Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 295–313
www.elsevier.com/locate/entcs

An Abstract Interpretation Toolkit for μCRL

Jaco van de Pol Miguel Valero Espada[1,2]

Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

This paper describes a toolkit that assists in the task of generating abstract approximations of process algebraic specifications written in the language μCRL. Abstractions are represented by *Modal Labelled Transition Systems*, which are mixed transition systems with *may* and *must* modalities. The approach permits to infer the satisfaction or refutation of *safety* and *liveness* properties expressed in the (action-based) μ-calculus. The tool supports the abstraction of states and action labels which allows to deal with infinitely branching systems.

Keywords: Abstract Interpretation, Modal Transition Systems, Abstract Model Checking, μCRL Toolset.

1 Introduction

The automatic verification of distributed systems is limited by the well known state explosion problem. Abstraction is a useful approach to reduce the complexity of such systems. From a *concrete* specification, it is possible to extract an *abstract* approximation that preserves some interesting properties of the original. In [32], we have presented the theoretical framework for abstracting μCRL [16] specifications. μCRL is a language that combines ACP style process algebra [3] with abstract data types. In this paper, we will describe the toolkit that implements the theory.

[1] Email: {vdpol, miguel}@cwi.nl
[2] Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

doi:10.1016/j.entcs.2004.08.070

Semantically, abstractions are represented by *Modal Labelled Transition Systems* [23], which are *mixed* transition systems in which transitions are labelled with actions and with two modalities: *may* and *must*. *May* transitions determine the actions that are part of some refinements of the system while *must* transitions denote the ones that necessarily appear in all refinements. The use of the two modalities allows to infer the satisfaction or refutation of formulas written in (action-based) μ-calculus [22] from the abstract to the concrete system.

The implementation of the previously developed theory is an indispensable step in order to apply abstract interpretation techniques to realistic systems. There exist different abstraction approaches that can be applied within the verification methodology. For example, *variable hiding* or *pointwise* abstraction in which, first, the value of some variables of the specification is considered as unknown, subsequently, extra non-determinism is added to the system when there are predicates over the abstracted variables. Another automated abstraction technique is the so-called *predicate abstraction* in which only the value of some conditions is retained and propagated over the dependent predicates of the specification. *Program slicing* is a technique that tries to eliminate all parts of the specification that are not relevant for the current verification.

The most common abstraction technique consists in interpreting the concrete specification over a smaller data domain. The user selects the set of variables to abstract and provides a new abstract domain that reflects some aspects of the original. This technique requires creative human interaction in order to select the parts of the system that are suitable to abstract and to provide the corresponding data domains. Furthermore, the user must ensure that the abstract interpretation satisfies some so-called safety requirements.

Our tool implements the automatic *pointwise* abstraction and, moreover, assists the user to create his own abstractions. The tool supports the use of two mainstream techniques for data abstraction. One proposed by Long, Grumberg and Clarke [6,24], in which the concrete and the abstract data domain are related via a homomorphic function and another based on Cousots' Abstract Interpretation theory (we use Abstract Interpretation with upper cases to refer to Cousots' work and abstract interpretation with lower cases to denote the general framework), see for example [7,8,20,21], in which data is related by Galois Connections. A lifting mechanism is also implemented which allows to automatically build Galois Connections from homomorphisms, see [28].

Standard abstraction frameworks are only based on the abstraction of states which make them unable to deal with infinitely branching systems with action labels. A unique feature of our tool is that it allows the abstraction of

both states and action labels. In the implementation, we try to reuse existing tools as much as possible. In particular, we encode *Modal*-LPEs as LPEs and *Modal*-LTSs as LTSs, in order to reuse the μCRL and CADP toolsets. We also provide a new method to reduce the 3-valued model checking problem to two 2-valued model checking problems.

This paper is structured as follows: first, we introduce the basic concepts of abstract interpretation, then, we describe the main functionalities of our tool. Subsequently, we give a short description of some case studies that have been analysed using the tool. The paper concludes with a comparison with other related tools.

2 Preliminaries

2.1 Transition Systems

The semantics of a system can be captured by a *Labelled Transition System* (LTS). We assume a non-empty set S of states, together with a non-empty set of transition labels Act, then:

Definition 2.1 A transition is a triple $s \xrightarrow{a} s'$ with $a \in Act$ and s,$s' \in S$. We define a *Labelled Transition System* (LTS) as tuple $(S, Act, \rightarrow, s_0)$ in which S and Act are defined as above and $\rightarrow$ is a possibly infinite set of transitions and s_0 in S is the initial state.

Basically, $s \xrightarrow{a} s'$ denotes that the state s can evolve into the state s' by the execution of an action a. To model abstractions we use a different structure that allows to represent approximations of the concrete system in a more suitable way. In a *Modal Labelled Transition System* (*Modal*-LTS), transitions have two modalities *may* and *must* which denote the possible and necessary steps in the refinements. This concept was introduced by Larsen and Thomsen [23]. The formal definition extends the definition of LTSs by considering the two modalities.

Definition 2.2 A *Modal Labelled Transition System* (*Modal*-LTS) is a tuple $(S, Act, \rightarrow_{may}, \rightarrow_{must}, s_0)$ where S, Act and s_0 are as in the previous definition and $\rightarrow_{may}, \rightarrow_{must}$ are possibly infinite sets of (may or must) transitions of the form $s \xrightarrow{a}_x s'$ with s, $s' \in S$, $a \in Act$ and $x \in \{may, must\}$. Every *must*-transition is a *may*-transition ($\xrightarrow{a}_{must} \subseteq \xrightarrow{a}_{may}$).

