Form Method Syst Des (2007) 30:249–273
DOI 10.1007/s10703-006-0029-7

An abstract interpretation toolkit for μCRL

Miguel Valero Espada · Jaco van de Pol

Published online: 29 December 2006

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 Model checking · Process algebra · muCRL · Abstract interpretation

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 [34, 37], we have presented the theoretical framework for abstracting μCRL [19] specifications. μCRL is a language that combines the Algebra of Communicating Processes (ACP) [4] with Abstract Data Types (ADT). In this paper, we will describe the toolkit that implements the theory.

Semantically, abstractions are represented by *Modal Labelled Transition Systems* (*Modal*-LTS) [28], 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

M. V. Espada (✉)
Dept. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, E-28040, Madrid, Spain
e-mail: mvaleroe@pdi.ucm.es

J. van de Pol
Centrum voor Wiskunde en Informatica, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: Jaco.van.de.Pol@cwi.nl

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 [26] from the abstract to the concrete system.

μCRL specifications consist of an ADT part, defining data operations, and a process specification part, specifying an event-based reactive system. Processes are defined using among others sequential and parallel composition, non-deterministic choice and hiding. Furthermore, atomic actions, conditions and recursion are present, and may depend on data parameters. The μCRL toolset transforms specifications to *linear process equations* (LPE), by eliminating parallel composition and hiding efficiently.

We implement abstract interpretation as a transformation from LPEs to *Modal*-LPEs. *Modal*-LPEs capture the extra non-determinism arising from abstract interpretation. They allow a single transition to lead to a *set* of states with a *set* of action labels. In [34], we have shown that the *Modal*-LTS generated from a *Modal*-LPE is a proper abstraction of the LTS generated from the original LPE.

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* [1] in which only the value of some conditions is retained and propagated over the dependent predicates of the specification. *Program slicing* [21] 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.

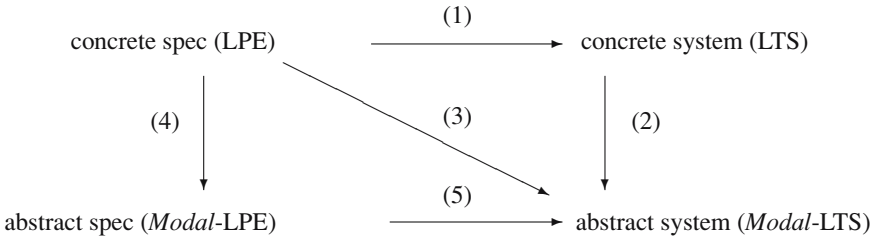
Our tool implements the automatic *pointwise* abstraction and, moreover, assists the user to create his own abstractions (predicate abstraction and program slicing are not implemented in the tool). The tool supports the use of two mainstream techniques for data abstraction. One proposed by Long, Grumberg and Clarke [8, 14], 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 [9, 10, 24, 25], in which data is related by Galois Connections. A lifting mechanism is also implemented, which allows to automatically build Galois Connections from homomorphisms, see [35].

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 [5] and CADP [16] 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 give an overview of the organisation and the main functionalities of our tool, then we introduce the basic theoretical concepts of abstract interpretation. We continue by describing the different components of the tool in more detail and we conclude by presenting a case study that was successfully analysed using the tool. The paper concludes with a comparison with other related tools.

2 ToolKit

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



From a concrete system, encoded as a linear process equation (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 [6]) that manipulate linear equations, for instance state space reduction, elimination of dead code, confluence analysis and symbolic model checking.

2.1 Overview of the tool

The following figure describes the tool architecture. Our three new tools (oval boxes with bold lines) are the abstractor, the abstraction loader, and the abstract model checker. They are shortly described below. The other tools (oval boxes) are from the CADP toolset (model checker) or from the μCRL toolset (instantiator, theorem prover and optimization tools on LPEs). The existing μCRL and CADP tools can be reused without any change, because we encode all objects as standard LPEs, LTSs, or regular mu-calculus formulas.

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 mentioned already, 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 6, we provide the guidelines to model check and to infer the results.

3 Theoretical background

3.1 Transition systems with may and must steps

The semantics of a system can be captured by a *Labelled Transition System* (LTS).

