A Framework for Automatic Construction of Abstract Promela Models

Maria-del-Mar Gallardo and Pedro Merino
{gallardo,pedro}@lcc.uma.es

Dpto. de Lenguajes y Ciencias de la Computacion
University of Malaga, 29071 Malaga, Spain

Abstract. One of the current trends in model checking for the verification of concurrent systems is to reduce the state space produced by the model, and one of the more promising ways to achieve this objective is to support some kind of automatic construction of more abstract models. This paper presents a proposal in this direction. The main contribution of the paper is the definition of a semantics framework which allows us to relate different models of the system, each one with a particular abstraction level. Automatic source-to-source transformation is supported by this formal basis. The method is applied to Promela models.

1 Introduction

Formal verification is a powerful method to ensure confidence regarding the correctness of many complex and critical systems [5, 8]. This technique is currently supported by many commercial and non-commercial tools such as SPIN [11, 12]. However, verification is only possible and fruitful if *useful formal models* of these systems are available. A useful model is an abstract representation of the real system, with exactly the details necessary to ensure that satisfaction of interesting properties in the model implies satisfaction in the real system. Excessive model detail may produce the well-known state explosion problem, which could prevent its analysis with current tools. Whereas research in recent years has mainly focussed on algorithms to improve automatic verification, mainly based on model checking [3, 15], it is now necessary to conduct research into methods for the automatic construction of useful abstract models (as defended by Amir Pnueli in the 4th SPIN worksho p).

One technique recently exploited to obtain more abstract models is abstract interpretation [2], which allows us to employ the new models in order to analyze specific properties using less time or memory [4, 6]. Abstract interpretation (A.I.) is an automatic analysis technique to statically deduce dynamic program properties, which is based on the idea of approximation. Every program data is approximated, by means of the so-called abstraction function α, by a higher level description (abstract denotation) which represents the data property of interest. Analysis is carried out by executing programs with the abstract data instead of the actual data. To do this, it is necessary to redefine the meaning of the

program instructions so that they can be applied to abstract data. In both [4, 6], transition systems are used to construct models and the abstraction is oriented to the verification of universal safety temporal properties, i.e., properties that hold in all states along e very possible execution path. Both works also extend their proposal to existential properties. In [4], properties are expressed with CTL formulas, while in [6] the modal μ-calculus is used.

This paper reports work in progress towards the construction of an environment for automatic verification based on transforming Promela models using abstract interpretation as a formal basis. The main components of the environment are shown in Figure 1. The key concept is that given a model M, the user must supply an abstraction function α to transform this model into a new abstract model M_i^*. The function can be provided from a library, and it can be refined for this particular model using the property to be analyzed. The properties must also be transformed when the model is transformed. Our aim is to keep all the formal descriptions (models and properties) as related as possible to the results, by using a specific management tool.

We present a semantics framework to support the transformations mentioned above based on A.I. This analysis technique basically defines a relation between two semantic levels, the concrete and the abstract one. Given a language L and $Sem : L \rightarrow (Det, \leq)$ a semantics of L which associates each program $M \in L$ with a denotation d belonging to the poset $(Det, \leq)$, the objective of A.I. is to automatically construct an abstract program M^* in which the program characteristics to be analyzed are preserved while the rest of the program characteristics are abstracted. The meaning of the new program M^* is given by an abstract semantics $Sem^* : L \rightarrow (Det^*, \leq^*)$. Correctness of the analysis is proved by means of the abstraction function $\alpha : (Det, \leq) \rightarrow (Det^*, \leq^*)$ and the concretization function $\gamma : (Det^*, \leq^*) \rightarrow (Det, \leq)$ which relate these two semantic levels. $((Det, \leq), (Det^*, \leq^*), \alpha, \gamma)$ usually forms a Galois connection and the correctness is fo rmalized using any of the two following equivalent expressions:

$$\alpha(Sem(M)) \leq^* Sem^*(M^*) \tag{1}$$

$$Sem(M) \leq \gamma(Sem^*(M^*)). \tag{2}$$

$\leq$ and $\leq^*$ represent the precision given by the two respective semantics, i.e., $d_1^* \leq^* d_2^*$ indicates that the semantics value d_1^* is more precise than d_2^*, or from another point of view, that d_2^* approximates d_1^*. Following this, $\alpha(d) \leq^* d^*$ means that d^* is an abstract approximation of d, and $d \leq \gamma(d^*)$ indicates that $\gamma(d^*)$ is more general than d. Considering this, (1) and (2) represent that $Sem^*(M^*)$ is a correct approximation of $Sem(M)$.

Many program aspects of the concrete semantics construction are not affected by the abstraction process, and therefore it is possible to define a semantics parameterized by the language aspects which are influenced by the process of abstraction. This idea was used in [13] for the A.I. of Prolog and more recently in [7] who defined the generalized semantics of constraint logic languages. We follow this idea to define a generalized semantics of a subset of Promela. This semantics will allow us to define, in a common semantics framework, different levels of

abstraction from an initial model and to easily compare them for precision. The key issue here is that all models, the abstract ones and the initial, are instances of the same semantic framework. Compared to other related works, our proposal is based on the automatic source-to-source transformation of Promela models, thus allowing the use of SPIN for verification of both concrete and abstract models. In [9] we presented previous resul ts on the use of A.I. for verifying abstract properties of programs.

