7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS’02)

University of Málaga, Spain
July 12-13, 2002

Rance Cleaveland, Hubert Garavel (Eds.)

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Local Organizing Commitee:
María del Mar Gallardo, Pablo López,
Jesús Martínez, Pedro Merino
Programme Committee

- T. Arts (Ericsson, S)
- M. Bernardo (Univ. of Urbino, I)
- R. Cleaveland, co-chair (SUNY and Reactive Systems, USA)
- W.J. Fokkink (CWI, NL)
- H. Garavel, co-chair (INRIA Rhone-Alpes, F)
- S. Gnesi (CNR/IEI Pisa, I)
- P. Godefroid (Bell Labs, USA)
- H. Hermanns (Univ. Twente, NL)
- T. Margaria (METAFrame Technologies, D)
- P. Merino Gómez, local organization chair (Univ. Málaga, E)
- I. Schieferdecker (GMD Berlin, D)
- S. Schneider (Royal Holloway, University of London, UK)
- M. Sighireanu (University of Paris-7 Jussieu, F)
- R. de Simone (INRIA Sophia Antipolis, F)
- U. Ultes-Nitsche (University of Southampton, UK)
- A. Valmari (Tampere University of Technology, Fi)
- W. Visser (RIACS/NASA Ames, USA)

Additional Reviewers

- Bahareh Badban
- Clara Benac Earle
- Tommaso Bolognesi
- Antonio Cerone
- Kousha Etessami
- Alessandro Fantechi
- Natalia Ioustinova
- Frederic Lang
- Izak van Langevelde
- Michael Leuschel
- Pablo Lopez
- Cecilia Mascolo
- Mieke Massink
- Radu Mateescu
- Simona Orzan
- Jun Pang
- Laurence Pierre
- Simon St James
- Laurent Thery
- Mikko Tiusanen

Local Organizing Committee

Software Engineering Group, University of Málaga

- M. del Mar Gallardo
- P. López
- J. Martínez
- P. Merino, local organization chair
# Table of Contents

**INVITED PRESENTATION**

Abstraction for Software Model-Checking ................................................................. 1  
*Andreas Podelski (Max-Planck Institut für Informatik, Germany)*

**SESSION 1 - ABSTRACTION AND MODEL-CHECKING**

Predicate Abstraction and Refinement for Model Checking VHDL State Machines ........ 3  
*M. Bourahla, M. Benmohamed*

A Tool for Abstraction in Model Checking ............................................................... 19  
*M. del Mar Gallardo, J. Martinez, P. Merino, E. Pimentel*

**SESSION 2 - TESTING**

Validation and Automatic Test Generation on UML Models: The AGATHA Approach .... 35  
*D. Lugato, C. Bigot, Y. Valot*

Heuristic-Driven Techniques for Test-Case Selection ............................................. 51  
*J.C. Burguillo, M. Llamas, M.J. Fernández, T. Robles*

Scalable System level CTI Testing through Lightweight Coarse-grained Coordination .... 67  
*T. Margaria, B. Steffen*

**INVITED PRESENTATION**

Authenticity Types for Cryptographic Protocols ..................................................... 85  
*Andrew D. Gordon (Microsoft Research, Cambridge, UK)*

**SESSION 3 - INDUSTRIAL CASE STUDIES I**

A Methodological Process for the Design of a Large System: Two Industrial Case-Studies .... 87  
*N. Lopez, M. Simonot, V. Donzeau-Gouge*

Automatic Verification of the IEEE-1394 Root Contention Protocol with KRONOS and PRISM ................................................................. 107  
*C. Daws, M. Kwiatkowska, G. Norman*
INVITED PRESENTATION

Synthesis of Verified Real Time Software
Wang Yi (Uppsala University, Sweden)

SESSION 4 - INDUSTRIAL CASE STUDIES II

Specification and Analysis of the MPEG-2 Video Encoder with Timed-Arc Petri Nets
V. Valero, F.L. Pelayo, F. Cuartero, D. Cazorla

Properties of the Subtraction Valid for any Floating Point System
S. Boldo, M. Daumas