Definition 1. We define a *Labelled Transition System* (LTS) as a tuple $(S, Act, \rightarrow, s_0)$ in which S is a non-empty set of states, Act a non-empty set of transition labels, $\rightarrow$ is a possibly infinite set of transitions and s_0 in S is the initial state. A transition is a triple $s \xrightarrow{a} s'$ with $a \in Act$ and $s, s' \in S$.

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 [28]. The formal definition extends the definition of LTSs by considering the two modalities.

Definition 2. A *Modal Labelled Transition System* (*Modal*-LTS) is a tuple $(S, Act, \rightarrow_{\diamond}, \rightarrow_{\Box}, s_0)$ where S, Act and s_0 are as in definition 1 and $\rightarrow_{\diamond}$, $\rightarrow_{\Box}$ 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 $x \in \{\diamond, \Box\}$. We require that every *must*-transition is a *may*-transition ($\xrightarrow{a}_{\Box} \subseteq \xrightarrow{a}_{\diamond}$).

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

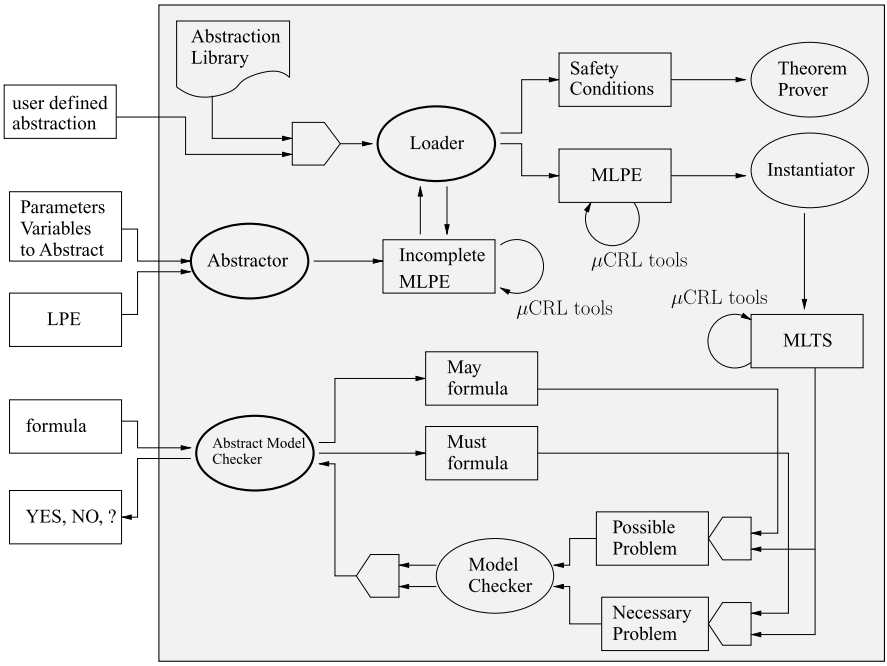


Fig. 1 Tool architecture

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 [34], we have presented the complete formal framework for abstracting, now we give an example (see Fig. 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 Fig. 1, we have the abstract *must* transition $S_0 \rightarrow_\Box S_0$ and $S_1 \rightarrow_\Box S_2$. 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 also be 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.

Figure 2 is an informal description of abstraction using homomorphisms. In this case, we have defined two mapping functions from concrete states to abstract states and from concrete action labels to abstract ones. Another approach is based on Galois Connections. Two functions α and γ over two partially ordered sets $(P, \subseteq)$ and $(Q, \preccurlyeq)$ such that $\alpha : P \rightarrow Q$ and $\gamma : Q \rightarrow P$ form a Galois Connection [31] if and only if the following conditions hold:

1. α and γ are total and monotonic.
2. $\forall p : P \cdot p \subseteq \gamma \circ \alpha(p)$.
3. $\forall q : Q \cdot \alpha \circ \gamma(q) \preccurlyeq q$.