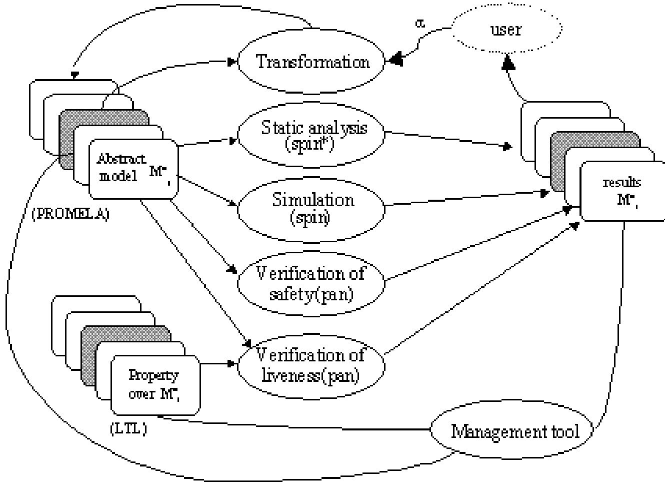

Fig. 1. Overview of an environment for verifying by transformation

The organization of the paper is as follows. Section 2 describes the generalized semantics of Promela and Section 3 explains the transformation method proposed and presents some correctness results. Section 4 contains an example which illustrates how to combine our method with the SPIN tool. In Section 5, we present conclusions and future work.

2 Promela generalized semantics

The objective of this section is to define the generalized semantics of a significant subset of the Promela language. As explained above, the generalized semantics describes the operational behavior of a program making explicit the domain-dependent model characteristics which are influenced by the abstraction as data and instructions. We first define the subset of Promela considered in the paper. We do not try to precisely explain the syntax or the meaning of each Promela instruction. Instead, some knowledge of the language is assumed.

Every model $M \in Promela$ is a sequence of processes $M = P_1||\ldots||P_n$ which run in parallel. Let $Inst$ be the set of basic instructions from which the processes are constructed. $Inst$ includes the assignment instruction $=$, the Boolean and arithmetic operators, the if and $goto$ instructions, and the instructions for sending (receiving) messages to (from) channels represented by the sets $Input$ and

Output, respectively. Let us define *Label* as a set of labels and *Decl* as the declarative part of the model. Using these definitions, every process is described as:

$$P = Decl; \{Label : Tran\}$$

where

$$Tran = \{(Input|Output|null); Inst\},$$

null being the empty instruction.

We intentionally omit the instruction *run* that creates the processes from the main program in order to make the description clearer. Also, we assume that *init* is one of the model's processes.

Every label is intended to represent an internal process state defined by the programmer. In each one of these logical states, the process will carry out a transition which will usually begin reading or sending a message through a channel and will follow this with a sequence of arbitrary instructions. The transition may end with a *goto* instruction which will provoke the process into jumping to another logical state. The example in Figure 4 follows the syntax defined above. We now present the generalized semantics of Promela, introducing the definition gradually in order to be clear.

1. Let *State* be the set of tuples which represent the internal state of the model, that is, every tuple, which is written as $(g, l_1, \ldots, l_n, c_1, \ldots, c_m, i_1, \ldots, i_n)$ contains the value of global variables of the model, local variables for each process, the messages stored in the model channels, and the instruction just executed by every process P_j $(j = 1, \ldots, n)$. g and l_j are also tuples, each one representing the actual value of a global or local variable at a point during the execution.
2. Let *Sequence* be the set of finite or infinite sequences of states.
3. Let $Inst_M$ and $Inst_j$ be the set of instructions of all the model processes and P_j, respectively.
4. Let $Initial : Promela \rightarrow State$ be the function which, given a model, returns its initial state, i.e., that in which its variables have been initialized, all channels are empty and each just-executed process instruction is a special one which precedes the first instruction in every process.
5. Let $next_inst_j : Inst_j \rightarrow Inst_j$ be the function which given a process P_j and an instruction i of P_j returns the instruction which follows i in the code of P_j. If i is the last instruction of P_j then $next_inst_j(i) = end$.
6. Let $just_exe_j : State \rightarrow Inst_j$ be the just-executed function which, given a process and a state, returns the last instruction of the process executed, i.e., $just_exe_j((g, l_1, \ldots, l_n, c_1, \ldots, c_m, i_1, \ldots, i_j, \ldots, i_n)) = i_j$.
7. Given $eval : BoolExp \times State \rightarrow \{false, true\}$, $eval(exp, s)$ returns the evaluation of the Boolean expression exp in the state s.
8. Let $exec_j : Inst_j \times State \rightarrow \{false, true\}$ be the executable function defined as
   - $exec_j(i, s) = eval(i, s)$, if i is a Boolean expression.
   - $exec_j(i, s) = false$, if i implies reading from an empty channel, reading a specified message from the channel which does not match with the first channel message, or writing on a full channel.

- $exec_j(i, s) = false$, if i is a non-deterministic instruction such as $if :: exp_1 - > i_1; \ldots :: exp_k - > i_k fi$ and $exec_j(exp_r, s) = false$, for all $1 \leq r \leq k$.
- $exec_j(i, s) = true$, otherwise.

In short, $exec_j(i, s)$ returns $true$ if the instruction i of process P_j does not suspend in the state s and returns $false$, otherwise.

