Using XML to implement Abstraction for Model Checking [*]

María del Mar Gallardo, Jesús Martínez, Pedro Merino, Estefanía Rosales
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga
Campus Teatinos, 29071 Málaga, SPAIN
{gallardo,jmcruz,pedro,estefania}@lcc.uma.es

ABSTRACT

Model Checking has become one of the most powerful methods for automatic verification of software systems. However it is widely accepted that this technique is only usable when the behavior of the system to be analyzed is given by small models, in order to avoid the state explosion problem. The paper presents αSPIN, an XML-based tool for obtaining abstract versions from a given model written in PROMELA, which can be verified with the model checker SPIN. This tool follows the theoretical basis presented in [9].

Keywords

XML, Model Checking, Abstraction, SPIN

1. INTRODUCTION

For many years, verification of software systems followed the traditional deductive approach, where a theorem-proving tool could assist in the debugging task. But due to the need of ingenuity to obtain good results, only a few teams have been able to apply this approach to industrial-scale software. *Model Checking* has appeared as a clear opponent to the traditional verification method, particularly in the development of reliable software for concurrent systems [2, 4]. With this technique it is necessary to write two formal descriptions of the problem. One represents the actual behavior of the system, the current design, and it is usually called the *model*. The second description represents the *properties* the model is expected to have. Properties are mainly expressed with Temporal Logic [17], using formulas such as $\Box$ p ("p is always true"), $\Diamond$ p ("eventually p will be true") or pUq ("q will be true, and p will be true in all the previous states"), p and q being any kind of propositions or even temporal formulas. Verification usually consists of ensuring that every execution path, in the current design for the software, meets a particular property. This task is carried out automatically by generating and analyzing all the potential (finite) states of the program (exhaustive analysis). When required, the tool produces the counter-examples to detect the paths that violate the property.

Current model checking tools can perform the analysis using an *on-the-fly* method, in such a way that executions are constructed on demand (depending on the formula) and the generation of many states can be avoided in order to decide about the satisfaction/non-satisfaction of the formula. However, real world systems generate as many states as analyzable with current technology, thus producing the *state explosion* problem. Among other techniques, *abstraction* is currently one of the most exciting methods to reduce the cost of verifying these very complex systems. This method replaces the original model by a simpler one (an abstract model) removing details which are irrelevant for the property to be checked.

One of the most well-known abstraction techniques is abstract interpretation [6] which consists of replacing a set of concrete values with a given abstract value, by using the so-called abstraction function (usually denoted as α). As one abstract value represents a set of concrete values, there is a loss of information that must be considered when executing the instructions. For example, it is usual for a Boolean expression to always return true for a particular combination of abstract values, even if the value must be false for some of the represented concrete values. This is done to represent the more general behavior for the concrete values. Many proposals to use abstractions with model checking are based on this idea [3, 15, 7, 9, 10]. Most of these papers focus on the theoretical basis to ensure the correctness of the abstraction (i.e. the conditions to ensure that the results in the abstract model can be related to results in the initial one). In general, they do not focus on implementation performance or practical usability.

Despite the potential benefits of abstraction as a general optimization method, it is only practical if the user is assisted in choosing the correct abstraction functions (the α's) and in obtaining the final abstract model. The selection of the abstraction functions must be driven by the property to be verified, in such a way that the abstract model preserves relevant information to check the property. Furthermore, these functions should satisfy correctness conditions, but it is not desirable to perform the correctness analysis every time they are employed. An additional problem is to find an automatic method to obtain the abstract code, removing the potential errors introduced by a hand-made abstraction.

[*]This work has been partially supported by the FEDER TAP-1FD97-1269-C02-02 and CICYT-TIC99-1083-C02-01 projects

SAC 2002 Madrid, Spain
© 2002 ACM 1-58113-445-2/02/03 ...$5.00.

Generally, expensive implementations are used, all of which are related to an specific tool. This paper focuses on the implementation aspects of these topics to obtain a tool that makes abstraction a practical task for users of model checkers.

Our work is based on the theoretical proposal given in [9], where the abstraction could be obtained by a source-to-source transformation of the initial model, in such a way that the same model checker could perform the verification task with the initial and the abstract model. In [9] we present the approach to perform the transformation of software models written in PROMELA, which is the input language of the model checker SPIN. We study the conditions to obtain correct abstractions, and we propose that the abstraction functions should be stored in an abstraction library after analyzing correctness, so as to be available for users. This approach allows the use of current and future optimization techniques implemented by SPIN, and it also makes it possible to maintain the new specification accessible to the user.

