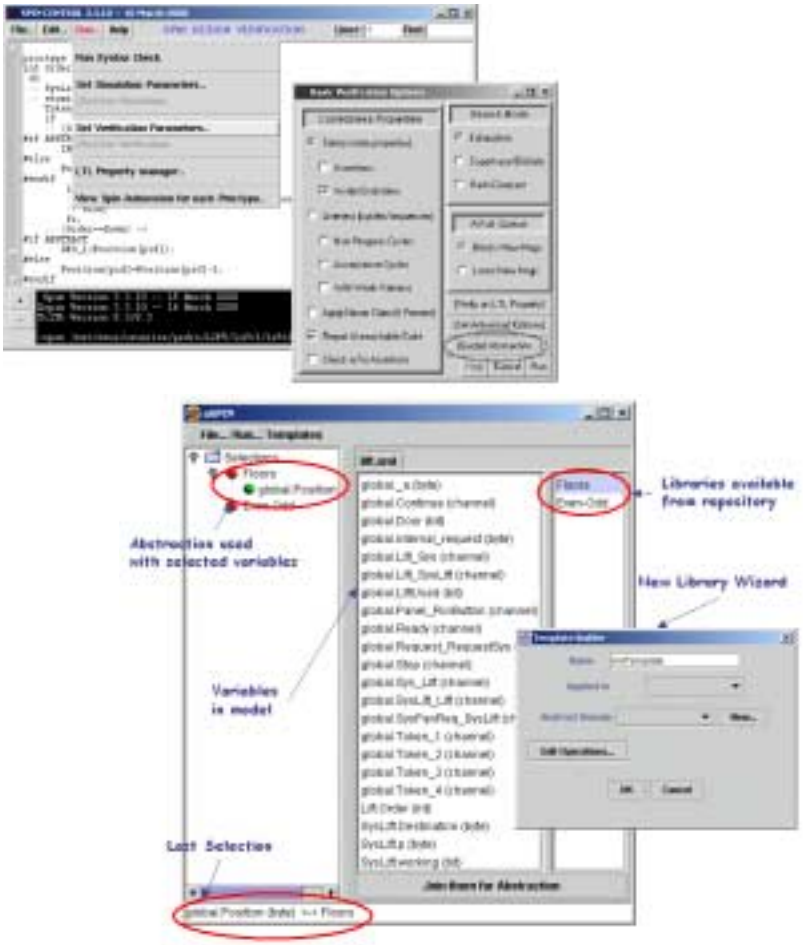

Fig. 10. Graphical User Interface

Each abstract value has an identifier and, optionally, a number indicating its precision degree (greater numbers correspond to less precise values). With this additional data, users can easily extract the partial order defined over the abstract domain.

Abstract operations have a name which must match the name used by the XML metalanguage when translating the original PROMELA models. The definition of each operation includes a collection of transforming options <*options*>, every of which has one <*test*> tag to represent the argument of the operation and a <*effect*> tag to represent its result. When more complex expressions appear, they are incorporated as <*expression*> tags in order to be evaluated by the back-end module as follows. When a test is satisfied, its effect may be a constant value, as an attribute when no tag is nested or, otherwise, the result of the evaluation of a complex expression.

6 Evaluating αSPIN

Fig. 10 shows part of the Graphical User Interface for the current version of αSPIN. The GUI gives information about the variables contained within the model (name, type and context: global or local), the available templates in the abstraction library and the assignment of templates to variables. The user can establish/delete relations of variables with templates (step 4 in Fig. 7). Then the PROMELA output is obtained (steps 5 to 7 in Fig. 7).

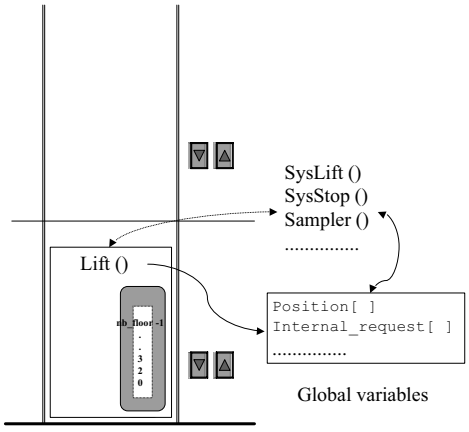

Fig. 11. Lift schema

Regarding the implementation of the architecture in Fig. 7, we have implemented two parsers Promela2XML and XML2Promela, which are fully in-

tegrated within XSPIN and whose use is hidden from the user. The abstraction module is written in Java, using the JDOM library.