SESSION 5 - MODEL CHECKING

Simple and Efficient Translation from LTL Formulas to Buchi Automata
X. Thirioux

Liveness Checking as Safety Checking
A. Biere, C. Artho, V. Schuppan

Stuttering-Insensitive Automata for On-the-Fly Detection of Livelock Properties
H. Hansen, W. Penczek, A. Valmari

Context-Sensitive Visibility
A. Valmari, H. Virtanen, A. Puhakka
A Tool for Abstraction in Model Checking

María del Mar Gallardo, Jesús Martínez
Pedro Merino, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, 29071 Málaga, Spain

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 model checking of more complex systems. Nevertheless, there is a lack of tools actually supporting this technique. 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. The abstraction of the model should be followed by the abstraction of the temporal formulas to be checked. This paper presents $\alpha$Spin, a tool for the integration of several abstraction approaches (for models and formulas) into the well known model checker Spin. In particular, $\alpha$Spin integrates two dual approaches, the classic abstraction method, based on under-approximating properties, and an alternative approach, proposed by the authors, where abstraction provides an over-approximation of the formulas.\footnote{Work supported by projects TIC99-1083-C02-01 and TIC2001-2705-C03-02}

Key words: Model Checking, Abstraction, Tools, Spin

1 Introduction

Computer based verification methods, such as model checking \cite{1}, have become realistic techniques to be used in the development of critical systems. However, model checking is only fruitful when a useful model of a system is available. By useful we mean an abstract representation of the system, containing only the details which ensure that satisfaction (non-satisfaction) of certain properties provides us with information about the actual behavior of the system. Models describing an excess of details may produce the state explosion problem, which could prevent the use of current tools to fully analyze them. This problem affects both the symbolic method, and explicit model checking; both of them employ ideas of abstract interpretation \cite{5} to construct abstract state spaces or models \cite{2,6,4,13}. Whereas most proposals to implement abstraction focus

\footnote{Email: gallardo,jmcruz,pedro,ernesto@lcc.uma.es}
on the symbolic approach, there is a great demand for tools for the second approach. This paper presents a method to extend explicit model checkers with abstraction. Although our technique can be applied to different tools, we describe αSpin, an implementation on top of Spin [16,17].

Extending a model checker with automatic abstraction should improve some of the classical steps enumerated by Clarke et al. in [2]: a) defining one abstraction function suitable for the temporal property to be verified, b) constructing the abstract model, and c) relating the verification results to the behavior of the original (concrete) model.

As regards step (a), we propose the use of an abstraction library with previously defined abstraction functions that can be used depending on the properties to be analyzed [10]. This idea is also employed in [9] and [13].

Our method to construct the abstract model (b) is based on syntactic transformation of the model and the formulas. This allows us to reuse the same tool (Spin) to verify the abstract model. The approach also gives us a framework to reason about correctness of the transformation, as discussed in [10]. Finally, we based the transformation on the use of XML [19] in order to be as independent as possible of the actual modelling language [12].

Regarding the relation of results (step c), the classic method to abstract temporal logic is based on defining an abstract satisfiability relation which under-approximates the standard one [2,6]. As a consequence, it is suitable to check whether a temporal formula is true for all execution paths (satisfaction of universal properties). We introduce a new approach based on over-approximation of temporal formulas [11], which can be employed to ensure that it is impossible for any path in the abstract model to satisfy a formula (refutation of existential formulas). Our experience suggests that, given a model, both dual methods can be efficiently employed in the verification work. We have thus integrated both techniques in our current implementation, αSpin. Furthermore, the implementation also allows us to explore how to obtain more benefits from the combination of both approaches in the same formula.

Abstraction by syntactic transformation is also supported by other tools, but they are mainly oriented to programming languages, and not to formal description techniques. Furthermore, these tools produce the abstract model in a different language, thus lacking the advantage of reusing the model checker. Tools like FeaVer [18], Bandera [9] or the first version of JPF [15] are considered abstraction tools for Spin because they produce (extract) PROMELA code from the source code (C, Java). We believe that αSpin is complementary to these tools because they deal with different problems. In model extraction, the major aim seems to be how to “remove” great amounts of code to obtain the PROMELA model. In our case, we start with a relatively simple model, and our work focuses on incrementally applying abstraction to the initial and the new PROMELA models. See [7] for an overview of abstraction techniques and associated tools.
The paper is organized as follows. Section 2 contains some preliminary background on Spin and its input languages. Sections 3 and 4 present the theoretical basis to support correct abstraction by transformation of PROMELA and temporal logic, respectively. In Section 5, we explain how to use αSpin with a previously published lift controller system as the case study [8]. Section 6 is devoted to the implementation details of αSpin. In the last section we discuss the main contributions of the work, and outline some future works.