From a concrete system described by an LTS we can generate an abstraction of it by relating concrete states and action labels with abstract ones. Given the abstraction relation, we construct a double approximation of the concrete

system modelled by a *Modal*-LTS. The *may*-transitions correspond to an over-approximation of the original and the *must* ones to an under-approximation. In [32], we have presented the complete formal framework for abstracting, now we give an example (see Figure 1) to introduce the basic intuition (note that we use lower case to denote concrete states and concrete action labels and upper case for abstract states and action labels; arrows without source indicate the initial states).

If all concrete states related to an abstract state S have a transition to a concrete state related to an abstract state S', then there is a *must* transition between S and S'. Therefore, in Figure 1, we have the abstract *must* transition $S_0 \rightarrow_{must} S_0$ and $S_1 \rightarrow_{must} S_2$. If there is some concrete state related to an abstract state S with a transition to another state related to an abstract state S', then there is a *may* transition between S and S'. In the figure, these abstract transitions are marked by the dashed arrows. Actions labels can be also abstracted, in the example the concrete labels: $\{a_0, a_1\}$ are mapped to the abstract label A and $\{b_0, b_1\}$ to B as is shown in the figure. Whenever there is a *must* transition, there is also a *may* one, we do not explicitly draw such cases.

2.2 Process Equations

μCRL is a formal language for specifying protocols and distributed systems in an algebraic style. A μCRL specification consists of two parts: one part specifies the data types, the other part specifies the processes.

The specification of a process is constructed from action names from a set `ActNames` (`ActN` for short), recursion variables and process algebraic operators. Actions and recursion variables carry zero or more data parameters. There are two predefined processes in μCRL: δ represents deadlock, and τ a hidden action. They never carry data parameters.

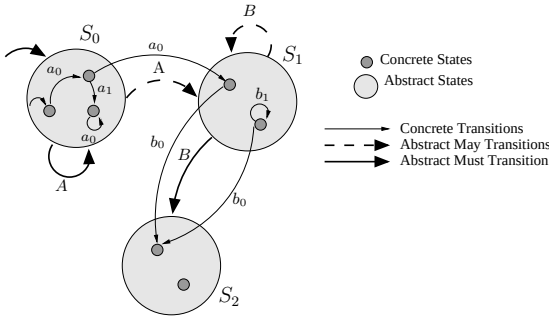


Fig. 1. Example of Modal Abstraction of an LTS.

Processes are represented by process terms, which describe the order in which the actions may happen. Processes are constructed using the following algebraic operators: $p{\cdot}q$ which denotes sequential composition and $p + q$ non-deterministic choice, summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D, and the conditional construct $p \lhd b \rhd q$ with b a data term of data type *Bool* behaves as p if b and as q if $\neg b$. Parallel composition $p \parallel q$ interleaves the actions of p and q; moreover, actions from p and q may also synchronize to a communication action.

A data type (or sort) consists of a signature in which a set of function symbols, and a list of axioms are declared. For every specification, we assume the existence of the booleans (*Bool*), with the constants true and false (T and F) and their standard functions. The syntax and semantics of μCRL are given in [16].

Every μCRL specification may be encoded by a *Linear Process Equation*:

$$X(d:D) = \sum_{a \in \mathtt{ActN}} \sum_{e_a : E_a} a(f_a(d, e_a)).X(g_a(d, e_a)) \lhd c_a(d, e_a) \rhd \delta \qquad \text{(LPE)}$$

An LPE is a concise representation of all possible interleavings of a system in which parallel composition is eliminated. In the definition, d denotes a vector $[d_0 : D_0, ..., d_n : D_N]$ that represents the state of the system in every moment. We use the keyword *init* to declare the initial vector of values of d. The process is composed by a finite number of summands. Every summand has a list e_a of local variables $[e_{a_0}, ..., e_{a_M}]$, of possibly infinite domains $[E_{a_0}, ..., E_{a_M}]$, and it is of the following form: a condition $c_a(d, e_a)$, if the evaluation of the condition is true the process executes the action a with the parameter $f_a(d, e_a)$ and will move to a new state $g_a(d, e_a)$, which is a vector of terms of type D. $f_a(d, e_a), g_a(d, e_a)$ and $c_a(d, e_a)$ are terms built recursively over parameters, local variables, and function symbols f defined in the data specification. To every LPE corresponds a *Labelled Transition System* (LTS), that represents the full behavior of the system. The semantics of the system described by an LPE are given by the following rules:

- $s_0 = \text{init}_{lpe}$
- $s \xrightarrow{a(d)} s'$ if there exists $e_a \in E_a$ such that $c_a(s, e_a) = \mathsf{T}$, $g_a(s, e_a) = s'$ and $d = f_a(s, e_a)$

Basically, the abstraction process consists of a symbolic transformation of the original specification into an intermediate format (*Modal*-LPE) that encodes the modal abstraction. *Modal*-LPEs capture the extra non-determinism arising from abstract interpretation. They allow a simple transition to lead to a

set of states with a *set* of action labels.

$$X(d:\mathcal{P}(D)) = \sum_{a\in \mathtt{ActN}} \sum_{e_a:E_a} a(F_a(d,e_a)).X(G_a(d,e_a)) \lhd C_a(d,e_a) \rhd \delta \qquad (MLPE)$$