We have implemented the tool αSPIN following the proposal in [9] and using XML (eXtensible Markup Language [19]) as an intermediate language to perform the PROMELA to PROMELA transformation, and also to represent the abstraction functions. The abstraction is actually carried out in an XML representation of the initial PROMELA model. Then, the abstract PROMELA version is generated. This approach provides a number of advantages: a) The transformation work requires an intensive search for patterns in the model, taking into account not only the syntax, but also the semantics of the elements. The work can be efficiently done with several available tools and APIs to process XML (see [1]); b) XML is clearly suitable as a common interchange format among tools [16]. The representation of the PROMELA models with this language could be exploited for other development phases, such as code generation [5]; c) The construction of the abstraction tool independently of the actual modeling language allows the reuse of most of the code to support the same abstraction by transformation scheme with other specification languages (for instance, we are planning a variant of the tool for SDL following the proposal in [10]).

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 [18].

The paper is organized as follows. Section 2 contains some preliminary presentations of the SPIN tool and the abstraction by transformation method proposed in [9]. Sections 3 and 4 are devoted to the design aspects of the XML based tool and to the implementation details, respectively. Finally, in Section 5, we present some conclusions.

2. PRELIMINARIES

2.1 SPIN/PROMELA

In the last few years, SPIN has become one of the most employed model checkers in both academic and industrial areas (see [8]). It supports the verification of usual safety properties in PROMELA models (like absence of deadlock) and also the analysis of complex requirements expressed with Linear Time 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 of an exhaustive exploration of the state space in search of 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 the counterexample is not found, then the model satisfies the LTL property (all possible execution branches satisfy the property). Formulas that are satisfied in all executions are called *universal temporal formulas*.

PROMELA is designed for systems composed of concurrent processes that communicate asynchronously (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 syntax is a mixture of C and CSP, as shown in Fig 1. This figure contains the code for a typical system in model checking, the ABP protocol, which will be employed in Section 3. The ABP protocol is designed to enable the transmission of data over an unreliable channel. The error-free behavior of the protocol consists of `abp_sender` sending data message with a control bit and receiving a reply message with the same bit. Control bits between `abp_sender` and `abp_receiver` alternate for every new data. Our version for this protocol also considers the process `Channel` to model an unreliable channel between `abp_sender` and `abp_receiver`. This process has been omitted. The users of ABP are the processes `sender` and `receiver`. The first one sends a number of data, and the second checks the data to find odd and even data.

Detailed descriptions of SPIN and PROMELA can be found in [12] and [13].

2.2 Abstracting Promela by transformation

In this section, we give an overview of the proposal presented in [9]. The abstraction scheme is based on defining the so-called *generalized semantics* of PROMELA. Semantics $Gen(-, effect)$: PROMELA $\rightarrow \wp(Trace)$[1] associates each model M with the set of traces $Gen(M, effect)$, where $Trace$ represents the set of possible model simulations. Function $effect$ defines the way to interprete PROMELA basic instructions. Thus, if $effect^P$ denotes the standard behavior, then $Std(M) = Gen(M, effect^P)$ gives the usual meaning to each model M. An abstract semantics $Std^\alpha(M) = Gen(M, effect^\alpha)$ is obtained from $Std(M)$ by abstract interpretation as follows: first, we must abstract the domain of some variables by means of an abstraction function, usually denoted as α, and secondly, we must define a function $effect^\alpha$ in order to give the proper abstract meaning to the model sentences. Correctness of the abstraction can be analyzed by reasoning about the two semantics.