2 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 [16,17]. 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 expressed with Linear Temporal Logic (LTL). It is also used as the platform to try new powerful algorithms to attack the state explosion problem.

2.1 Modelling with PROMELA

PROMELA is a language designed for describing systems composed of concurrent asynchronous communicating processes. A PROMELA model \( P = Proc_1 || \ldots || Proc_n \) consists of a finite set of 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 (see Fig. 1). A PROMELA process is defined as a sequence of possibly labelled sentences preceded by the declarative part (see example in Fig. 2). 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.

---

Fig. 1. Part of PROMELA Syntax

| Process ::= [active[“NumberOfInstances”]] | proctype ProcessTypeID {Decl; InstSeq} |
| InstSeq ::= [ || Inst{[ || Inst}]∗ | Inst ::= Basic|Jump|If|Do|Atomic|D_Step |
| Basic ::= BExp | Assign | Input | Output | Rendez |
| If ::= if BranchSeq fi | Jump ::= goto 1 | break |
| Atomic ::= atomic “{” InstSeq “}” | Do ::= do BranchSeq od |
| Input ::= ChannelId ? ExpSeq | D_Step ::= d_step “{” InstSeq “}” |
| Rendez ::= Input | ExpSeq |
| BranchSeq ::= Branch{Branch}∗ | Output ::= ChannelId ! ExpSeq |
| Branch ::= :: Inst |

21
of channels. In addition, PROMELA has other non-basic sentences like the non-deterministic If and Do sentences.

2.2 Temporal logic

Spin verifies LTL formulas against PROMELA models. Well-formed formulas of linear temporal logic (LTL) 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 "□", eventually "◇", next "O", and until "U". Formulas are interpreted with respect to model state sequences $t_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. Each sequence 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. 3 where $p$ is a proposition, and $f$ and $g$ are temporal formulas. For the sake of convenience, we assume that all formulas are in negation normal form, that is, negations only appear in propositions. Note that we have not included a rule defining the satisfaction of a negated formula. Instead, we treat the evaluation of negated propositions independently of their corresponding non-negated ones. The reason for this will be explained in Section 4. The two last rules in Fig. 3 define the semantics of the universal and existential temporal formulas. There, $M$ represents the set of execution traces produced by the model.

2.3 Spin

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 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 as a universal property. The same verification scheme can be employed to check whether a formula cannot be satisfied by any path (refutation of existential properties). These two ways of using LTL are presented in a user friendly interface called XSpin, as shown in Fig. 2. The first case corresponds to marking ”All Executions” and the second one to ”No Executions”.

Although there are many real examples where the verification can be done with standard exhaustive verification, Spin also implements optimization techniques to deal with complex systems. Partial order reduction replaces several interleaved sequences of events (sentences) by only one that represents the whole set. State compression reduces the use of memory by compressing the representation of the states without losing information. Bit-state hashing represents states as bits in a hash table, so in many cases the analysis is only partial. Our new tool preserves these optimization techniques.

3 Abstracting PROMELA

The first step for realizing abstract model checking is to reduce the model to be analyzed. In [10], we described a method based on the source-to-source transformation to abstract PROMELA models. The main idea in this work is that for abstracting models it suffices to replace the original type definitions (data and basic operations) by simpler ones, in such a way that the control part of the program (high level operations like non-determinism selection and loops, co-routines and so on) remain unchanged. From a practical point of view, this observation is very important, because it allows us to isolate the program points that must be changed when abstracting a model independently of the complexity of the language constructions. In addition, this modular vision facilitates the definition of abstractions, the analysis of the correctness of the abstraction and even the implementation.