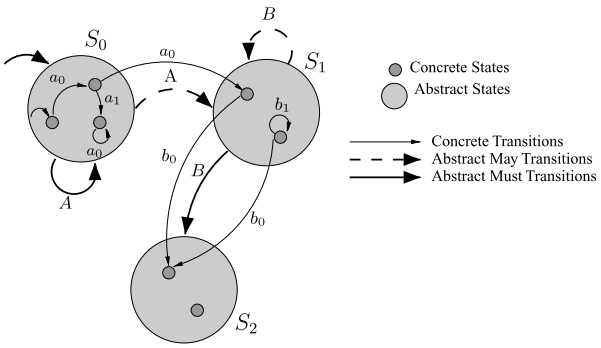


Fig. 2 Example of modal abstraction of an LTS

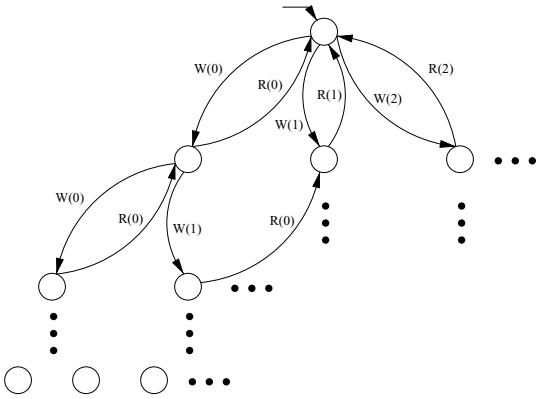


Fig. 3 Concrete transition system of a bounded Buffer

Concrete and abstract domains would be represented by partially ordered sets. The order is a relation based on the precision of the information contained in the elements of the domain. The Galois Connection approach permits to relate concrete values to more than one abstract value, which allows to define more expressive abstractions. Technical details and definitions can be found in [37]. Now, we include a small example to illustrate the difference between the two theories. Figure 3 describes a bounded buffer (with capacity greater than 1) containing natural numbers, the action W is used to write new items and R to read.

Figure 4 presents a homomorphic abstraction in which the initial state is mapped to the abstract state e (empty), the states in which the buffer is full are mapped to f (full) and the rest to m (middle). On the left hand side of the figure only states are abstracted, on the right hand side both, states and action labels are abstracted.

We see, that by abstracting only states, the system cannot be completely represented, even if the set of states is finite, because it is infinitely branching. Abstraction frameworks only based on abstraction of states cannot handle this kind of problems. In the final system, by the combination of both abstractions, we have removed all the information about the values that are in the buffer and the transferred data, only preserving the information about whether the buffer is empty, full or neither of them. This abstraction allows to have a small finite model, which keeps some information about the original; Section 6 describes which properties are preserved. The example clearly illustrates the importance of the abstraction of action labels to avoid infinitely branching abstractions.

Figure 5 presents an abstraction using the Galois Connection approach. The abstract domain of states contains nF (nonFull) that represents the states in which the buffer is