9. Let $next_j : Inst_j \times State \rightarrow \wp(Inst_j) \cup \{delay, end\}$ be the function which given a process instruction returns the next instruction to be executed, i. e.,
   - $next_j(i, s) = delay$, if $exec_j(next_inst_j(i), s) = false$,
   - $next_j(i, s) = end$, if $next_inst_j(i) = end$.
   - $next_j(i, s) = \{exp_{r1}, \ldots, exp_{rs}\}$, if $next_inst_j(i)$ is a non-deterministic instruction such as $if :: exp_1 - > i_1; \ldots :: exp_k - > i_k fi$ and some instructions $exp_{r1}, \ldots, exp_{rs}$ exist such that $exec_j(exp_{rm}, s) = true$ $(1 \leq m \leq s)$.
   - $next_j(i, s) = next_inst_j(i)$, otherwise.
10. Let $S : (Inst_1 \cup \ldots \cup Inst_n) \times State \rightarrow State$ be a semantics function which gives meaning to each Promela instruction. $S(i, ((g, l_1, \ldots, l_j, \ldots, l_n, c, i_1, \ldots, i_j, \ldots, i_n)) = s' = (g', l_1, \ldots, l'_j, \ldots, l_n, c', i_1, \ldots, i, \ldots, i_n)$ means that executing the instruction i belonging to a given process P_j, when the model is in the state s, produces the evolution of the model towards the state s'. S is a generic function, that is, it is unspecified; we only substitute i_j by i in the state to indicate that i is the last instruction executed in the process P_j. The high level behavior of the model is not dependent on this function and this is why we do not define it. In [14], the meaning of every Promela instruction can be found.
11. Let $Trans : State \rightarrow \wp(State) \cup \{end, deadlock\}$ be the transition function which, given a model state s, returns the set of next states to which it can evolve from s. $Trans$ is defined as:
   - $Trans(s) = \cup_{j=1 \ldots n}(\cup_{i \in (next_j(just_exec_j(s), s) \cap Inst_j)}\{S(i, s)\})$, if $j \in \{1, \ldots, n\}$ exists such that $next_j(just_exec_j(s), s) \notin \{delay, end\}$
   - $Trans(s) = end$, if $\forall j = 1, \ldots, n$, $next_j(just_exec_j(s), s) = end$, and
   - $Trans(s) = deadlock$, otherwise.

Next, we define the generalized semantics $Gen : Promela \rightarrow \wp(Sequence)$ using the function *Trans*.

$$Gen(M) = \{s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow \ldots \in Sequence / Initial(M) = s_0,$$
$$\forall j > 0.(Trans(s_{j-1}) \notin \{deadlock, end\}, s_j \in Trans(s_{j-1}))\} \bigcup$$
$$\{s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \not\rightarrow \in Sequences / Initial(M) = s_0,$$
$$Trans(s_k) \in \{deadlock, end\}, \forall 0 < j \leq k.(Trans(s_{j-1}) \notin \{deadlock, end\},$$
$$s_j \in Trans(s_{j-1}))\}$$

Gen associates each *Promela* model M with the set of all possible state sequences that M can display in different model executions. This semantics is useful for our purposes due to each sequence corresponding to a possible model execution that we must analyze when we are carrying out model checking. In

addition, since the meaning of the model operations is unspecified, it is possible to change this without modifying the operational behavior of the model. As the operational behavior of a given model $M \in Promela$, represented by $Gen(M)$, depends on the functions *eval* and S, we denote it with $Gen(M, eval, S)$, *eval* and S being the parameters of the generalized semantics.

3 Abstract generalized semantics

In this section, we explain how to obtain an abstract model from an original model. The meaning of the abstract model is given by the generalized semantics defined above to which we have added two conditions to guarantee correctness. To do the abstraction, we assume the existence of an abstraction function which transforms actual data and operations into abstract ones. An automatic source-to-source transformation will result in an abstract model. The object model of the transformation will be obtained from the correct implementation of the abstract instructions produced by the data abstraction. The benefit of this method is that the abstract model obtained is a Promela model which can be analyzed by the SPIN tool.

We first explain the initial step to obtain abstract models given the concrete ones, then present the abstract semantics (generalized semantics + correctness conditions), prove some correctness results, and finally present how to obtain correct abstract models in Promela.

3.1 Automatic transformation from concrete models to abstract models

Next, we use the following definitions. As before, let $M = P_1||\ldots||P_n$ be a Promela model involving the concurrent execution of n processes.

- Let us suppose that M contains s global and local variables $v_1, \ldots, v_s$, each one ranging over a (non-empty) set of values D_i. Let D be $D_1 \times \ldots \times D_s$.
- Let us suppose that M contains m channels $c_1, \ldots, c_m$, and let C_j be the domain of all possible values that c_j may store. Let C be $C_1 \times \ldots \times C_m$.
- Finally, let $Inst$ be $Inst_1 \times \ldots \times Inst_n$, $Inst_j$ being the set of instructions which form P_j.

From these definitions, we have $State = D \times C \times Inst$. Let $\alpha = \alpha_d \times \alpha_c : D \times C \rightarrow D^* \times C^*$ be an abstraction function which transforms each concrete state into an abstracted one. Note that temporarily the instructions are not being taken into account. We assume that $(D^*, \leq_d^*)$ and $(C^*, \leq_c^*)$ are posets, where the partial order represents the relative precision of the approximation of every abstract data, as is classic in abstract interpretation. Sometimes, we will use α_d and α_c over simple variables and channels instead of tuples. We allow this abuse of notation for clarity in the exposition.

From α_d and α_c, we define an approximation of the instructions. α_{inst} denotes the function which transforms every concrete instruction into an abstract one,

by renaming the original instruction and changing data and messages by the corresponding abstract ones using α. Let $Inst^*$ be $Inst^*_1 \times \ldots \times Inst^*_n$, each $Inst^*_j$ being the set of abstract instructions of the process P_j. Finally, let M^* be the abstract model obtained by substituting each instruction i of P_j by $\alpha_{inst}(i)$.

$\alpha = \alpha_d \times \alpha_c$ and α_{inst} define an abstraction of the model states, denoted by α_s, in the following way:

$\alpha_s((g, l, c, i_1, \ldots, i_n)) = (\alpha_d((g, l)), \alpha_c(c), \alpha_{inst}(i_1), \ldots, \alpha_{inst}(i_n))$

Given two abstract states $s^*_1 = (g^*_1, l^*_1, c^*_1, inst^*_1)$ and $s^*_2 = (g^*_2, l^*_2, c^*_2, inst^*_2)$, we say that $s^*_1 \leq^*_s s^*_2$ (s^*_1 is more precise than s^*_2) iff the following conditions hold: (1) $(g^*_1, l^*_1) \leq^*_d (g^*_2, l^*_2)$, (2) $c^*_1 \leq^*_c c^*_2$ and (3) $Inst^*_1 = Inst^*_2$.