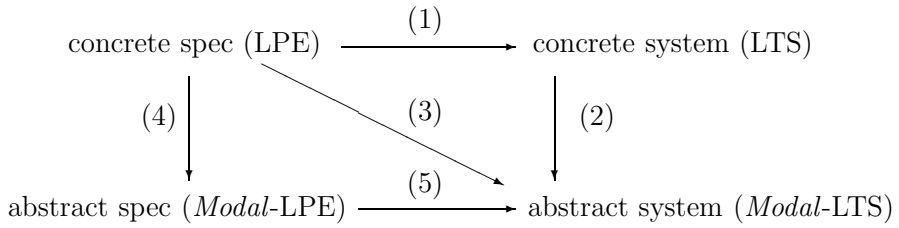
The definition is similar to the one of *Linear Process Equation*, the difference is that the state is represented by a list of power sets of abstract values and for every C_a returns a non empty set of booleans, G_a a non empty set of states and F_a also a non empty set of action parameters. To every *Modal*-LPE corresponds a *Modal Labelled Transition System*. The semantics of the system described by a *Modal*-LPE are given by the following rules:

- $S_0 = \text{init}_{mlpe}$
- $S \overset{a(D)}{\rightarrow}_{must} S'$ if there exists $e_a \in E_a$ such that $\mathsf{F} \notin C_a(S,e_a)$, $D = F_a(S,e_a)$ and $S' = G_a(S,e_a)$
- $S \overset{a(D)}{\rightarrow}_{may} S'$ if there exists $e_a \in E_a$ such that $\mathsf{T} \in C_a(S,e_a)$, $D = F_a(S,e_a)$ and $S' = G_a(S,e_a)$

The next section describes the tool and the methodology to apply abstract interpretation of process algebraic specifications.

3 ToolKit

The next figure shows the different possibilities to extract abstract approximations from concrete specifications.



From a concrete system, encoded as an LPE, we can:

- Generate the concrete transition system (1), from which we compute the abstraction (2). Even though the resulting abstraction is optimal, this option is not very useful for verification because the generation of the concrete transition system may be impossible (or too expensive) due to the size of the state space.
- Generate directly the abstract *Modal*-LTS (3), by interpreting the concrete specification over the abstract domain. This solution avoids the generation of the concrete transition system.

- First, generate a symbolic abstraction of the concrete system (4), and then extract the abstract transition system (5).

Typically, standard abstract interpretation frameworks implement the second approach (arrow (3) of the figure), however we believe that the third (arrow (4) followed by (5)) one is more modular. *Modal*-LPEs act as intermediate representation that may be subjected to new transformations. There exists several tools and algorithms (see [4]) that manipulate linear equations that do, for example, symbolic model checking, state space reduction, elimination of dead code, confluence analysis, ...

3.1 Overview of the tool

The following figure describes the tool architecture, whose main components are:

Abstractor. It is in charge of performing the symbolic transformation from LPEs to *Modal*-LPEs. It gets a μCRL specification in linear format and, typically, a set of parameters and variables to abstract, then it generates a new specification. The new specification is the skeleton of the abstraction, it has to be completed by adding the abstract data specification. The tool allows the use of different ways of abstracting (homomorphisms, Galois Connections and lifted homomorphisms), the resulting specification will depend on the user's choice.

Abstraction Loader. It is in charge of managing the data specifications. From the *Modal*-LPE skeleton, the *Loader* may export the abstract signature that the user has to provide in order to complete the specification. It is also used to import abstract data types from external files, and to generate automatic abstractions by *hiding* variables. As we previously marked, abstract interpretations have to be proved correct, the tool generates the safety criteria that abstract functions have to satisfy. Some safety requirements can be automatically proved correct using the μCRL theorem prover, the others need human interaction.

Abstract Model Checker. The transition system generated from an abstraction represents a double approximation of the original. We use a 3-valued logic in order to infer the satisfaction or refutation of properties. The 3-valued model checking problem can be transformed to two standard 2-valued problems. Hence one can use the existing model-checking tools.

Action labels may be abstracted. Therefore, formulas have to be abstracted according to the abstract action labels. Due to the abstraction of formulas, in some cases, we cannot infer the exact result of the model checking of the concrete formula; in section 3.4, we provide the guidelines to model check and