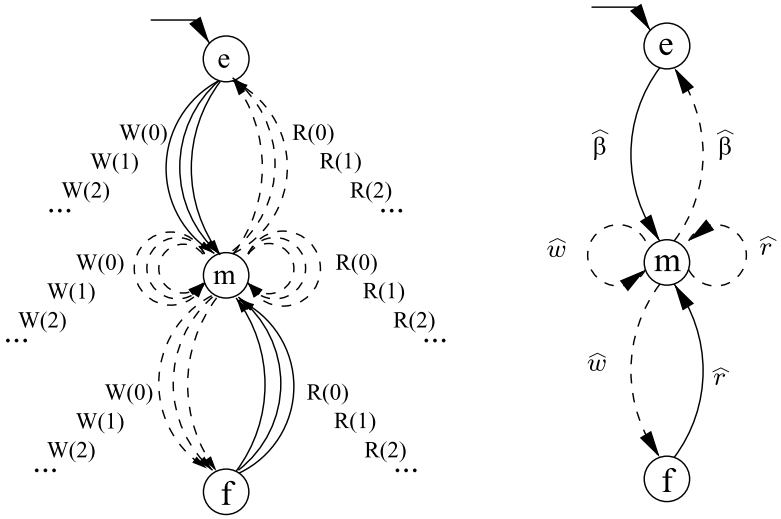


Fig. 4 Abstraction using homomorphisms

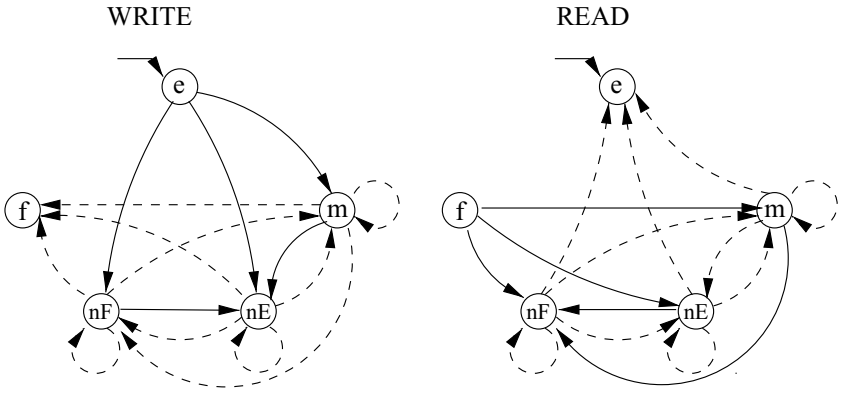


Fig. 5 Abstraction using Galois connections

not full, and *nE* (nonEmpty). The order defined over the abstract lattices gives a relation about the precision of the information contained in the values. For example, *empty* has more accurate information than *nonFull*.

In general, the Galois Connection approach is more precise than the homomorphic one. For instance, in Fig. 5 we can prove a property saying that from an empty buffer it is certainly possible to write two entries[1] because from the state *empty* there is a write *must*-transition to the state *middle* and from the later there is also a write *must*-transition to *nonEmpty*. However, in the homomorphic abstraction of Fig. 4 the same property does not hold because there is no *must*-transition outgoing from *middle*.

[1] As is presented in Section 6, this property depends on the write *must*-transitions. It is true in the concrete because we have assumed that the size of the buffer is greater than one.

3.2 Process equations with and without modalities

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*,[2] 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.

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.

Atomic actions may have data parameters. The operator $|$ allows synchronous parameterised communication. If two actions are able to synchronise we can force that they occur always in communication using the encapsulation operator (∂_H). The operator τ_I hides enclosed actions by renaming into τ actions. The initial behaviour of the system can be specified with the keyword **init** followed by a process term:

$$\mathsf{System} = \tau_I \partial_H(p_0 \parallel p_1 \parallel \ldots)$$
$$\mathsf{init\ System}$$

A data type consists of a many-sorted signature in which a set of function symbols, and a list of axioms are declared. For every specification, we assume the existence of the Boolean sort (*Bool*), with the constants true and false (T and F) and their standard functions. Data terms are interpreted over a model, which is a mathematical structure consisting of a universe of values and total functions, in which all axioms are valid. The precise syntax and semantics of μCRL are given in [19].

The following μCRL process specifies a bounded buffer implemented using a list. 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 than 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.

$$\begin{aligned} \mathit{Buffer}(l : \mathit{List}) = {} & \sum_{b:\mathit{Bit}} \mathit{write}(b) \cdot \mathit{Buffer}(\mathit{add}(b, l)) \lhd \mathit{lt}(\mathit{len}(l), \mathit{MAX}) \rhd \delta \\ & + \mathit{read}(\mathit{head}(l)) \cdot \mathit{Buffer}(\mathit{tail}(l)) \lhd \mathit{not}(\mathit{isEmpty}(l)) \rhd \delta \end{aligned}$$

This process definition assumes the specification of natural numbers (with the standard operations equality *eq*, successor *succ* and predecessor *pred*), and bits. The data type *List*

[2] From now on we will use *ActNames* (*ActN* for short) to refer just to the labels and *Act* to the set of action labels together with the arguments

can then be defined as follows:

sort	$List, Bool, Nat$	**var**	$l : List$
func	$emptyList :\rightarrow List$		$b, b' : Bit$
	$cons : Bit \times List \rightarrow List$	**rew**	$add(b, emptyList) = cons(b, emptyList)$
	$add : Bit \times List \rightarrow List$		$add(b, cons(b', l)) = cons(b', add(b, l))$
map	$head : List \rightarrow Bit$		$head(cons(b, l)) = b$
	$tail : List \rightarrow List$		$tail(cons(b, l)) = l$
	$len : List \rightarrow Nat$		$len(emptyList) = 0$
	$isEmpty : List \rightarrow Bool$		$len(cons(b, l)) = succ(len(l))$
			$isEmpty(l) = eq(0, len(l))$

In the specification of the data type we have used the following keywords: **sort** to declare the name of the data type, **func** and **map** to declare sorted operations (constructors and defined functions, respectively), **var** to declare auxiliary variables and **rew** to define the defining equations of the type.