In the rest of the section we summarize the theoretical background supporting the source-to-source transformation method. Let State denote the set of system states. We define functions \( \text{effect} : \text{Basic} \times \text{State} \rightarrow \text{State} \) and \( \text{test} : \text{BExp} \times \text{State} \rightarrow \{\text{false, true}\} \) which describe the effect of executing a
basic sentence and a test in a given state, respectively. The semantic function \( G(\cdot, \text{effect}, \text{test}) : \text{PROMELA} \rightarrow \wp(\text{State}^a) \) associates each model \( M \) with the set of traces \( G(M, \text{effect}, \text{test}) \) representing all possible executions of \( M \), in which functions \( \text{effect} \) and \( \text{test} \) are used when executing a basic sentence or a Boolean expression. Note that functions \( \text{test} \) and \( \text{effect} \) represent the standard implementation of the model data types.

In order to simplify the analysis of properties over \( G(M, \text{effect}, \text{test}) \), we must choose an adequate set of reduced states \((\text{State}^a, \leq^a)\) and an abstraction function \( \alpha : \text{State} \rightarrow \text{State}^a \) which transforms concrete states into their abstractions. Each abstract data is intended to represent a set of concrete states sharing some characteristic which is abstracted. \((\text{State}^a, \leq^a)\) is usually a complete lattice, and the partial order \( \leq^a \) represents the degree of precision of each abstract state. Finally, to obtain the abstract behaviour of the model we must also define abstract versions of \( \text{effect} \) and \( \text{test} \), that is, functions \( \text{effect}^a : \text{Basic} \times \text{State}^a \rightarrow \text{State}^a \) and \( \text{test}^a : \text{BExp} \times \text{State}^a \rightarrow \{\text{false}, \text{true}\} \), giving the proper meaning to the basic PROMELA sentences when executed over abstract states. Given the previous discussion, \( G(M, \text{effect}^a, \text{test}^a) \in \wp((\text{State}^a)^a) \) defines an abstract behavior, easier to be analyzed, for the same model \( M \).

For instance, Fig. 2 shows an excerpt of a PROMELA model that represents the behavior of a lift (extracted from [8]). In order to simplify the exposition, we assume that system states are given by the value of the variable \( \text{Position}[\text{pid}] \) that is an integer number between the values 0..\( \text{nb\_floor} - 1 \). Variable \( \text{Position}[\text{pid}] \) always stores the current floor for the lift identified by \( \text{pid} \). To reduce the model size, consider the poset \((\text{FLOORS}, \leq^a)\) illustrated in Fig. 4 and the abstraction function \( \alpha : [0..\text{nb\_floor} - 1] \rightarrow \text{FLOORS} \) defined as \( \alpha(0) = \text{Lower}, \alpha(\text{nb\_floor} - 1) = \text{Upper} \) and \( \forall 0 < j < \text{nb\_floor} - 1, \alpha(j) = \text{Middle} \). The use of the partial order \( \leq^a \) allows us to include the notion of approximation in the abstract domain \( \text{FLOORS} \): the abstract value \( \text{noUpper} \) approximates any floor different from the \( \text{Upper} \) one, thus \( \text{noUpper} \) is an abstract value less precise than both \( \text{Lower} \) and \( \text{Middle} \). Value \( \text{Unknown} \) is the least precise abstract data since it represents any floor. Finally, value \( \bot \) is used to represent illegal values.

The redefinition of states involves the redefinition of the effect of basic sentences. The table in Fig. 4 shows a definition of the abstract effect.