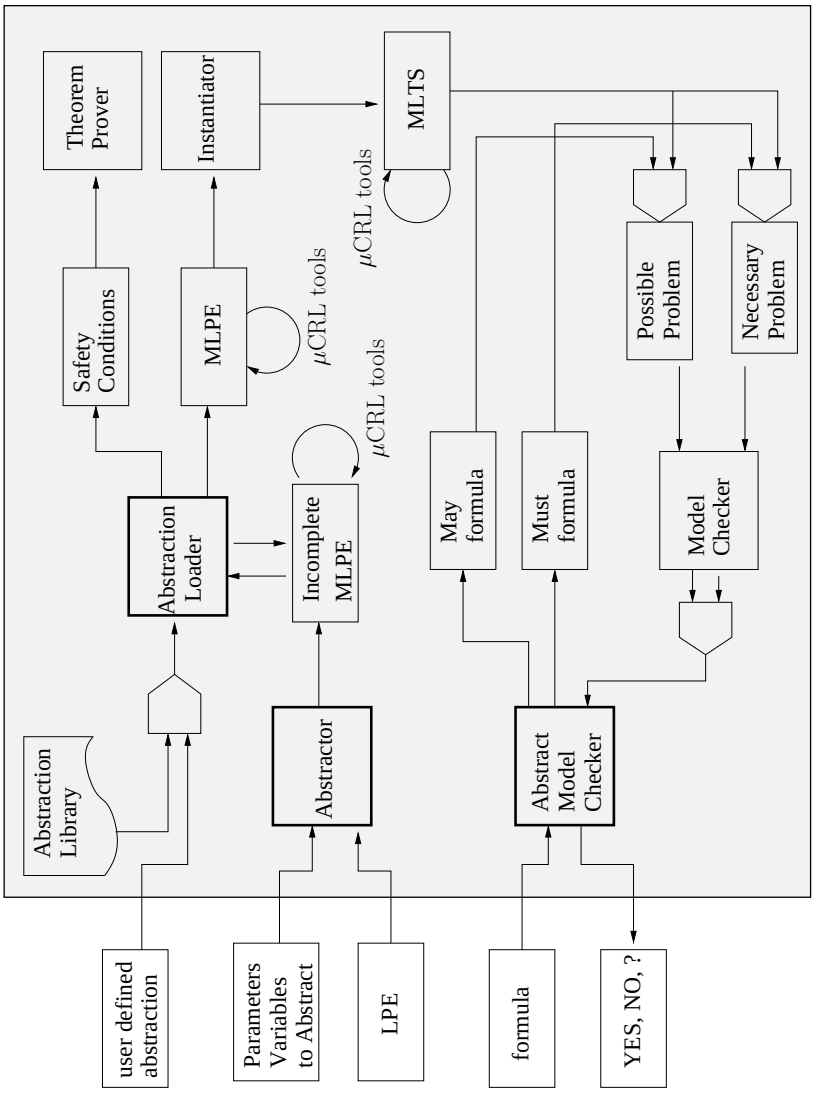


Fig. 2. Tool Architecture

to infer the results.

3.2 Abstractor

The *Abstractor* replaces the data terms by their abstract counterparts. The user can select the parameters and variables to abstract, then the abstraction is propagated over the data terms of the specification. Concrete function symbols $f : X \to Y$ are replaced, when needed, by their abstract versions, which, in

general, have the following form: abs_F : abs_X $\to \mathcal{P}$(abs_Y). Let us consider a very simple example:

$$X(l : List) = \sum_{b:Bit} write(b).X(cons(b,l)) \lhd lt(len(l), MAX) \rhd \delta + \\ read(head(l)).X(tail(l)) \lhd lt(0, len(l)) \rhd \delta$$

The linear process specifies a bounded buffer. The process can non-deterministically choose between executing a *write* or a *read* action. The *write* can only be performed if the buffer is not full, i.e., the length of the list that models the buffer is smaller that the maximal length (MAX). The *read* action can be performed if the buffer is not empty. In the first case, the state parameter is updated by concatenating a new bit to the list; in the second case, the first element of the list is removed. The concrete specification has the following signatures:

- $cons : Bit \times List \to List$
- $len : List \to Nat$
- $lt : Nat \times Nat \to Bool$
- $head : List \to Bit$
- $tail : List \to List$

If the user selects the parameter l to be abstracted then the propagation of the abstraction will yield the following signatures (for the full list of transformation rules, please, consult the technical report [30]):

- $abs\_cons : Bit \times \mathcal{P}(abs\_List) \to \mathcal{P}(abs\_List)$
- $abs\_len : \mathcal{P}(abs\_List) \to \mathcal{P}(Nat)$
- $lt : \mathcal{P}(Nat) \times \mathcal{P}(Nat) \to \mathcal{P}(Bool)$
- $abs\_head : \mathcal{P}(abs\_List) \to \mathcal{P}(Bit)$
- $abs\_tail : \mathcal{P}(abs\_List) \to \mathcal{P}(abs\_List)$

To complete the specification, the user has to provide the domain of the abstract list, abs_List, the relation between the concrete domain and the abstract one and the definitions for the new functions (we will present an example in the next section). All the functions needed to manipulate sets of values are automatically provided by the tool by performing a pointwise application of the non-abstracted ones.

The tool allows the use of two mainstream techniques to relate concrete and abstract domains: homomorphisms and Galois Connections. In the homomorphic approach a concrete data value is related to a single abstract value, via a mapping function H. In the Galois Connection approach a concrete data value might be related to several abstract values. Typically, the abstract do-

main is structured as a lattice. The order in the lattice determines the grade of precision of the abstract values. To define a Galois Connection, apart from the ordered domains, we need an abstraction function α and a concretization function γ.

In general the latter approach preserves more information about the original system but state space reductions may be bigger using the homomorphic technique. Furthermore, the homomorphic approach is intuitively easier for the user since it is simpler to think in terms of mappings between values than in term of Galois Connections between lattices.

The *Abstractor* supports the use of both approaches and also the combination of them that consists of the lifting of an homomorphism to a Galois Connection. In practice, this possibility is very fruitful because it permits the user just to provide the mapping between the concrete and the abstract data domain and the definition of the abstract functions. The tool automatically lifts the structure to a Galois Connection. For further details, please consult [30].

Modal-LPEs can be transformed back to standard *Linear Process Equations*. This allows the reuse of the μCRL tools that are conceived to manipulate LPEs. To do that, first we extend the action labels by adding two suffixes. Let $\texttt{ActNames}$ be the set of action labels of a *Modal*-LPE, we define $\texttt{ActNames}_{may/must} = \{a\_may \mid a \in \texttt{ActNames}\} \cup \{a\_must \mid a \in \texttt{ActNames}\}$. Then, we duplicate the number of summands generating for every summand of the *Modal*-LPE two new ones, one for the *may* transitions and the other for the *must* transitions. These new summands are built following the patterns presented below. By $\vec{G}_a$ we denote the sort of elements of G_a (the same holds for $\vec{F}_a$). The pattern for Galois Connections and lifted homomorphisms is:

$$
\begin{aligned}
X(d:\mathcal{P}(D)) = & \sum_{a\in\texttt{ActN}} \sum_{e_a:E_a} a\_may(F_a(d,e_a)).X(G_a(d,e_a)) \\
& \lhd\, member(\mathsf{T}, C_a(d,e_a)) \\
& \rhd\, \delta + \\
& \sum_{a\in\texttt{ActN}} \sum_{e_a:E_a} a\_must(F_a(d,e_a)).X(G_a(d,e_a)) \\
& \lhd\, not(member(\mathsf{F}, C_a(d,e_a))) \\
& \rhd\, \delta
\end{aligned}
\qquad (\textit{MLPE to LPE (GC)})
$$