Consider the simple PROMELA model M illustrated in the

[1] In [9], Gen has another parameter, denoted by $test$, which represents the executability of PROMELA sentences.

```
mtype = {ack, err, mesg, next,accept};


proctype abp_sender(chan in, chin, chout){
  byte data_s;
  bit seq_out_s = 0, seq_in_s;

  do
  :: in ? next(data_s) ->
     do
     ::chout ! mesg(data_s, seq_out_s) ->
         if
         ::chin ? ack(seq_in_s) ->
            if
            ::seq_in_s == seq_out_s -> seq_out_s = !seq_out_s;
                                         break
            ::seq_in_s != seq_out_s ->  skip
            fi
         ::chin ? err(_) ->  skip
         ::timeout ->  skip;
         fi
     od
  od
}

proctype abp_receiver(chan out, chin, chout){
  byte data_r;
  bit seq_in_r, seq_exp_r = 0;

  do
  ::chin ? mesg(data_r, seq_in_r) ->
     if
     ::seq_in_r  == seq_exp_r -> seq_exp_r = !seq_exp_r;
                                 chout ! ack(seq_in_r);
                                 out ! accept(data_r)
     ::seq_in_r != seq_exp_r  -> chout ! ack(seq_in_r)
     fi
  ::chin ? err(_,_) -> chout ! ack(seq_in_r)
  od
}
proctype sender(chan out) {
  int n = 0;
  do
  ::(n <= TRANSFER_LIMIT) -> out ! next(n); n++
  ::(n > TRANSFER_LIMIT) -> out ! next(END);break
  od
}
proctype receiver(chan in) {
  int n;
  do
  ::in ? accept(n) ->
      if
      ::n != END ->
         if
         ::(n % 2) == 0 -> printf("received even: %d", n)
         ::(n % 2) != 0 -> printf("received odd: %d", n)
         fi
      ::n == END -> printf("Received end of file");break
      fi
od
}

 chan AtoC = [1] of { mtype, byte, bit };
 chan CtoB = [1] of { mtype, byte, bit };
 chan BtoC = [1] of { mtype, bit };
 chan CtoA = [1] of { mtype, bit };
 chan UserAout  = [1] of {mtype, byte };
 chan UserBin  = [1] of {mtype, byte };
 init {
   atomic { run sender(UserAout); run receiver(UserBin);
     run abp_sender(UserAout, CtoA, AtoC);
     run abp_receiver(UserBin, CtoB, BtoC);
     run channel(AtoC, CtoA, BtoC, CtoB);
    }
}
```

Figure 1: Part of Promela model for ABP system

```
                                 #define even 0
int x;                           #define odd 1
active proctype p1()             #define evenodd 2
{                                byte x = even;
start:if                        active proctype p1(){
      :: x = x + 2;             start:if
         goto start;                  ::x=(x==even->even:odd);
      :: goto end;                       goto start;
      fi                              :: goto end;
end: skip }                           fi
                                end: skip }
```

Figure 2: Concrete and Abstract model Even

left column of Fig. 2. The model has only one process ($p1$) and one global variable x ranging over Int, the set of the integer numbers. Assuming that program states are represented by the integer value v_x stored in x, the standard effect of the sole instruction in the model $x = x + 2$ is $\mathit{effect}^P(x = x+2, v_x) = v_x + 2$. Thus, the standard semantics $Std(M)$ contains the set of the finite traces $\{0, 0 \rightarrow \mathit{effect}^P(x = x+2, 0), 0 \rightarrow 2 \rightarrow \mathit{effect}^P(x = x+2, 2), \dots\}$ together with the infinite trace $0 \rightarrow 2 \rightarrow 4 \rightarrow \dots$. Now consider the abstraction function $\alpha : Int \rightarrow \{even, odd\}$ which transforms each even integer into *even* and each odd integer into *odd* (in what follows it will be the abstraction Even-Odd). As before, an abstract state is the abstract value a_x of x, that is, one of the values *odd* or *even*. The redefinition of data given by α makes it necessary to abstract the function effect^P to implement the effect of executing the instruction $x = x+2$ in abstract states. We could arbitrarily define effect^α, but the natural definition coherent with the intuitive meaning of α is given by $\mathit{effect}^\alpha(x = x+2, a_x) = a_x$, since adding 2 does not modify the parity of a_x. With this definition, the abstract semantics $Std^\alpha(M)$ contains the set of the finite traces $\{even, even \rightarrow \mathit{effect}^\alpha(x = x+2, even), even \rightarrow even \rightarrow \mathit{effect}^\alpha(x = x+2, even), \dots\}$ together with the infinite trace $even \rightarrow even \rightarrow even \rightarrow \dots$.

Assuming that α is first extended to traces and then to sets of traces, correctness of abstraction is represented by the expression $\alpha(Std(M)) \subseteq Std^\alpha(M)$, which intuitively means that each trace in $Std(M)$ is approximated by an abstract trace in $Std^\alpha(M)$. In [9] there is an exhaustive study of the conditions that effect^α must verify for $Std^\alpha(M)$ to be a correct approximation of $Std(M)$.

Besides the previous correctness result, some practical results regarding the preservation of properties are also obtained in [9]. On the one hand, we establish the conditions under which the absence of deadlock in $Std^\alpha(M)$ implies the absence of deadlock in $Std(M)$. On the other, with respect to the preservation of *universal temporal formulas*, we obtain several results that relate the satisfaction of formulas in $Std^\alpha(M)$ to the satisfaction in $Std(M)$. In particular, we study when the following assertion holds: "If f is a universal formula and $Std^\alpha(M) \models f^\alpha$, then $Std(M) \models \overline{f}$", where f^α is the abstract version of the temporal formula f. $Std(M)$ verifies the weaker formula $\overline{f}$ instead of f, due to the loss of information produced by the abstraction process.