### 3.1 Correctness

Given an abstraction function \( \alpha \), it is clear that functions \( \text{effect}^a \) and \( \text{test}^a \) may be arbitrarily defined. However the interest of the approach is in preserving some correction properties between \( G(M, \text{effect}, \text{test}) \) and \( G(M, \text{effect}^a, \text{test}^a) \). In [10] there is an exhaustive study of the correctness conditions that \( \text{test}^a \) and \( \text{effect}^a \) must verify for \( G(M, \text{effect}^a, \text{test}^a) \) to be a correct over-approximation of \( G(M, \text{effect}, \text{test}) \).
Correctness conditions guarantee that the reduced/abstract model simulates the original one in the sense that for each non-deadlocked trace \( t = s_0 \rightarrow s_1 \rightarrow \ldots \in G(M, \text{effect}, \text{test}) \) there exists a non-deadlocked abstract trace \( t^\alpha = s_0^\alpha \rightarrow s_1^\alpha \rightarrow \ldots \in G(M, \text{effect}^\alpha, \text{test}^\alpha) \) that over-approximates it, which is denoted by \( \alpha(t) \leq^\alpha t^\alpha \), where \( \alpha(t) \) represents the abstract trace \( \alpha(s_0) \rightarrow \alpha(s_1) \rightarrow \ldots \) and we define \( \alpha(t) \leq^\alpha t^\alpha \) as the relation \( \forall i \geq 0. \alpha(s_i) \leq^\alpha s_i^\alpha \). Note that we explicitly exclude deadlocked traces because the abstraction process may modify this safety property of the system. For instance, the concrete trace

\[
  t = 0 \xrightarrow{i=i+1} 1 \xrightarrow{i=i+1} 2 \xrightarrow{i=i-1} 1 \xrightarrow{i=i-1} 0 \xrightarrow{\text{skip}} \ldots
\]

could be approximated by the abstract trace

\[
  t^\alpha = \text{Lower} \xrightarrow{i=i+1} \text{Middle} \xrightarrow{i=i+1} \text{noLower} \xrightarrow{i=i-1} \text{noUpper} \xrightarrow{i=i-1} \text{noUpper} \xrightarrow{\text{skip}} \ldots
\]

We have labelled each transition with the basic instruction executed. Note that we have used the table in Fig. 4 to realize each abstract transition.

We implement the loss of information when executing abstract operations using specific abstract constants instead of sets of constants as employed in [9]: for example, when the value \( \text{Middle} \) is incremented, we use the value \( \text{noUpper} \) instead of the set \( \{\text{Lower}, \text{Middle}\} \). When applied to abstract tests, this means that function \( \text{test}^\alpha \) always produces a safe result, that is, it returns \( \text{true} \) iff in some concrete execution the value \( \text{true} \) may be returned. Thus, given \( p \in BExp \) and \( s^\alpha \in \text{State}^\alpha \), function \( \text{test}^\alpha \) is defined as

\[
  \text{test}^\alpha(p, s^\alpha) = \bigvee_{s \in \text{State}^\alpha \alpha(s) \leq^\alpha s^\alpha} \text{test}(p, s) \quad (\text{Over})
\]

In addition, in the following section, we will make use of the abstracted constants to implement the over-approximation method for evaluating temporal formulas.

\[3.2 \text{ Syntactic transformation of Promela}\]

The syntactic transformation of a PROMELA model \( M \) to obtain a new model \( M^\alpha \) is based on replacing each basic instruction in \( M \) by a standard PROMELA code that implements \( \text{test}^\alpha \) and \( \text{effect}^\alpha \). Then the verification is carried out by only executing standard PROMELA instructions. This approach corresponds to implementing a verifier for \( G(M^\alpha, \text{test}, \text{effect}) \).

\[25\]
For instance, the next code shows FLR\_INC, a possible implementation of abstract increment $i = i + 1$ defined in Fig. 4.

```c
#define FLR_INCR(v) (((x==Lower))->Middle:
    (((x==Middle))->noLower:
    (((x==noUpper))->noLower:
    (((x==noLower))->noLower:
    (((x==Unknown))->noLower: (ILLEGAL)))))
```

In this code, the constant ILLEGAL is employed to represent $\bot$. The code in Fig. 2 is now replaced by the following one, that illustrates the use of the abstract instruction (effect) to increase the variable Position[].