Figure 2 illustrates the first part of the transformation method guided by an abstraction presented above. Model M has only one process (*init*) and one global variable i ranging over the integer numbers. We consider the classic abstraction $\alpha : int \rightarrow \{\bot, even, odd, evenodd\}$ defined as $\alpha(2n) = even$, and $\alpha(2n+1) = odd$, for all $n \geq 0$. $\bot$ and *evenodd* are the bottom and top of the domain, respectively. The partial order defined on this domain is $\forall x \in \{\bot, even, odd, evenodd\}.(\bot \leq x \leq evenodd)$.

```
init { int i;
   start:  if
           :: i = i + 1; goto start;
           :: goto end;
           fi
   end:skip }


    Model M
```

```
#define even 0
#define odd 1
#define evenodd 2
init { int i =* even;
   start: if
             :: i =* i ++ odd;goto start;
             :: goto end
            fi;
 end:skip }


Model M*
```

Fig. 2. Concrete and abstract models

In the transformed model M^*, the constants 0 and 1 have been substituted by *even* and *odd*, respectively. In addition, the operation $+$ and the instruction $=$ have been replaced by $+^*$ and $=^*$, though they are not yet defined. The automatic transformation from M to M^* can be easily made, and only the definition of the abstract operations and instructions have been left out. In Section 3.4, we complete the transformation by substituting each abstract operator and each abstract instruction by a Promela code, which is a correct approximation, with respect to the semantics defined in the following section, of the respective concrete operation and instruction.

3.2 Abstract semantics

Now we define the abstract semantics of a Promela model. Given a model M whose semantics is given by $Gen(M, eval, S)$ and an abstraction function α defined as in Section 3.1, we construct the abstract model M^* using α as exposed above. The abstract generalized semantics $Gen^*(M^*, eval^*, S^*)$ of M^* is defined as Gen. Next we use the superindex * to refer to the abstract states, abstract sequences, and other elements of the generalized semantics of the abstract model M^*. We impose the following conditions on the meaning of the

abstract operations and instructions ($eval^*$ and S^*), to assure the correctness of the transformation $M \rightarrow M^*$:

1. Let us suppose that $eval^* : BoolExp^* \times State^* \rightarrow \{false, true\}$ verifies the following correctness relation: $\forall s^* \in State^*.(\forall s \in State.\alpha_s(s) \leq_s^* s^* \Rightarrow eval(exp, s) \leq_b^* eval^*(\alpha_{inst}(exp), s^*)))$.
   The partial order used in the set $\{false, true\}$ is $false \leq_b^* true$, which in our context means that if $eval^*$ returns $false$ then the evaluation of $\alpha_{inst}(exp)$ in each concretization of the abstract state s^* is $false$; otherwise $eval^*$ will return $true$. Thus $eval(exp, s) = false \not\Rightarrow eval^*(\alpha_{inst}(exp), \alpha_s(s)) = false$, however $eval^*(exp^*, s^*) = false \Rightarrow eval(exp, s) = false$, for all Boolean expressions exp and states s such that $\alpha_{inst}(exp) = exp^*$ and $\alpha_s(s) \leq_s^* s^*$.
2. Let $S^* : (Inst_1^* \cup \ldots \cup Inst_n^*) \times State^* \rightarrow State^*$ be an abstract function verifying the relation: $\forall s^* \in State^*.(\forall s \in State.(\alpha(s) \leq_s^* s^* \Rightarrow \alpha(S(i, s)) \leq_s^* S^*(\alpha_{inst}(i), s^*)))$.
   S^* gives abstract and correct meaning to the instructions of the abstract model M^*. As before, S^* is not specified, we have only declared its correctness relation with S.

3.3 Correctness

In this section, we prove a correctness result of the abstraction. We impose the condition that every state in each concrete execution path corresponds to an abstract state in an abstract execution path. This is a strong result as we need the whole concrete computation to be simulated step-by-step by the abstraction. Weaker correctness conditions can be defined by imposing the condition that the approximation holds at some specified points of the concrete model (and not in all states) in a similar way to the collecting semantics used in abstract interpretation [2].

In the following, let us suppose that $\alpha = \alpha_d \times \alpha_c : D \times C \rightarrow D^* \times C^*$ is an abstraction function of a concrete Promela model $M = P_1 || \ldots || P_n$, in the same conditions as the previous discussion, $Gen(M, eval, S)$ and $Gen^*(M^*, eval^*, S^*)$ being the semantics of M and its transformation M^*.

Given $seq = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow \ldots \in Sequence$, we define $\alpha_{seq}(seq)$ as $\alpha_s(s_0) \rightarrow \alpha_s(s_1) \rightarrow \ldots \rightarrow \alpha_s(s_k) \rightarrow \ldots \in Sequence^*$.

Given a channel c, let $|c| \in N$ denote the number of messages stored by c in a particular state s. In addition, if $|c| = n$ and $1 \leq j \leq n$, c_j represents the j-th message.

Definition 1. *Given seq^*, $seq^{*'} \in Sequence^*$, then $seq^* \leq_{seq}^* seq^{*'}$, iff for all $i \geq 0$, $s_i^* \leq_s^* s_i^{*'}$.*

Definition 2. *An abstraction α preserves the length of the channels iff for each channel c, $|c| = |\alpha_c(c)|$.*

Lemma 1. *If α is an abstraction which preserves the length of the channels, then $\forall i \in Inst.(\forall s^* \in State^*.(\forall s \in State.(\alpha_s(s) \leq_s^* s^*, exec_j^*(\alpha_{inst}(i), s^*) = false \Rightarrow exec_j(i, s) = false)))$.*

Proof. Let us consider $s \in State$ and $s^* \in State^*$ such that $\alpha_s(s) \leq_s^* s^*$.