The pattern for homomorphisms is:

$$
\begin{aligned}
X(d:\mathcal{P}(D)) = & \sum_{a\in \texttt{ActN}} \sum_{e_a:E_a} \sum_{f_a:\vec{F}_a} \sum_{g_a:\vec{G}_a} a\_may(f_a).X(\{g_a\}) \\
& \lhd member(\mathsf{T}, C_a(d,e_a)) \wedge \\
& member(f_a, F_a(d,e_a)) \wedge \\
& member(g_a, G_a(d,e_a)) \\
& \rhd \delta + \\
& \sum_{a\in \texttt{ActN}} \sum_{e_a:E_a} \sum_{f_a:\vec{F}_a} a\_must(f_a).X(G_a(d,e_a)) \qquad \textit{(MLPE to LPE (H))} \\
& \lhd not(member(\mathsf{F}, C_a(d,e_a))) \wedge \\
& singleton(F_a(d,e_a)) \wedge \\
& member(f_a, F_a(d,e_a)) \wedge \\
& singleton(G_a(d,e_a)) \\
& \rhd \delta
\end{aligned}
$$

The patterns are derived from the semantics of *Modal*-LPEs presented in section 2.2. For the homomorphism, we require the states of the process and the arguments of the actions to be single abstract values, because every concrete value is mapped to only one abstract one. However, for the Galois Connection we allow them to be sets of values.

For the above example, using the Galois Connection approach, the resulting $\text{LPE}_{may/must}$ will be:

$$
\begin{aligned}
& X(l:\mathcal{P}(abs\_List)) = \\
& \quad \sum_{b:Bit} write\_may(b).X(abs\_cons(b,l)) \lhd member(\mathsf{T}, lt(abs\_len(l), \{MAX\})) \rhd \delta + \\
& \quad \sum_{b:Bit} write\_must(b).X(abs\_cons(b,l)) \lhd not(member(\mathsf{F}, lt(abs\_len(l), \{MAX\})) \rhd \delta + \\
& \quad read\_may(abs\_head(l)).X(abs\_tail(l)) \lhd member(\mathsf{T}, lt(\{0\}, abs\_len(l))) \rhd \delta + \\
& \quad read\_must(abs\_head(l)).X(abs\_tail(l)) \lhd not(member(\mathsf{F}, lt(\{0\}, abs\_len(l))) \rhd \delta
\end{aligned}
$$

The equivalence of the *Modal*-LPE and the $\text{LPE}_{may/must}$ is given by the following proposition:

Proposition 3.1 *Let $\mathcal{M}$ be a* Modal-*LPE, and let $m\mathcal{L}$ be the corresponding* Modal-*LTS $(S, Act, \rightarrow_{may}, \rightarrow_{must}, s_0)$. Moreover, let $\mathcal{M}_{may/must}$ be the equivalent $LPE_{may/must}$ of $\mathcal{M}$, and let $\mathcal{L}$ be its corresponding LTS $(S, Act_{may/must}, \rightarrow, s_0)$. Then, for all $s, s' \in S$ and $a \in$ `ActNames`, which a possibly empty vector $\bar{d}$ of arguments, we have:*

- $s \overset{a\_may(\bar{d})}{\rightarrow} s' \iff s \overset{a(\bar{d})}{\rightarrow}_{may} s'$

- $s \overset{a\_must(\bar{d})}{\longrightarrow} s' \iff s \overset{a(\bar{d})}{\longrightarrow}_{must} s'$

The proposition holds for both types of abstraction. The proof can be found in [30].

3.3 Abstraction Loader

The *Abstractor* returns the skeleton of the abstraction, i.e, an incomplete *Modal*-LPE. In order to generate the corresponding *Modal*-LTS, the user has to complete the *Modal*-LPE by providing the abstract domains and the definition of the abstract functions. The *Abstraction Loader* assists the user to manage abstract domains by providing import/export mechanisms and an automatic abstraction generator.

In the previous example, *abs_List* may be described by a domain with three values $\{empty, one, more\}$, determining when the list is empty, has a single element or more, removing the information about the value of the stored elements. Then, the user has to provide the mapping $H : List \to abs\_List$ [3], as for example:

- $H(nil) = empty$, $H(cons(b, nil)) = one$ and $H(cons(b, cons(b', l))) = more$

Furthermore, he has to provide the definition of the abstracted functions, for instance:

- $abs\_cons(b, empty) = \{one\}$, $abs\_cons(b, one) = \{more\}$ and $abs\_cons(b, more) = \{more\}$
- $abs\_len(empty) = \{0\}$, $abs\_len(one) = \{1\}$ and $abs\_len(more) = \{2, 3, ..., maxLength\}$ [4]
- $abs\_head(l) = \{b_0, b_1\}$
- $abs\_tail(one) = \{empty\}$ and $abs\_tail(more) = \{one, more\}$

The mode *export* of the *Loader* lists the functions needed to complete the specification, we remember that the functions needed to manipulate sets are automatically generated by the tool. The mode *load* is used to import the definitions. The mode *auto* automatically performs the pointwise abstraction of the sorts and functions.