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

The same method is employed to implement test. For example, the next definition contains the implementation of FLR\_EQ (abstract test for $(i==j)$)

```c
#define FLR_EQs(x,y) ( (x==Lower && y==Lower) || (x==Upper && y==Upper) )
#define FLR_EQw(x,y) (((x==Upper)&&(y==noLower)) || ((x==noLower)&&(y==Upper)) ||
((x==Lower)&&(y==noUpper)) || ((x==noUpper)&&(y==Lower)) ||
((x==Middle)&&(y==noUpper)) || ((x==noUpper)&&(y==Middle)) ||
((x==Middle)&&(y==noLower)) || ((x==noLower)&&(y==Middle)) ||
((x==Unknown)) || ((y==Unknown)))
#define FLR_EQ(x,y) (FLR_EQw(x,y) || FLR_EQs(x,y))
```

Function FLR\_EQ verifies the correctness conditions (studied in [10]) necessary for the abstract model to correctly simulate the original one. Informally, FLR\_EQ\(_{(x,y)}_k\) is true when $a == b$ holds for some concrete data $a$ and $b$ abstracted by $x$ and $y$, respectively, as defined in (Over) equation. This explains why FLR\_EQ(Upper, noLower) returns true. The reason for defining FLR\_EQ using two macros (FLR\_EQs and FLR\_EQw) will be explained below.

The user has to select the variables to be abstracted and the abstraction to be applied ($\alpha$), and then the transformation is automatically performed.

### 4 Abstracting Temporal Logic

Once the model has been reduced using the method described in the previous section, the following step is to define the satisfaction of a temporal formula over the abstract model (which is called the abstract satisfaction) and to relate it with the satisfaction the formula over the original one.

Atomic propositions in LTL formulas regarding PROMELA models are Boolean expressions. Thus, considering the standard notion of satisfiability models are Boolean expressions. Thus, considering the standard notion of satisfiability given in Fig. 3, and following the same idea used for abstracting the model, we may assert that in order to define the abstract satisfaction of a temporal formula
it suffices to define the abstract satisfaction of the atomic propositions. One clear possibility is to use the function $test^\alpha$, as defined in the previous section \((Over)\), to evaluate the atomic propositions. Using $test^\alpha$ leads us to construct the so-called over-approximation method for abstracting temporal formulas, which has been studied at length in [11]. An alternative possibility is to use the following function $test^\alpha_c$ to evaluate the atomic propositions. Given $p \in BExp$ and $s^\alpha \in State^\alpha$, $test^\alpha_c$ is defined as

$$test^\alpha_c(p, s^\alpha) = \bigwedge_{\{s \in State.\alpha(s) \leq^\alpha s^\alpha\}} test(p, s)$$ \((Under)\)

Classic papers integrating model checking and abstraction [2,6] use function $test^\alpha_c$ to evaluate temporal formulas. Function $test^\alpha$ incorporates the dual method that may be of interest in some occasions, as discussed in the next section. Now, we summarize the main theoretical results concerning the relation between the abstract satisfaction of temporal formulas over the abstract model (using both the classic and the over-approximated method) and the satisfaction over the concrete model.

In the rest of the section, we write:

(i) $s \models p$ when $test(p, s)$ holds,

(ii) $s^\alpha \models^\alpha p$ when $test^\alpha(p, s^\alpha)$ holds, and

(iii) $s^\alpha \models^\alpha_c p$ when $test^\alpha_c(p, s^\alpha)$ holds.

We also extend $\models$, $\models^\alpha$ and $\models^\alpha_c$ to abstract traces defining the meaning of temporal operators as in Fig. 3. Note that, for instance, when we write $M^\alpha \models^\alpha_c \forall f$, we mean that $\forall t^\alpha \in G(M, effect^\alpha, test^\alpha).t^\alpha \models^\alpha_c f$, and so on.

The following theorem presents two direct results of the previous definitions. In the theorem, we assume that $G(M, effect^\alpha, test^\alpha)$ is a correct over-approximation of model $G(M, effect, test)$ in the sense described in the previous section, and that the original model is deadlock-free.