As regards implementation, we propose a source-to-source transformation based on replacing each instruction in the original model M by a standard PROMELA code that implements the meaning given by effect^α in order to obtain a

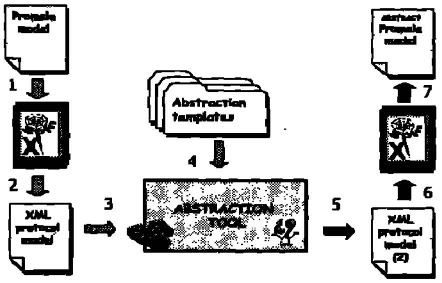


Figure 3: Architecture of αspin

new model M^α. Then the model checker works only with standard instructions. This approach corresponds to verifying $Gen(M^\alpha, \mathit{effect}^P)$. For example, the second column in Fig. 2 shows one possible transformation of the model M following the abstraction Even-Odd. The standard effect of executing the instruction $x = x + 2$ has only been substituted by one implementation of the function effect^α.

3. XML-BASED TRANSFORMATION

As every model checker uses a particular modeling language, each one has a specific parser and additional support to convert the model specification into a suitable internal data structure to be used in 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 needed in abstraction. Our approach consists of using an additional language based on XML for representing and storing the models for verification. This language must be expressive enough and must allow an easy analysis and manipulation.

We have defined a vocabulary of tags and a DTD (Document Type Definition). The DTD file helps to verify that an XML document was well formed, i.e., respecting the full PROMELA grammar, which represents a prerequisite for later abstraction processing. Documents are built with the aid of two new tools embedded into XSPIN: one for translating PROMELA models into XML documents, and one working in the opposite direction.

Fig. 3 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 the transformation of the original model into an XML document suitable for external manipulation. Then the abstraction tool takes two files as input: the XML model obtained previously and a file defining a repository of abstraction templates, also in XML format.

Fig. 4 shows part of the *promela.dtd* file, the DTD used as template for building valid XML models. Each element, in DTD terminology, has a corresponding tag. The first tag found is *<model>*, which may contain other nested tags, representing processes, declarations of global variables, new type definitions, startup scripts, etc. The figure also shows the elements *<declaration>* and *<process>*. They are particularized with attributes like *name*, or *type* in declarations.

```
 <!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>
```

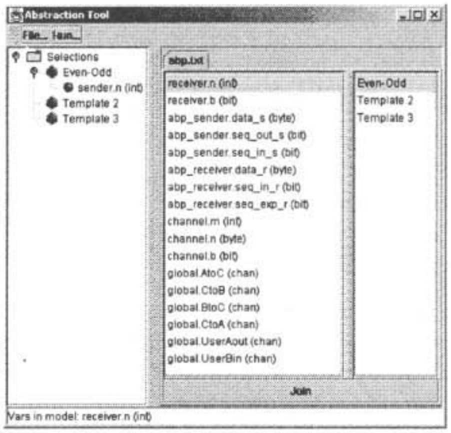
Figure 4: Part of promela DTD

Our abstraction tool αSPIN includes a Graphical User Interface (see Fig. 5) giving information about the number of variables contained within the model: name, type and context (global or local). We also include information about available templates. The user only has to match variables with a corresponding template, thus obtaining the XML output (steps 3,4 and 5 in Fig. 3). When this process is finished, resulting XML output can be imported into XSPIN (the GUI for SPIN) and converted back into PROMELA (steps 6 and 7).

Let us consider the ABP model in Fig. 1, and the abstraction function α partially defined in Section 2.2. The first step is to convert the PROMELA model into an XML format to be browsed with the GUI. In Fig. 5, all the model variables are also listed, and the user can select one of them to match it with one of the recognized templates of the abstraction repository (right list). Note that the local variable n, on the `receiver` and `sender` processes, constitutes a good selection for an Even-Odd abstraction scheme. The *Run...* menu option does the abstraction work, giving as output the abstracted version of the ABP file, in XML format. Applying a conversion back to PROMELA is required for the verification process with SPIN. SPIN produces 43731 states for deadlock-free analysis in the concrete model, whereas for the abstract one it only explores 6980. There is a more important state-reduction in the verification of the LTL formula for the property *"When an even message is sent, eventually an even message is received"*: 143896 states stored in concrete model versus 15364 in the abstract one.