A *Modal*-LTS, generated from an abstract *Modal*-LPE (over and under) approximates the original system, if every pair of functions (f, abs_F) satisfies a formal requirement. The list of safety criteria is generated by the *Loader* in the format of the μCRL prover [29]. For the example above, the following conditions will be generated [5]:

- $\forall\, b, l :\ H(cons(b, l)) \in abs\_cons(b, H(l))$

[3] or $\alpha : \mathcal{P}(List) \to abs\_List$ depending on the type of abstraction selected by the user.
[4] Concrete lists are considered of bounded length ($maxLength$). Alternatively, one could abstract the sort *Nat* as well.
[5] The form of the safety conditions depends also on the type of abstraction.

- $\forall\, l:\ len(l) \in abs\_len(H(l))$
- $\forall\, l:\ head(l) \in abs\_head(H(l))$
- $\forall\, l:\ H(tail(l)) \in abs\_tail(H(l))$

3.4 Abstract Model Checking

To integrate the abstract interpretation techniques in the verification methodology we have to provide the relation between the satisfaction of a formula over the abstract system and its reflection to the concrete. This section describes the abstract model checking process for the homomorphic approach, the Galois Connection one may be defined in an equivalent way. Typically, the process is as follows:

(i) The user gives a *concrete* formula φ to prove in the concrete system (from now on M).

(ii) The arguments of the actions in φ, which are given as concrete sorts, are abstracted, resulting in abs_φ.

(iii) We check the satisfaction of abs_φ over the abstract model (abs_M, which is described by a *Modal*-LTS).

(iv) The result of the satisfaction is inferred to the concrete system. The inferences, as we will see, have some restrictions.

(step i) Concrete properties φ are described by the following logic (which is a very expressive subset of the regular alternating-free action-based μ-calculus [25]). There are three types of formulas, action (α), regular (β) and state formulas (φ), expressed by the following grammars:

$$\begin{aligned}
\alpha &::= T \mid F \mid \neg\,\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid a(\bar{d}) \\
\beta &::= \alpha \mid \beta_1.\beta_2 \mid \beta_1|\beta_2 \mid \beta* \mid \beta+ \\
\varphi &::= T \mid F \mid \neg\,\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\beta]\varphi \mid \langle\beta\rangle\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi
\end{aligned}$$

a stands for an action label from `ActNames`, and $\bar{d}$ for a, possibly empty, list of arguments. When the list is empty, we just write a. $a(\bar{d})$ matches transitions with the same action label and exactly the same arguments. T matches all actions with any argument, $\neg\,\alpha$ matches all actions but the ones matched by α. F matches no action, it could have been expressed by $\neg\, T$.

Regular formulas match sequences of actions; '.' stands for the concatenation operator, '|' is the choice operator, '$*$' is the transitive and reflexive closure operator, and '$+$' is the transitive closure operator.

The semantics of the state formulas are standard. $[\beta]\varphi$ states that all continuations by sequences matching β satisfy φ. $\langle\beta\rangle\varphi$ states that exists at

least one β sequence satisfying φ. μ and ν are the minimal and maximal fixpoint operators. We assume that states formulas are alternation free, in order to reuse the CADP toolset.

(step ii) As we have shown in the previous section, action arguments may be abstracted and/or lifted to sets during the abstraction process. In order to prove φ, we transform it to abs_φ by substituting every concrete argument of the actions by its abstract counterpart, i.e, $a(d)$ will be rewritten to $a(H(d))$.

(step iii) Following [20], an abstract formula is interpreted dually over an *Modal*-LTS, i.e. there will be two sets of states that satisfy it. A set of states that necessarily satisfy the formula and a set of states that possibly satisfy it. From the practical point of view, an interesting fact is that the 3-valued model checking problem can be easily transformed in two standard 2-valued problems. This allows the use of existing model checking tools such as the evaluator of the CADP toolset [13].

To do the translation, we follow the ideas of [5,14]. Basically, given a formula abs_φ we generate two different formulas abs_φ_{must} and abs_φ_{may}, the first one will be used to determine when a system *necessarily* satisfies a property and the second when it *possibly* does. They have the same structure as abs_φ but are built over $\texttt{ActNames}_{may/must}$ instead of over `ActNames`. For this purpose, we define two recursive operators $\mathcal{T}_{may}$ and $\mathcal{T}_{must}$. See below, the definition of the first one ($\mathcal{T}_{must}$ is dual):

- $\mathcal{T}_{may}(\neg\, abs\_\varphi) = \neg\mathcal{T}_{must}(abs\_\varphi)$
- Replace each occurrence of $[\beta]$ in abs_φ by $[\beta_{must}]$
- Replace each occurrence of $\langle\beta\rangle$ in abs_φ by $\langle\beta_{may}\rangle$
- For the rest of the cases, $\mathcal{T}_{may}$ is pushed inwards.

β_{may} replaces all occurrences of α by α_{may} which is defined as follows:

- if $\alpha = a(\bar{d})$ then $\alpha_{may} = a\_may(\bar{d})$.
- if $\alpha = T$ then $\alpha_{may} = T_{may}$. It matches all *may* actions.
- if $\alpha = F$ then $\alpha_{may} = \neg\,(T_{may})$. It matches actions that are not *may*. $\neg\,(T_{may})$ is equivalent to T_{must}.
- if $\alpha = \neg\,(\alpha')$ then $\alpha_{may} = \neg\,\alpha'_{may} \wedge T_{may}$. It matches all may actions that do not match α'_{may} .