**Theorem 4.1** Given a temporal formula $f$

$$M^\alpha \models^\alpha_c \forall f \Rightarrow M \models \forall f$$

$$M^\alpha \not\models^\alpha_c \exists f \Rightarrow M \not\models \exists f$$

The first result corresponds to the classic weak preservation of universal properties studied in [6]. Using the classic methodology, the satisfaction of universal properties is directly preserved from the abstract to the concrete model. The second result is the dual preservation result. Using the over-approximation method, the refutation of existential properties is directly preserved from the abstract to the concrete model. Note that these results are not equivalent because they deal with negation using non-standard and dual approaches. Given a proposition $p$ and an abstract state $s^\alpha$, using definition \((Under)\), it is possible that for the classic method neither $test^\alpha_c(p, s^\alpha)$ nor $test^\alpha_c(\neg p, s^\alpha)$ hold. In contrast, due to definition \((Over)\) for the over-
approximation method, it is possible that both $\text{test}^{\alpha}(p, s^{\alpha})$ and $\text{test}^{\alpha}(-p, s^{\alpha})$ hold. This is why we skipped the negation rule from Fig. 3. Thus, considering that formula $\neg f$ is in negation normal form, we have that $M^{\alpha} \not\models ^{\alpha}\forall f \not\models M^{\alpha} \models ^{\alpha}\exists \neg f$, and, in addition, $M^{\alpha} \models ^{\alpha}\forall f \not\models M^{\alpha} \models ^{\alpha}\exists \neg f$.

4.1 Syntactic transformation of LTL

The syntactic transformation of temporal formulas is straightforward on the basis of the previous discussion. The first step consists of writing the formula in negative normal form (if necessary). Then the propositions are automatically replaced by the abstract implementation of the test, depending on the method to be employed. For the implementation of the over-approximation method, propositions are defined using the same definition of $\text{test}^{\alpha}$ employed to transform the model. But the implementation of $\text{test}^{\alpha}$ must be more restrictive than $\text{test}^{\alpha}$ in order to ensure the criterium defined above (Under). A definition like

\[
\text{#define FLR_EQs(x,y) \((x==\text{Lower}&&y==\text{Lower})||(x==\text{Upper}&&y==\text{Upper})\)}
\]

implements the (classic) abstract test for $(i==j)$. Informally, $\text{FLR_EQs}(x,y)$ is true when $a == b$ holds for every concrete data $a$ and $b$ abstracted by $x$ and $y$, respectively, as defined in (Under). Note that $\text{FLR_EQ}$ uses the two macros $\text{FLR_EQs}$ and $\text{FLR_EQw}$ in order to consider the cases where only some concrete states satisfy $(i==j)$.

5 Using $\alpha$Spin: A case study

In this section, we describe the main functionality that $\alpha$Spin adds to Spin/XSpin. Our case study is a variant of the PROMELA code for an elevator controller presented in [8]. In this section, we show how to employ the dual approaches for the refutation and verification of temporal properties.

The first experiment is to discard errors by refutation (with the over-approximation method). The second one consists of checking a desired universal property with the classic method.

5.1 The model

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. Following the rules about how to construct suitable models for verification, we have made a set of changes to work with one lift (see Fig. 5). The input to the system is modelled by a process that produces user requests from inside the lift (internal requests). The lift is represented with the $\text{Lift()}$ process that receives orders to move up, down and stop, thus updating the Position of each one. The control part receives the inputs and sends the orders to the $\text{Lift()}$ process. This part is divided into several processes ($\text{SysLift()}$,
Global variables

Fig. 5. a) Scheme of the lift system b) One view in αSpin

SysStop(), and Sampler() that communicate via rendezvous channels and global variables. The main variable to control the flow in every process is the global variable Position, that always stores the current floor for the lift. The global array internal_request[nb_floor] stores the pending requests to move to specific floors, nb_floor being the actual number of floors in the system. The code in Fig. 2 shows the updating of this variable in the Lift() process depending on the order from the control part (Up, Down, Stop).

5.2 Discarding errors

One critical property to check whether the control system works properly is the absence of movement in the absence of requests. The property NoMove says that “the lift never starts the movement without any request”. If we want to check the property when the lift is on the lower floor, we can encode it as the temporal formula

\[
\text{NoMove: } \langle \langle posL \land \text{no_request} \rangle \langle posAboveL \rangle
\]

and then we can use Spin to verify that there are no executions satisfying the formula (done in Fig. 2), where the propositions posL and posAboveL represent whether the lift is currently on or above the lower floor, and no_request represents that there are no users for the lift. These propositions are defined according to their interpretation standard or abstract as defined in Section 4.