Every μCRL system can be transformed to a special format, called *Linear Process Equation* or *Operator* [20, 36]. An LPE (see definition below) is a single μCRL process, which represents the complete system and from which parallel composition, encapsulation and hiding have been eliminated.

$$X(d : D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i[d, e_i]) \cdot X(g_i[d, e_i]) \lhd c_i[d, e_i] \rhd \delta \tag{1}$$

In the definition, d denotes a vector of parameters d of type D that represents the state of the system at every moment. We use the keyword *init* to declare the initial vector of values of d. Action labels a_i are selected from a set of action names *ActNames*. The process is composed by a finite number I of summands, every summand i, has a list of local variables e_i, of possibly infinite domains, and it is of the following form: a condition $c_i[d, e_i]$, if the evaluation of the condition is true the process executes the action a_i with the parameter $f_i[d, e_i]$ and will move to a new state $g_i[d, e_i]$, which is a vector of terms of type D. $f_i[d, e_i]$, $g_i[d, e_i]$ and $c_i[d, e_i]$ are terms built recursively over variables $x \in [d, e_i]$, applications of function over terms $t = f(t')$ and vectors of terms. For example, we compose two *buffers* in parallel, as follows:

$\mathsf{comm}\ read_0|write_1 = w$
$\mathsf{System} = \tau_{\{w\}}\partial_{\{read_0, write_1\}}(Buffer_0(emptyList) \parallel Buffer_1(emptyList))$
$\mathsf{init\ System}$

where $Buffer_0$ is equal to the process *Buffer* in which l is renamed to $l0$, *write* to $write_0$ and *read* to $read_0$ (similar for $Buffer_1$). We obtain the following linear form

$$\begin{aligned} X(l0, l1\colon List) = \sum_{b:Bit} & write_0(b) \cdot X(add(b, l0), l1) \lhd lt(len(l0), MAX) \rhd \delta \\ & + \tau . X(tail(l0), add(head(l0), l1)) \\ & \lhd not(isEmpty(l0)) \wedge lt(len(l1), MAX) \rhd \delta \\ & + read_1(head(l1)) \cdot X(l0, tail(l1)) \lhd not(isEmpty(l1)) \rhd \delta \end{aligned}$$

To every LPE specification corresponds a labelled transition system. The semantics of the system described by an LPE are given by the following rules:

- $s_0 = \text{init}_{lpe}$
- $s \xrightarrow{a} s'$ if and only if exists $i \in I$ and exists $e : E_i$ such that $c_i[s, e] = \mathsf{T}$, $a_i(f_i[s, e]) = a$ and $g_i[s, e] = s'$

The LTS corresponding to the *Buffer* LPE can be generated for any finite value of the constant *MAX*. The *Buffer* is modelled to contain bits, which makes it finite. If we change the specification to have a container of natural numbers then the system will have an infinitely branching behaviour, as the one presented in Fig. 3.

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.

$$X(d : \mathcal{P}(\hat{D})) = \sum_{i \in I} \sum_{e_i : E_i} a_i(F_i[d, e_i]) \cdot X(G_i[d, e_i]) \lhd C_i[d, e_i] \rhd \delta \tag{2}$$

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 i: C_i returns a non-empty set of Booleans, G_i a non-empty set of states. Actions are parameterised with non-empty sets of values F_i, as well. From a *Modal*-LPE we can generate a *Modal Labelled Transition System* following these semantic rules:

- $S_0 = \text{init}_{mlpe}$
- $S \xrightarrow{A}_{\Box} S'$ if and only if exists $i \in I$ and exists $e \in E_i$ $(e \neq \bot)$ such that $\mathsf{F} \notin C_i[S, e]$, $A = a_i(F_i[S, e])$ and $S' = G_i[S, e]$
- $S \xrightarrow{A}_{\Diamond} S'$ if exists $i \in I$ and exists $e \in E_i$ $(e \neq \bot)$ such that $\mathsf{T} \in C_i[S, e]$, and $A = a_i(F_i[S, e])$ and $S' = G_i[S, e]$

Modal-LPEs allow to capture in a uniform way both approaches: Galois Connection and Homomorphism. In the second case, we restrict the rules by letting S_0, S, A and S' be only singleton sets. The next section describes the tool and the methodology to apply abstract interpretation of process algebraic specifications.

4 Abstractor

The Abstractor gets as input a *Linear Process Equation* and a set of parameters and local variables to abstract. Alternatively, the user can provide a list of sorts. In this case all parameters and variables of the selected sorts will be abstracted. Subsequently, the input is transformed conforming the user selection by replacing the different symbols that appear in the specification by their abstract counterparts.