These transformations are done in linear time. The difference between this approach and the one used by Godefroid and al. [14] is that instead of generating two different models and use one single formula, we use a single model and two versions of the formula. In general formulas are much smaller than the systems and their duplication is less expensive.

(step iv) The result of the abstract model checking process gives a 3-valued logic:

- abs_M necessarily satisfies abs_φ.
- abs_M possibly satisfies abs_φ but not necessarily satisfies abs_φ.
- abs_M not possibly satisfies abs_φ.

In the first case, we are able to infer the satisfaction of φ, i.e., $abs\_M \models \mathcal{T}_{must}(abs\_\varphi) \Rightarrow M \models_H abs\_\varphi$. In the third case, we are able infer the refutation of φ , i.e., $abs\_M \not\models \mathcal{T}_{may}(abs\_\varphi) \Rightarrow M \not\models_H abs\_\varphi$ However, the second case does not give any information about satisfaction or refutation of the property. The inference of the satisfaction or refutation of the concrete formulas is not straightforward. The reason is that by abstracting actions we have lost the exact information about concrete transitions.

Above, $\models_H$ defines the satisfaction of an abstract formula over a concrete system. The semantics of state and regular formulas do not change. We represent by $[\![abs\_\alpha]\!]_H$ the set of concrete actions that satisfy and abstract action formula. The semantics are given below:

$$
\begin{aligned}
[\![T]\!]_H &= Act & [\![F]\!]_H &= \emptyset \\
[\![abs\_\alpha_1 \wedge abs\_\alpha_2]\!]_H &= [\![abs\_\alpha_1]\!]_H \cap [\![abs\_\alpha_2]\!]_H \\
[\![abs\_\alpha_1 \vee abs\_\alpha_2]\!]_H &= [\![abs\_\alpha_1]\!]_H \cup [\![abs\_\alpha_2]\!]_H \\
[\![\neg\, abs\_\alpha']\!]_H &= Act \setminus [\![abs\_\alpha']\!]_H \\
[\![a(abs\_d)]\!]_H &= \{a(d) \mid H(d) = abs\_d\}
\end{aligned}
$$

We give, now, an example. let us consider the system of the following figure:

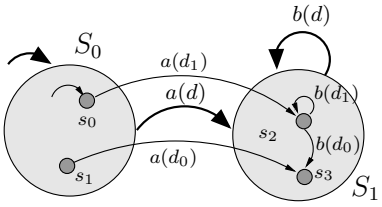


Fig. 3. Example of Abstract Model Checking.

The abstraction is built by mapping s_0 and s_1 to S_0, s_2 and s_3 to S_1 and d_0 and d_1 to d. We want to prove the following properties:

- *"It is possible to do a transition $a(d_0)$ from the initial state"* $s_0 \models \langle a(d_0)\rangle T$. The abstract version of the formula is $\langle a(d)\rangle T$, which trivially holds for S_0. Therefore, we can infer that exists x such that $H(x) = d$ for which $\langle a(x)\rangle T$ holds in s_0. In other words, $s_0 \models \langle a(d_0) \vee a(d_1)\rangle T$ which implies that $s_0 \models \langle a(d_0)\rangle T$ or $s_0 \models \langle a(d_1)\rangle T$.

- *"It is not possible to do a transition $b(d_0)$ from the initial state"* $s_0 \models [\, b(d_0) \,] F$. The abstract version of the formula is $[\, b(d) \,] F$, which trivially holds for S_0. Therefore, we can infer that for all x such that $H(x) = d$ implies $[\, b(x) \,] F$ holds in s_0. In other words, $s_0 \models [\, b(d_0) \vee b(d_1) \,] F$ which implies that $s_0 \models [\, b(d_0) \,] F$ and $s_0 \models [\, b(d_1) \,] F$.

In the first case, we have less information than we requested due to the abstraction, and we cannot infer the exact satisfaction or refutation of the original formula in the concrete model. In the second case we have enough to infer the exact result.

Note that in the special case of action labels without data arguments abs_φ will be equal to φ so the abstract model checking problem coincides with the classical theories based on state abstraction only.

4 Case Studies

The tool has been applied within the verification process of several case studies, as for example, to the study of JavaSpaces applications [11,33]. JavaSpaces is a coordination architecture that implements a shared repository that external agents can use to communicate by sharing objects. It provides extra support for implementing reliable applications. Systems may use transactions, a notification mechanism and timeouts on resource allocation. By abstracting the contents of the shared space to some significant values and the state of the external agents, we could prove some safety and liveness properties for more than 100 parallel processes, of a characteristic sort of JavaSpaces application. The characteristic application consists of a computationally intensive problem that is accomplished by breaking it into a number of smaller tasks that can be executed in parallel.

Furthermore, we have studied a real-life distributed system for lifting trucks (lorries, railway carriages, buses and other vehicles) [15]. The system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own micro-controller. On each lift there are some buttons that control its movement. The micro-controllers of the different lifts belonging to a system are connected to a 'cyclical' CAN (Controller Area Network). A safety property was proved correct for any number of lifts. These two case studies are documented in [27]

The tool was also used to prove liveness properties of the bounded retransmission protocol. The BRP is a simplified variant of a Philips' telecommunication protocol that allows to transfer large files across a lossy channel. Files are divided in packets and are transmitted by a sender through the channel. The receiver acknowledges every delivered data packet. Both data and con-

firmation messages, may be lost. The sender will attempt to retransmit each packet a limited number of times.