1. If i is a Boolean expression then, by hypothesis $eval^*(\alpha_{inst}(i), s^*) = false$, and since $eval^*$ verifies $eval(i, s) \leq_b^* eval^*(\alpha_{inst}(i), s^*)$, then we deduce that $eval(i, s) = false$, that is, $exec_j(i, s) = false$.
2. If $i = c?msg$ then we consider two cases:
   (a) i imposes no condition over the message read. In this case, as α preserves the length of the channels $|c| = |\alpha_c(c)|$, and as $exec_j^*(\alpha_{inst}(i), s^*) = false$, we have $|c| = |\alpha_c(c)| = 0$, and hence $exec_j(i, s) = false$.
   (b) i imposes some matching condition over the message read from c. The case $|c| = 0$ has been proved in (a). Therefore, let us assume that $|c| > 0$. As $exec_j^*(a_{inst}(i), s^*) = false$, the first abstract message of $\alpha_c(c)$, $\alpha_c(c)_1$, does not verify this condition, which really is the Boolean test $\alpha_c(c)_1 ==^* \alpha_d(msg)$. Thus, by 1, as $eval^*(\alpha_c(c)_1 ==^* \alpha_d(msg), s^*) = false$, we deduce that $eval(c_1 == msg, s) = false$ or equivalently that $exec_j(i, s) = false$.
3. If $i = c!msg$ as before, if α preserves the length of the channels then $exec_j^*(\alpha_{inst}(i), s^*) = false \Rightarrow exec_j(i, s) = false$.
4. No other instruction suspends in the concrete model or in the abstract one. The instruction if is analyzed using cases 1, 2 and 3. [1]

In the rest of the section, we always assume that α preserves the length of the channels.

Lemma 2. *If $s_1, s_2 \in State$ and $s_1^* \in State^*$ verify that $s_2 \in Trans(s_1)$ and $\alpha_s(s_1) \leq_s^* s_1^*$ then an abstract state $s_2^* \in State^*$ exists such that $s_2^* \in Trans^*(s_1^*)$ and $\alpha_s(s_2) \leq_s^* s_2^*$.*

Proof. $s_2 \in Trans(s_1)$ implies that $j \in \{1, \ldots, n\}$ and $i \in Inst_j$ exist such $s_2 = S(i, s_1)$. This means that $exec_j(i, s_1) = true$, and applying Lemma 1 we have that $exec_j^*(\alpha_{inst}(i), s_1^*) = true$. So, we can choose the instruction $\alpha_{inst}(i) \in Inst_j^*$ to evolve from s_1^* to $S^*(\alpha_{inst}(i), s_1^*)$. Let s_2^* be $S^*(\alpha_{inst}(i), s_1^*)$. By definition, $s_2^* \in Trans^*(s_1^*)$, and also by the correctness condition (point 2) we have $\alpha(S(i, s_1)) \leq_s^* S^*(\alpha_{inst}(i), s_1^*)$, that is, $\alpha(s_2) \leq_s^* s_2^*$.

Lemma 3. *$\forall s \in State.(\forall s^* \in State^*.(\alpha_s(s) \leq_s^* s^*, Trans(s) = end \Rightarrow Trans^*(s^*) = end))$.*

Proof. If $\alpha(s) \leq_s^* s^*$ then the process counters of both states (s and s^*) are pointing to the same concrete or abstract instructions. So, $Trans(s) = end$ means that all the counters in s are pointing to the end of each model process and the same for s^*.

Theorem 1. *Let $Gen(M, eval, S)$ and $Gen^*(M^*, eval^*, S^*)$ be the semantics of the models M and M^* verifying the conditions presented in Section 3.2, then for each deadlock-free sequence $seq \in Gen(M, eval, S)$, an abstract deadlock-free sequence $seq^* \in Gen^*(M^*, eval^*, S^*)$ exists, such that $\alpha_{seq}(seq) \leq_{seq}^* seq^*$.*

[1] We assume that there is not rendezvous in the model M. It is possible to include it but this would unnecessarily complicate the presentation.

Proof. Let *seq* be $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \in Gen(M, eval, S)$, using Lemma 2 we build seq^* as follows:

1. Let s_0^* be $\alpha_s(s_0)$ the initial state of every execution path of M^*.
2. Let s_1^* be the abstract state given by Lemma 2 using $s_0^* \in State^*$, and s_0, $s_1 \in State$. $s_1^* \in State^*$ verifies that $s_1^* \in Trans^*(s_0^*)$ and $\alpha(s_1) \leq_s^* s_1^*$.
3. Applying Lemma 2, we successively obtain the abstract states s_2^*, s_3^*, etc.
4. By Lemma 3, if *seq* is a finite sequence, we have that seq^* is also finite.

Definition 3. *Given two Promela models M, $M^* \in Promela$, then we say that M^* α-approximates M, and we denote it with $M \sqsubseteq_\alpha M^*$, if $Gen(M, eval, S)$ and $Gen^*(M^*, eval^*, S^*)$ are related as explained in Section 3.2, and verify the hypotesis of Theorem 1. In particular, α preserves the length of the channels and M is deadlock-free.*

The next propositions explain the relationship between the concrete and abstract models when proving temporal properties. One property F over one model M is built with the usual temporal operators, Boolean connectives and propositions. Propositions are tests over data, channels or labels. For convenience, we assume that all formulas are in negation normal form, that is, negations only appear in propositions. In the following, we call abstract properties to those which are defined over abstract models (propositions evaluated over abstract states). Given a concrete property F (or proposition P) over a model M, we can obtain its abstract version, denoted by $\alpha_f(F)$ ($\alpha_f(P)$), by preserving the formula structure (Boolean and temporal operators) and abstracting data.