4. IMPLEMENTATION NOTES

Regarding the implementation of the architecture in Fig. 3, we have implemented two parsers: PromelatoXML and XMLtoPromela, which are fully integrated within XSPIN and whose use is hidden from the user. The abstraction module is written in Java, using JDOM library [14]. PromelatoXML, includes a new module, written in C (over 1300 lines of code), used for generating XML tags and full documents. XMLtoPromela is an external tool to SPIN, written in Java. Its design was inspired by object oriented flexibility and software design patterns. The main goal for this tool is to be a framework for producing output files in a variety of languages, based on a well-formed XML document, as described above. In the first version a PROMELA code is obtained but with a few modifications made in a flexible way, a Java or



Abstraction Tool

Selections
- Even-Odd
  - sender.n (int)
- Template 2
- Template 3

abp.txt

receiver.n (int)
receiver.b (bit)
abp_sender.data_s (byte)
abp_sender.seq_out_s (bit)
abp_sender.seq_in_s (bit)
abp_receiver.data_r (byte)
abp_receiver.seq_in_r (bit)
abp_receiver.seq_exp_r (bit)
channel.m (int)
channel.n (byte)
channel.b (bit)
global.AtoC (chan)
global.CtoB (chan)
global.BtoC (chan)
global.CtoA (chan)
global.UserAout (chan)
global.UserBin (chan)

Even-Odd
Template 2
Template 3

Vars in model: receiver.n (int)

Figure 5: User GUI

C++ code can also be obtained. Although these are not high level modeling languages, they can be very useful for designers to obtain code skeletons that are produced from the verified PROMELA model. The original design was inspired by the Interpreter pattern [11]. PROMELA grammar in BNF notation conditioned the behavior of the classes and their relationship. A SAX event-handler class was designed for passing data from XML tags to appropriate object instances. The refined pattern supporting the mapping process is shown as UML class diagram in Fig. 6. InterpreterParser performs the whole translation process.

5. CONCLUSIONS

The main contributions of the paper are the definition of a DTD for PROMELA and the implementation of the tools to make translations between XML and PROMELA. This work has been done taking into account the abstraction method proposed in [9], on the basis of which we have obtained a fully working tool to perform abstraction based model checking. The paper also explores automatic verification as a novel application area for XML in software engineering.

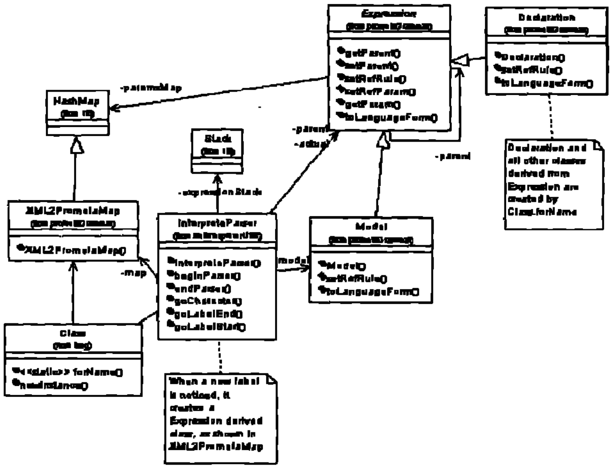


Figure 6: UML diagram class of XMLtoPromela

6. REFERENCES

[1] P. B. S. Abiteboul, D. Suciu. *Data on the Web : From Relations to Semistructured Data and Xml.* Morgan Kaufmann Publishers, 1999.

[2] E. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[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.* The MIT Press, 2000.

[5] J. C. Cleaveland. *Program Generators with XML and Java.* Prentice-Hall, 2001.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *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 TOPLAS*, 19(2):253–291, 1997.

[8] D. Dams, R. Gert, S. Leue, and M. Massink, editors. *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*. Springer, 1999.

[9] M. Gallardo and P. Merino. A framework for automatic construction of abstract promela models. *In [8]*, pages 184–199, 1999.

[10] M. Gallardo and P. Merino. A practical method to integrate abstractions into SDL and MSC based tools. In *Proc. of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, pages 84–89. GMD Report 91, 2000.

[11] E. Gamma, H. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Pub Co., 1995.

[12] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[13] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[14] J. Hunter and B. McLaughlin. The JDOM project. Available in http://www.jdom.org, 2000.

[15] 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.

[16] M. W. M. Jngel, E. Kindler. Towards a generic interchange format for petri nets. In *Proc. of Meeting on XML/SGML based Interchange Formats for Petri Nets*, 2000.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer-Verlag, New York, 1992.

[18] αSPIN project. University of Málaga. http://www.lcc.uma.es/ gisum/fmse/tools.

[19] W3Consortium. Extensible markup language (xml) 1.0 (second edition). Available in http://www.w3.org/XML/, 2000.