One of the models employed to test our approach is a variant of the PROMELA code for an elevator controller presented in [9] (see Fig. 11). The original specification considers a controller system to manage n lifts, and our aim is to verify that the same control structure also works for only one lift. The control part receives the inputs from the environment and sends the orders to the Lift() process. This part is divided into several processes that communicate via rendezvous channels and global variables. The main variables to control the flow in every process are the global variable `Position`, which always stores the current floor for the lift, and the global array `internal_request[nb_floor]`, which stores the pending requests to move to specific floors, `nb_floor` being the actual number of floors in the system. The code in Fig. 3 shows the updating of this variable in the `Lift()` process, depending on the order from the control part (`Up`, `Down`, `OpenDoors`,...). The whole model can be found at [25].

We have chosen one of the most critical and time consuming temporal properties to show the power of automatic abstraction. The property `Move` says that *"the lift always starts the movement to the requested floor"*

As in any verification work, we start checking that the system is deadlock free using several configurations of floors. Fig. 12 shows how the number of transitions and visited states increases as the number of floor increases (transitions and states in the concrete model). The full-state search can only be performed for a small number of floors. For example, the configuration with 100 floors produces 3.485841e+7 states, using bit-state partial verification.

The verification of the temporal property `Move` is performed after being sure that the model is deadlock free. This property can be formalized as follows:

```
Move: [] ((reqL && posU ) -> <> posBelowU) && ((reqU && posL) ->
          <>posAboveL) && ((reqM && noPosM)-> <> posM))
```

where the propositions `reqL`,`reqU` and `reqM` represent requests from *lower*, *upper* and *middle* floors, respectively. Propositions `posU`, `posBelowU`, `posL`, `posAvobeL`, and `noPosM` represent whether the lift is currently at, above or below, an specific floor. These propositions are defined according to the interpretation standard or non-standard as defined in Section 4.2.

The main problem in verifying the concrete model is that the verification time is highly dependent on the number of floors, and it is not scalable when this parameter is increased to high values. Fortunately, propositions in the formula `Move` give us a guide on how to abstract. As the evaluation of these propositions mainly relies on the value of the variable `Position`, and this variable is used as a counter, we could employ the `FLOORS` abstraction to reduce the state space to be visited. However, the use of `FLOORS` implies that the global array `internal_request[nb_floor]` has to be abstracted by an array with only three components (this information is suggested by the abstraction tool by analyzing the XML representation of the model).

The results verifying the abstract model are shown in Fig. 12. Note that it is necessary to ensure that the system is deadlock-free before checking the temporal

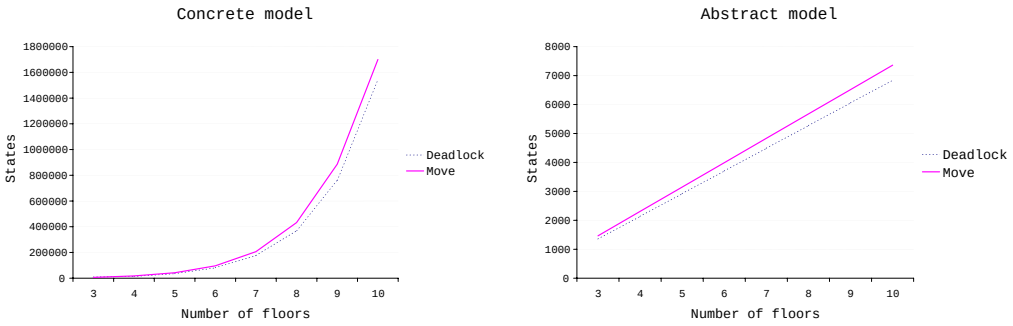

Fig. 12. Verification results

property `Move`. In this way, we can apply all the practical results related to the verification of universal temporal properties. Property `Move` is satisfied by all the analyzed configurations, but as expected, the number of visited states is very small compared to the concrete model. Furthermore, the variation of the number of states is linear with respect to the number of floors. Using the results in Theorem 2, the verification of `Move` in the abstract model gives us information about the behavior of the original model.

7 Discussion

The use of XML provides new interesting capabilities compared to other tools that perform abstraction for model transformation. In particular, the encoding of abstraction functions with XML allows us to extend the classical definition of abstract operations with new information. For example, the `infolevel` attribute defines the precision in the underlying lattice. The tags `test` and `effect` can support executability conditions (a key element in PROMELA).