Definition 4. *Let us assume that $M \sqsubseteq_\alpha M^*$. Let P^* be an abstract proposition, G^*, H^* and F^* abstract temporal formulas and $seq = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow \in Gen(M, eval, S)$, then*

1. *$s \in State$ satisfies P^* ($s \models P^*$) iff an abstract state $s^* \in State^*$ exists such $\alpha_s(s) \leq_s^* s^*$ and $s^* \models P^*$.*
2. *$seq \models []G^*$ iff for each $i \geq 0$, $s_i \models G^*$.*
3. *$seq \models \diamond G^*$ iff $i \geq 0$ exists such that $s_i \models G^*$.*
4. *$seq \models G^* U H^*$ iff $i \geq 0$ exists such that $s_i \models H^*$ and for all $0 \leq j \leq i-1, s_j \models G^*$.*
5. *$seq \models G^* \wedge H^*$ iff $seq \models G^*$ and $seq \models H^*$.*
6. *$seq \models G^* \vee H^*$ iff $seq \models G^*$ or $seq \models H^*$.*
7. *$M \models F^*$ iff for all execution paths $seq \in Gen(M, eval, S)$, $seq \models F^*$.*

Proposition 1. *Let F^* be a universal abstract property over M^* (which must hold on all execution paths). If $M \sqsubseteq_\alpha M^*$ and $M^* \models F^*$ then $M \models F^*$.*

Proof. By induction over the formula structure using Theorem 1 (see [10]).

This proposition proves that if the abstract model M^* verifies an abstract property F^*, the concrete model M also verifies F^*. If the abstraction function α and the abstract domain $D^* \times C^*$ define useful information for the user, this

result can be a powerful means for debugging the model. For instance, with the typical abstraction $\alpha : int \rightarrow \{\perp, even, odd, evenodd\}$ defined in Section 3.1, it could be interesting to verify $F^* = [](x == even)$, assuming that x is a global variable of a model M. However, the user could also be interested in proving a concrete formula F (over the concrete domain) by proving $\alpha_f(F)$ (or other abstract version F^* of F) over M^*. Next, we discuss how to relate F^* and F.

Definition 5. *Given P_1^*, P_2^*, F_1^* and F_2^* abstract propositions and formulas*

1. $P_1^* \Rightarrow^* P_2^*$ *iff* $\forall s \in State.(s \models P_1^* \Rightarrow s \models P_2^*)$.
2. $F_1^* \Rightarrow^* F_2^*$ *iff* $\forall seq \in Gen(M, eval, S).\ (seq \models F_1^* \Rightarrow seq \models F_2^*)$.

Proposition 2. *Given an abstract proposition P^* and a state $s \in State$, if $s \models P^*$ then a concrete proposition P exists such that $s \models P$ and $\alpha_f(P) \Rightarrow^* P^*$.*

Proof. If $s \models P^*$ then $\exists s^* \in State^*.(\alpha_s(s) \leq_s^* s^* \text{ and } s^* \models P^*)$. This means that $\alpha_s(s)$ verifies a stronger version Q^* of P^* (the data and the contents of channels of $\alpha_s(s)$ are more precise than the ones of s^*). So, s verifies the proposition P obtained by substituting the abstract values of Q^* in $\alpha_s(s)$ by the corresponding concrete data in s. By construction, $\alpha_f(P) = Q^*$ and $\alpha_f(P) \Rightarrow^* P^*$.

Proposition 3. *Given a model M and an abstract property F^*, if $M \models F^*$ then $M \models \vee\{F : \alpha_f(F) \Rightarrow^* F^*\}$.*

Proof. By induction over the formula structure using Proposition 2 (see [10]).

This proposition gives us the relationship between the abstract formula proved in the abstract model and the concrete formulas which hold in the concrete model. This result is very useful when the set $\{F : \alpha_f(F) \Rightarrow^* F^*\}$ has only one element, this is, when there is only one concrete formula F, such that $\alpha_f(F) \Rightarrow^* F^*$. Under this condition, the user knows that F holds in M. So, when the objective of one abstraction is to prove a concrete formula F over the concrete model M, we must define the abstraction function α in such a way that only one concretization of $\alpha_f(F)$ exists. If this happens and the system proves that $M^* \models \alpha_f(F)$, Proposition 3 guarantees that $M \models F$, as was expected by the user. One example of this situation is shown in [4]. The user wants to prove the formula $F = [](a = 42 \rightarrow b = 42)$ over a model M (with only two global variables a and b) and chooses the abstraction function $\alpha : Int \rightarrow \{0, 1\}$ defined as $\alpha(42) = 0$, $\alpha(i) = 1$, if $i \neq 42$ to build the abstract model M^*. With this definition if $\alpha_f(F) = F^* = [](a = 0 \rightarrow b = 0)$ and $M^* \models \alpha_f(F)$ we have that $M \models F$, as expected by the user.

In the previous discussions, we have assumed that the concrete model M is deadlock-free. The next Proposition studies how to analyze deadlock in M. For this purpose we need to impose some conditions which are presented in the following definition.

Definition 6. *Given an abstraction function α, we say that α verifies the executability conditions iff $exec_j(i, s) = exec_j^*(\alpha_{inst}(i), s^*)$, for each pair of states $s^* \in State^*$, $s \in State$, such that $\alpha_s(s) \leq_s^* s^*$, and for each $i \in Inst_j$.*

Proposition 4. *Let M^* be an abstraction of M obtained by means of the function α, verifying the conditions presented in Section 3.2. Let us assume that α also verifies the executability conditions presented above. If $Gen^*(M^*, eval^*, S^*)$ has no execution sequence which deadlocks then $Gen(M, eval, S)$ has no deadlock either.*

Proof. Given $seq = s_0 \rightarrow \ldots \rightarrow s_k \rightarrow deadlock \in Gen(M, eval, S)$ by Theorem 1, we can construct an abstract sequence $s_0^* \rightarrow \ldots \rightarrow s_k^*$ such that $\forall 0 \leq i \leq k.(\alpha_s(s_i) \leq_s^* s_i^*)$. Let us consider the states s_k and s_k^*. By hypothesis, $Trans(s_k) = deadlock$, this is, $\forall j = 1, \ldots, n$, if $i_j = just_exec_j(s)$ then $exec_j(next_inst_j(i_j), s_k) = false$, and by Definition 6 we have
$(exec_j^*(next_inst_j^*(\alpha_{inst}(i_j), s_k^*)) = false)$, that is, $Trans^*(s_k^*) = deadlock$.