The main problem in verifying the concrete model (with standard meaning for propositions) 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 NoMove give us a guide on how to abstract. As the evaluation of these propositions mainly
relied 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 structure of the model, and it can also be guessed by the user from the output like the one in Fig. 5. 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 suitable for the variables in the temporal formula and the current binding of variables to abstraction functions (`Position[1]` will be abstracted using `FLR`). When the abstraction functions have been selected, `αSpin` performs the syntactic transformation of the model depending on the user’s choice. When the choice is `Property holds for No Executions (error behaviour)`, as shown in Fig. 6, the code is produced to employ the over-approximation of the formula. The macros `FLR_EQ`, `FLR_GT`, `FLR_NE`, `...` implement this over-approximation, `test^n`, as described before. As shown in Fig. 6 the error is not present either in the abstract model or, using Theorem 4.1, in the concrete model.

The benefits of using the abstract formula to discard this error are summarized in Fig. 8. The formula employed to check movement is the previous one extended to also consider departure for upper and middle floors. The expected number of visited states is greatly reduced compared to the concrete model (see Fig. 8). Furthermore, the variation of the number of states is linear with respect to the number of floors.

5.3 Verification of a desired behaviour

After discarding the key critical error behaviors, we proceed by verifying that the lift system works to provide the intended service. The property `Move` says that “the lift always starts the movement to the requested floor”

The version of the property as a desirable behaviour could be 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`, `posAboveL`, and `noPosM` represent whether the lift is currently at, above or below, a specific floor. Again, these propositions are defined according to the interpretation standard or non-standard, depending on how the verification is to be performed (with concrete or abstract model).

The model is transformed (automatically) in the same way that the refutation case, but when selecting `Property holds for All Executions (desired behaviour)`, the formula is transformed (automatically) to employ the classic method. Note that in Fig. 7 the propositions in the formula are defined using the macros `FLR_EQs`, `FLR_GTs`, `FLR_NEs`, `...` that implement the classic
under-approximation test. Now the verification result “valid” in the abstract formula ensures that the concrete model satisfies the property. The benefits of verifying with this method are shown in Fig. 8.

6 Implementation

The main design criteria in our abstraction tool is to obtain as much independence with respect to particular modelling languages and model checkers as possible. So we consider XML as the unique internal representation to perform the abstraction by transformation as shown in Fig. 9. The actual modelling language can be translated into this representation by a front-end module (steps 1 and 2 in Fig. 9) 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.

In addition to 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 (see [12]). 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...
Gallardo, Martínez, Merino, Pimentel

Fig. 8. Verification results

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 or locating the points where a particular variable is employed.

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, associativity rules, etc (see Fig. 10). Furthermore, the whole abstraction library can be defined as an XML repository.

The current implementation is composed of the modules shown in Fig. 9. Most of them have been completed, and we are now working on the abstraction prover, that will assist in generating new correct abstraction functions to be included in the library.

7 Concluding Remarks

The main contribution of this paper is the presentation of a tool to perform abstraction by syntactic transformation in the context of explicit model checking. We have presented the actual state of αSpin, a tool that integrates the classic method for abstraction and our over-approximation method. Documentation and current and future versions of αSpin can be found at [20].

The theoretical approach that support the transformation gives us a safe framework to extend implementation preserving the relation between the results in the abstract and the concrete models (and formulas). For example, we have implemented a method to check existential properties (those that are true for at least one path) [6]. To do that, the model has to be transformed
using the most constrained versions of abstract \textit{effect} and \textit{test} (\textit{effect}$_\alpha$ and \textit{test}$_\alpha$), and the formula is transformed using \textit{test}$_\alpha$. We are now extending the theoretical framework to give support to new transformations (a related work can be found in [4]).

Other interesting contributions are the use of abstraction libraries and the use of XML to support the abstraction process. The library should be employed to store new functions that are revealed as useful in the verification experiences. It is even possible to give a taxonomy to these functions to make their use easier [14]. Again, XML is a good candidate to store this information.

Our future work is to add strategies to automatically analyze the correctness of abstraction functions using PVS. Another line of work is to improve counterexample analysis [3].

\section*{Acknowledgements} We would like to thank the referees for their helpful and constructive comments.

\section*{References}


34