The protocol has a number of parameters, such as the length of the lists, the maximum number of retransmissions and the contents of the data, that cause the state space of the system to be infinite and limit the application of automatic verification techniques such as model checking. By abstracting some of these parameters, model checking could be successfully applied (see [31]). In a different context, the tool has been applied to perform simple abstractions to many different systems (see [26]).

5 Conclusion and Related Work

The existing tool closest to ours is αSpin [12] which provides an interface for abstracting PROMELA specifications. The user can select abstractions from a library. The tool produces an over-approximation of the system. The Bandera toolset [17] implements the same method of abstraction, furthermore it provides algorithms for *program slicing* and data dependencies analysis in order to automatically find suitable variables to abstract. Bandera generates PROMELA code from simple Java programs.

FeaVer [19] and abC [10] abstract C programs by hiding variables. The first one translates the code to PROMELA, furthermore it also allows the user to define his own abstractions, the latter abstracts directly the C code by implementing an extension of the GCC compiler. Java PathFinder [18], BeBop [2] and SLAM [1] use *predicate abstraction*. We refer to [9] for an extended overview of tools and techniques for abstract model checking.

All the enumerated tools only generate over-approximations, therefore there are only able to check for the satisfaction of *safety* properties. Our tool supports μ-calculus, therefore, we can use indistinctly *safety* and *liveness* properties. Furthermore, the transformation from LPEs to *Modal*-LPEs allows to reason about the abstract system on a syntactic level, and embeds all the techniques in the existing μCRL tools. Finally, an important feature that is not provided by any other tool is the possibility of abstracting action labels. Extra information about the tool can be found at:
http://www.cwi.nl/~miguel/Abstraction/.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, LNCS, vol. 2057, pages 103–122, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, LNCS, vol. 1885, pages 113–130, 2000.

[3] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *TCS* 5(2), pages 77–121, 1985.

[4] S. Blom, J. F. G., I. van Langevelde, B. L., and J.C. van de Pol. New developments around the μCRL tool set. In *ENTCS*, Elsevier, vol. 80, 2003.

[5] G. Bruns, and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *CAV*, LNCS, vol. 1633, pages 274–287, 1999.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *POPL*, ACM, pages 342–354, 1992 .

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, ACM, pages 238–252, 1977.

[8] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, 1996.

[9] D. Dams. Abstraction in software model checking: Principles and practice (tutorial overview and bibliography). In *SPIN*, LNCS, vol. 2318, pages 14–21, 2002.

[10] D. Dams, W. Hesse, and G. J. Holzmann. Abstracting C with abC. In *CAV*, LNCS, vol. 2404, pages 515–520, 2002.

[11] E. Freeman, S. Hupfer and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, 1999.

[12] M. M. Gallardo, J. Martinez, Pedro Merino, and E. Pimentel. αSPIN: Extending SPIN with Abstraction. In *STTT*, 5(2-3), pages 165 - 184, 2004.

[13] H. Garavel, F. Lang, and R. Mateescu, An Overview of CADP 2001, In EASST Newsletter, vol. 4, pages 13–24, 2002.

[14] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, LNCS, vol. 2154, pages 426–440, 2001.

[15] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. In *JLAP*, 56(1-2):21-56, 2003.

[16] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In *ACP*, Workshops in Computing Series, pages 26–62, 1995.

[17] J. Hatcliff, M. B. Dwyer, C. S. Pasareanu, and Robby. Foundations of the Bandera abstraction tools. In *The Essence of Computation*, LNCS, vol. 2566, pages 172 – 203, 2002.

[18] K. Havelund and J. Skakkebaek. Applying Model Checking in Java Verification. In *SPIN*, LNCS, vol. 1680, pages 216–232, 1999.

[19] G.J. Holzmann and M.H. Smith. A practical method for verifying event-driven software. In *ICSE*, ACM, 1999.

[20] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *ESOP*, LNCS, vol. 2028, pages 155–169, 2001.

[21] N. D. Jones and F. Nielson. *Abstract Interpretation: A Semantics-Based Tool for Program Analysis.* In *Handbook of Logic in Computer Science*, Oxford University Press, pages 527–636, 1995.

[22] D. Kozen. Results on the propositional μ-calculus. In *ICALP*, LNCS vol. 140, pages 348–359, 1982.

[23] K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, IEEE, pages 203–210, 1988.

[24] D.E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.

[25] R. Mateescu. *Verification des proprietes temporelles des programmes paralleles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[26] S. Orzan, J.C. van de Pol, and M. Valero Espada. A state space distribution policy based on abstract interpretation. In *PDMC*, ENTCS, pages to Appear, 2004.

[27] J. Pang, J.C. van de Pol, and M. Valero Espada. Abstraction of parallel uniform processes with data. In *SEFM*, IEEE, pages to Appear, 2004.

[28] D. Schmidt. Binary relations for abstraction and refinement, 1999.

[29] J.C van de Pol. A prover for the μCRL toolset with applications. Technical Report SEN-R0106, CWI, 2001.

[30] J.C. van de Pol and M. Valero Espada. An abstract interpretation toolkit for μCRL (extended version). Technical Report to Appear, CWI, 2004.

[31] J.C. van de Pol and M. Valero Espada. Modal abstraction in μCRL. In *AMAST*, LNCS, vol. 3116, pages 409–425, 2004.

[32] J.C. van de Pol and M. Valero Espada. Modal abstraction in μCRL (extended version). Technical Report SEN-R0401, CWI, 2004.

[33] J.C. van de Pol and M. Valero Espada. Verification of JavaSpaces parallel programs. In *ACSD*, IEEE, pages 196–205, 2003.