3.4 Model transformation based on abstract interpretation

Proposition 5. *Let $M = P_1|| \ldots ||P_n$ be a Promela model and $\alpha = \alpha_d \times \alpha_c : D \times C \rightarrow D^* \times C^*$ an abstraction function verifying the conditions of Section 3.1 (α preserves the length of the channels). Let $Inst^*$ be the set of abstract instructions derived from α. Let M^* be the model obtained by abstracting all the constants and instructions of M. Let us suppose that for each instruction $i^* \in Inst^*$ a Promela implementation exists verifying the correctness conditions imposed in Section 3.2. Under these conditions, the model MI^*, obtained by atomically substituting each model instruction by its implementation, verifies Theorem 1.*

Example in Figure 2 (continuing): The model obtained by substituting $+^*$ and $=^*$ by its implementation is:

```
#define even 0
#define odd 1
#define evenodd 2
init {  byte i = even;
start:    if
          :: d_step { if
                      ::i == evenodd -> i=evenodd
                      ::i==even -> i=odd
                      ::i==odd -> i=even
                      fi };goto start;
          :: goto end;
          fi;
end:skip;
}              Model MI*
```

4 Example

We illustrate the concepts presented above by verifying the version of the ABP protocol given in Figures 3 and 4. This version of the protocol enables the transmission of data from process `A` to process `B` over an unreliable channel. The error-free behavior of the protocol consists in `A` sending data message with a control bit equal to `1` and receiving a reply message also with bit `1` (states `S1`

and S2 in A). In this error-free scenario, B waits for data messages with bit 1 and replies with the same bit (states S2 and S1 in B). Other states in A and B are used to recover the transmission from an error in the channel. Figure 3 shows the state machines for every process in the protocol (extracted from SPIN documentation). Our version for this protocol also contains the process C to model an unreliable channel between A and B. This process controls the maximum number of errors in the line in order to preserve the correct behavior of the protocol.

In order to ensure the correctness of the protocol, the user can use SPIN to analyze at least the two following properties.

Invalid end states. The intended behavior of process A is to send data messages infinitely often, so there are no end labels in the model to specify legal end states in process A or B, and every deadlock should be reported as an invalid end state.

Temporal property. As suggested in SPIN documentation, the designer of the protocol would like to know if it has the following property: "In all computations, every message sent by A is received error-free at least once and accepted at most once by B." Following the idea of Holzmann's assertion, this property can be written with the temporal formula

$F =$ `[] (B_accept -> B_accept_ok)`

where the informal meaning of the propositions B_accept is that process B will immediately accept the data received in mr, and B_accept_ok means that the expected value in mr follows the value stored in lmr.

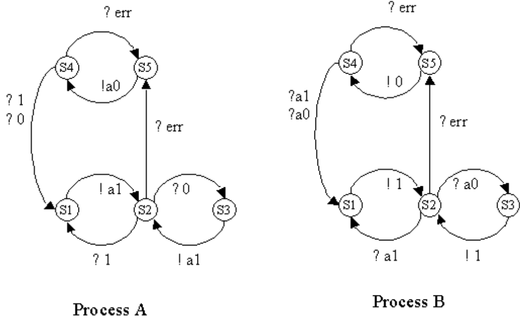

Fig. 3. State machines for the ABP protocol

The state-space of the model in Figure 4 (with 4 data to be sent) has 955 states. This space state can be obviously explored with current computers in order to ensure the desired correctness requirements. But let us assume that we do not have enough memory in the computer. Then, the use of abstractions could simplify our work.

As explained in the previous section, the construction of a (more) abstract model for a given model must be driven by the property to be verified. If we examine the meaning of the propositions in F, we see that the evaluation of

B_accept does not depend directly on data values. So we choose an abstraction function that makes easier the evaluation B_accept_ok. The model $M1^*$ is an abstraction of *M1* replacing the real data value in the message with only two abstract values: *even* and *odd*. This means that the global variables mt, mr and lmr and the variable m in process C will range over $\{\perp$ *,even, odd, evenodd* $\}$. The abstract model Promela $M1^*$ is obtained by transforming the assignment sentence and the sum operations in processes A and B (in a similar way that in Section 3.4).

```
#define MAX 4
byte mt, mr, lmr;
proctype A(chan in, out)
{      byte  vr;
S1:    mt = (mt+1)%MAX;
       out!mt,1;
       goto S2;
S2:    in?vr;
       if
       :: (vr == 1) -> goto S1
       :: (vr == 0) -> goto S3
       :: (vr == error) -> goto S5
       fi;
S3:    out!mt,1;
       goto S4;
S4:    in?vr;
       if
       :: goto S1
       :: (vr == error)-> goto S5
       fi;
S5:    out!mt,0;
       goto S4
}
```

```
proctype B(chan in, out)
{       byte ar;
        goto S2;
        S1: lmr = mr;
        out!1;
        goto S2;
 S2:    in?mr,ar;
        if
        :: (ar == 1) -> goto S1
        :: (ar == 0) -> goto S3
        :: (ar == error)-> goto S5
        fi;
 S3:    out!1;
        goto S2;
 S4:    in?mr,ar;
        if
        :: goto S1
        :: (ar == error)-> goto S5
        fi;
        S5: out!0;
        goto S4
}
```

```
proctype C(chan a2c, c2b, b2c, c2a)
{ byte m, v,errors_a = 0, errors_b = 0;
S1:if
   :: a2c ? m,v -> if
                   :: atomic{errors_b=0; c2b ! m, v }-> goto S1
                   :: errors_b < 1 -> atomic{errors_b++; c2b ! m, error }-> goto S1
                   fi
   :: b2c ? v ->   if
                   :: atomic{errors_a=0; c2a ! v}-> goto S1
                   :: errors_a < 1 -> atomic{errors_a++; c2a ! error }-> goto S1
                   fi
   fi }

init { chan b2c = [2] of {byte}, c2a = [2] of {byte},
           a2c = [2] of {byte, byte}, c2b = [2] of { byte, byte };
       atomic {run A(c2a, a2c);run C(a2c, c2b, b2c, c2a); run B(c2b, b2c) } }
```

Fig. 4. Promela model of ABP (M1)

This abstraction verifies the hypothesis of Proposition 4, because we do not abstract any sentence that can suspend in *M1*, and the abstraction preserves the length of the channels. So we can employ $M1^*$ to verify the absence of deadlock in the model *M1*. This property is proved by inspecting 232 states in $M1^*$.