We have seen that data terms $f_i(d, e_i)$, $g_i(d, e_i)$ and $c_i(d, e_i)$ are composed by function symbols, parameters and local variables. Based on the parameters and variables selected by the user some (or all) function symbols are replaced by their abstract definition. In this section, we present the abstraction criteria that the tool implements.

4.1 Abstraction of function symbols

The *Abstractor* will traverse the process specification, transforming the function symbols according to the user input. In case the arguments of a function are modified the tool will generate a new signature for the function and will replace the old one.

We recall that a way of capturing the non-determinism induced by the abstracted functions is using sets of values. For instance, we can consider the abstraction of the integers to their sign, i.e., $\{neg, zero, pos\}$; Intuitively, the definition of the abstract successor of *zero* and *pos* will in both cases be *pos*. However the abstract successor of *neg* can be either *neg* or *zero*, therefore, the sort of the abstract successor will be a set of abstract integers. We define *lifting* to be the operation of replacing single values to sets of values. We give now the rules for abstracting and lifting function symbols. Let us consider the function f: $S_0 \times \cdots \times S_{n-1} \to S_n$:

1. If there is a data term in the process specification in which the ith argument of f is abstracted then the signature of the function will change according to the following rules:

   (a) All sorts $S_j = S_i$ with $j \in [0, \ldots, n-1]$ will be abstracted.
   (b) If $S_n = S_i$ then the target sort of f will be abstracted and lifted.
   (c) If $S_n \neq S_i$ then the target sort of f will be lifted.

Let us consider again the abstraction of integers, with the following functions: $succ : Int \to Int$, $+ : Int \times Int \to Int$ and $< : Int \times Int \to Bool$. Then, following the above presented rules:

- If there is a data term in which the argument of *succ* is abstracted then the target sort of the abstract version of *succ* will be abstracted and lifted, i.e., $abs\_succ : abs\_Int \to \mathcal{P}(abs\_Int)$.
- If one argument of $+$ is abstracted then the other argument will be abstracted as well and the target sort will be abstracted and lifted, i.e., $\widehat{+} : abs\_Int \times abs\_Int \to \mathcal{P}(abs\_Int)$
- If one argument of $<$ is abstracted then the other argument will be abstracted as well and the target sort will be lifted, i.e., $\widehat{<} : abs\_Int \times abs\_Int \to \mathcal{P}(Bool)$

Let us consider, now, the abstraction of lists of type D, with the following standard functions: $cons : D \times List \to List$ and $head : List \to D$. Then, following the rules:

- If the first argument of *cons* is abstracted then the new signature will be: $abs\_cons : abs\_D \times List \to \mathcal{P}(List)$.
- If the second argument of *cons* is abstracted then the new signature will be: $abs\_cons : D \times abs\_List \to \mathcal{P}(abs\_List)$.
- If both arguments of *cons* are abstracted then the new signature will be: $abs\_cons : abs\_D \times abs\_List \to \mathcal{P}(abs\_List)$.
- If the argument of *head* is abstracted, the new signature will be: $abs\_head : abs\_List \to \mathcal{P}(D)$.

Furthermore:

2. If there is a data term in which the ith argument of f is lifted then:

   (a) The target sort of f will be lifted.

For example:

- If the argument of *succ* is lifted (but not abstracted) then the target will be lifted, i.e., $succ : \mathcal{P}(Int) \to \mathcal{P}(Int)$.
- If the argument of *succ* is lifted and abstracted then the target be abstracted and lifted, i.e., $abs\_succ : \mathcal{P}(abs\_Int) \to \mathcal{P}(abs\_Int)$. (Note that this is the result of the application of rule 1.b).