In general, the use of XML for representing both the model and the abstraction functions provides interesting advantages regarding static analysis of models. We can easily find information about variable dependence, type inference, etc., which are very useful in helping users to select the abstraction functions. And in particular, we can improve the evaluation of heterogeneous expressions. For example, if the variable `x` has been abstracted with the classical `SIGN` abstraction that approximates each integer number by one of the elements of the set $\{neg, zero, pos\}$, and the variable `y` in the instruction `x = x + y + 1` has not been abstracted, then it is possible to produce results with different degrees of precision. With our way of storing the expression and the abstraction, it is easy to discover that the concrete sum `y + 1` must be the first expression to be executed in order to obtain the most precise result.

As regards non-determinism due to the loss of information when executing abstract operations, we use specific abstract constants instead of sets of constants as employed in [5]: for example, we use the value *noupper* instead of the set

$\{lower, middle\}$. From the implementation point of view, the use of sets implies utilizing non-deterministic assignments when updating variables abstracted. In contrast, the use of specific constants removes the use of non-deterministic assignments. The first case provokes the direct creation of several branches, while our method allows us to delay this creation until a test over the updated variable is found. Furthermore, the use of constants does not make counterexample analysis more difficult, particularly when abstract counterexamples correspond to deterministic traces [24, 23].

8 Conclusions and Further work

We have presented the actual state of αSPIN, a tool for abstracting models in the context of explicit model checking. Documentation and current and future versions of αSPIN can be found at [25]. The most novel contribution of this work is to exploit the characteristics of XML as the intermediate language to support the abstraction process. This unusual application of XML defines a language-independent framework to develop tools for abstract model checking.

Our future work is oriented in two directions. The first is to add functionality to αSPIN, including features such as a friendly GUI to create new abstraction functions, and strategies to automatically analyze their correctness using PVS. The other line of work is to improve counterexample analysis. Current works on this subject have been developed mainly for symbolic model checking. We are now interested in finding new strategies for explicit model checking, which are usually based on the analysis of deterministic traces.

References

1. E. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.
2. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the Int. Conf. on Computer Aided Verification-CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, 2000.
3. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1245, 1994.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking and abstraction* The MIT Press, 2000
5. J. Corbett, M. Dwyer, J. Hatcliff, L. Shawn, C. Pasareau, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. Int. Conf. On Software Engineering*, pages 439 – 448, 2000.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
7. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

8. D. Dams, R. Gerth, S. Leue, and M. Massink, editors. *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*. Springer, 1999.
9. G. Duval and T. Cattel. From architecture down to implementation of safe process control applications. Design, verification and simulation. In *Proc. of the Thirtieth Annual Hawaii International Conference on System Sciences*, 1997.
10. M.M. Gallardo and P. Merino. A framework for automatic construction of abstract promela models. *In [8]*, pages 184–199, 1999.
11. M.M. Gallardo, J. Martínez, P. Merino and E. Rosales, Using XML to implement Abstraction for Model Checking. In *Proc. of ACM Symposium on Applied Computing*, 2002. To appear.
12. M.M. Gallardo, P. Merino, and E. Pimentel. Syntatic transformation of PROMELA for abstract model checking. *Submitted*.
13. M.M. Gallardo, P. Merino, and E. Pimentel. Abstract satisfiability of linear temporal logic. In *Proc. of I Jornadas sobre Programacion y Lenguajes* pages 163–178, 2001.
14. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proc. of the Conf. on Computer Aided Verification CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Verlag, 1997.
15. K. Havelund and T. Pressburger. Model checking java programs using java path finder. *Int. Journal On Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.
16. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
17. G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
18. G. Holzmann and M. Smith. A practical method for the verification of event driven systems. In *Proc. Int. Conf. On Software Engineering*, pages 597–608, 1999.
19. J. Hunter and B. McLaughlin. The jdom project. Available in http://www.jdom.org, 2000.
20. C. Loiseaux, S. Graf, J. Sifakis, and S. B. A. Boujjani. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer-Verlag, New York, 1992.
22. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
23. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Proc. of the Int. Conf. TACAS 2001* Volume 2031 of *Lecture Notes in Computer Science*, pages 284-298, 2001.
24. H. Saïdi. Model checking guided abstraction and analysis. In *Proc. of the Seventh Int. Static Analysis Symposium SAS2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 377-389, 2000.
25. αSPIN project. University of Málaga. http://www.lcc.uma.es/~gisum/fmse/tools
26. W3Consortium. Xml path language (xpath) version 1.0. Available in http://www.w3.org/TR/xpath, 1999.
27. W3Consortium. Extensible markup language (xml) 1.0 (second edition). Available in http://www.w3.org/XML/ 2000.
28. W3Consortium. Document object model (dom). Available in http://www.w3.org/DOM/, 2001.
29. W3Consortium. Xsl transformations (xslt) version 1.1. Available in http://www.w3.org/TR/xslt11/, 2001.