The verification of *M1* requires 448 states. The abstract propositions in F^* are defined as: `#define B_accept B[3]@S1` and `#define B_accept_ok mr != lmr`

We can easily prove that $M1 \sqsubseteq_\alpha M1^*$, and by Proposition 1, if $M1^* \models F^*$ then $M1 \models F^*$. So we only have to prove $M1 \models F^*$. Fortunately, SPIN checks that F^* is valid in all computations of $M1^*$ exploring only 106 states.

If we want to verify a concrete formula with a similar meaning over *M1*, such as

F' = `[] ((B[3]@S1 -> mr == (lmr +1) % MAX)`

`MAX` being the real number of different data messages, then we have to explore 448 states in *M1*. So we have saved memory using the abstract model.

	M1	*M1**	*M2*	*M2**
State space	448	232	952	297
Deadlock ? Explored States	No 448	No 232	No 952	No 297
Valid F ? Explored States	Yes 448	Yes 106	No 281	No 149

Table 1. Verification results

Table 1 shows all these results with *M1* and $M1^*$. The table also shows the verification of the models *M2* and $M2^*$. The model *M2* is similar to the original in the SPIN documentation (any number of errors), but with an intermediate process to model the channel. $M2^*$ is obtained with the same abstraction as $M1^*$. Results for *M2* and $M2^*$ show that violation of F^* over $M2^*$ does not provide a definitive result about its violation over *M2*. But deadlock can be directly analyzed in the abstract model.

5 Conclusions and further work

We have presented a generalized semantics of Promela which is suitable for justifying the use of abstract interpretation to automatically transform Promela models into more abstract versions. The transformed (abstracted) models can be employed to reduce the state space for the verification using SPIN. The same framework can be used to define the semantics of the original (concrete) Promela model as well as the new (abstracted) models, therefore, it allows us to relate different abstractions of the same system. The way of using abstract interpretation enables the verification of universal temporal properties of the original system by verifying a more abstract version, as in other related works. We also are working on the use of program specialization (partial evaluation) in order to prove existential properties, i.e., properties that hold along one execution path.

We are currently working on tools to integrate the abstract interpretation technique into XSPIN. Our aim is to put this technique into everyday use by

constructing a user-friendly environment that provides a library of standard abstraction functions.

Acknowledgement. We would like to thanks the anonymous referees for their useful comments and suggestions on this and earlier versions of the paper.

References

1. Barlett A., Scantlebury R.A., Wilkinson P.T.: A note on reliable full-duplex transmission over half-duplex lines. Communications of the ACM, 12 (5)(1969) 260–265
2. Cousot P., Cousot R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Conference Record of the 4th ACM Symposium on Principles of Programming Languages (1977)
3. Clarke E.M., Emerson E. A., Sistla A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. on Programming Languages and Systems, 8 (2), (1986) 244–263
4. Clarke E.M., Grumberg O., Long D.E.: Model Checking and Abstraction. ACM Transaction on Languages and Systems, 16(5) (1994) 1512–1542
5. Clarke E.M., Wing J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, ACM 50TH Anniversary Issue Workshop on Strategic Directions in Computing Research, 28(4) (1996)626–643
6. Dams D., Gerth R., Grumberg O.: Abstract Interpretation of Reactive Systems. ACM Transactions on Programming Languages and Systems, 19(2) March (1997) 253–291
7. Giacobazzi R., Debray S.K., Levi G.: A Generalized Semantics for Constraint Logic Programs. Proceedings of the International Conference on Fifth Generation Computer Systems (1992) 581–591
8. Gunter C., Mitchell J.: Strategic Directions in Software Engineering and Programming Languages. ACM Workshop on Strategic Directions in Computing Research, ACM Computing Surveys, 28(4)(1996) 727–737
9. Gallardo M.M., Merino P., Troya J.M.: Relating Abstract Interpretation with Logic Program Verification. Proceedings of the International Workshop on Verification, Model Cheking and Abstract Interpretation, Port Jefferson, USA, (1997)
10. Gallardo M.M., Merino P.: A Formal basis to Improve the Automatic Verification of Concurrent Systems. Technical Report LCC IT 99/10. Dpto. de Lenguajes y Ciencias de la Computacion. University of Malaga.(1999)
11. Holzmann G.J.: Design and Validation of Computer Protocols. Prentice-Hall, New Jersey (1991)
12. Holzmann G.J.: Designing Bug-Free Protocols with SPIN. Computer Communications, 20(2) (1997) 97–105
13. Jones, N.D., Søndergaard, H.: A Semantics-based framework for the abstract interpretation of Prolog. Abstract Interpretation of Declarative Languages. S. Abramsky and C. Hankin, Eds. Ellis Horwood, Chichester, U.K. (1987) 123–142
14. Natarajan V., Holzmann G.J.: Outline for an Operational Semantics of Promela. The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS Vol. 32 (1997) 133–152
15. Vardi M., Wolper P.: An Automata-Theoretic Approach to Automatic Program Verification. Proc. of the Symp. on Logic in Computer Science, Cambridge (1986)