The tool will automatically generate auxiliary functions and equations to manipulate sets, by providing the *pointwise* lifting of the not lifted ones. For instance, for a function f in which the ith sort has been lifted and the rest remains unlifted, i.e., f: $D_0 \times \cdots \times \mathcal{P}(D_i) \times \cdots D_{n-1} \to \mathcal{P}(D_n)$, the following equation will be generated:

- Let X be of type $\mathcal{P}(D_i)$ and x of type D_i
- $\mathsf{f}(\cdots, X, \cdots) = \cup\{\mathsf{f}(\cdots, x, \cdots) \mid x \in X\}$[3]

4.2 Abstraction of parameters and variables

The user selects the list of parameters and variables that he wants to abstract. The choice may influence the sorts of other related parameters. To determine the sorts of the abstract specification we follow the next rules:

3. If a parameter $d_i : D_i$ is selected to be abstracted then its sort will change. The new sort of the abstract parameter will be the powerset of the abstract version of its concrete sort, i.e., $d_i : \mathcal{P}(abs\_D_i)$. The explanation why abstracted parameters are also lifted is that after every recursion, the updated values of the parameters are computed from functions. And, as we have seen in the previous section, to capture the extra non-determinism, functions are lifted to sets. Therefore, the specification may contain assignments in which parameters receive sets of values.
4. If a variable $e_{a_i} : E_{a_i}$ is selected to be abstracted then its sort is changed to the abstract version of the concrete one, i.e., $e_{a_i} : abs\_E_{a_i}$. In this case, we do not lift the sorts of the values to powersets because their values are not induced from any abstracted data term.
5. If a parameter $d_i : D_i$ is not selected to be abstracted but there is an assignment of a data term in which appears an abstract parameter or an abstract variable, or a lifted or abstracted function then the parameter is lifted, i.e., $d_i : \mathcal{P}(D_i)$.

4.3 Abstraction of sorts

For every abstracted sort the user will have to provide the abstract domain and the relation with the concrete one. The tool supports three ways of relating the domains, the homomorphic and the Galois Connection approach and also the combination of them that consists of the lifting of a homomorphism to a Galois Connection. To define a Galois Connection from an homomorphism H we proceed as follows:

If we have the concrete domain D and the abstract abs_D, then we build the abstract lattice as the power set of abstract values $\mathcal{P}(abs\_D)$ ordered by the set inclusion operator. Furthermore, we define $\alpha : \mathcal{P}(D) \to \mathcal{P}(abs\_D)$ and $\gamma : \mathcal{P}(abs\_D) \to \mathcal{P}(D)$ as:

- $\alpha(S) = \{H(s) \mid s \in S\}$
- $\gamma(\hat{S}) = \{s \mid \exists\, abs\_s \in abs\_S \wedge H(s) = abs\_s\}$

[3] Note that function symbols are overloaded.

Not all Galois Connections can be represented by a lifted homomorphism, however the use of the lifted homomorphism may be convenient to perform rapid and powerful abstractions. The lifting technique reduces the number of abstract definitions that the user has to provide to specify the abstract system.

For every abstracted sort abs_D, the tool will generate the signature of the functions $alpha : \mathcal{P}(D) \to \mathcal{P}(abs\_D)$, $gamma : \mathcal{P}(abs\_D) \to \mathcal{P}(D)$ and $\preccurlyeq : \mathcal{P}(abs\_D) \times \mathcal{P}(abs\_D) \to \mathcal{P}(Bool)$. The first one represents the abstraction function, the second one the concretisation function and the third the order on the abstract domain. The user selects one of the three types of abstraction. Then in case of homomorphism or lifted homomorphism the following auxiliary function definitions will be generated:

- $H : D \to abs\_D$
- $H^{-1} : abs\_D \to \mathcal{P}(D)$
- $alpha(X) = \{H(x) \mid x \in X\}$
- $gamma(abs\_X) = \cup\{H^{-1}(abs\_x) \mid abs\_x \in abs\_X\}$

In case the user selects Galois Connections the tool will generate:

- $\alpha : \mathcal{P}(D) \to abs\_D$
- $\gamma : abs\_D \to \mathcal{P}(D)$
- $alpha(X) = \{\alpha(\{x\}) \mid x \in X\}$
- $gamma(abs\_X) = \cup\{\gamma(abs\_x) \mid abs\_x \in abs\_X\}$
- $lt(abs\_X, abs\_Y) = \forall\, abs\_x \in abs\_X\, \exists\, abs\_y \in abs\_Y . abs\_x \preccurlyeq abs\_y$

In the first case, the user will have to provide H and H^{-1} (if there are conflicting cases, see next section). In the second α, γ and the order $\preccurlyeq$. H^{-1} and γ do not have to be provided in general, they are used to solve the conflicts. In some cases, it is not possible to define them because they produce infinite sets. The need for the concretisation functions (H^{-1} and γ) can be always avoided by abstracting more parameters or variables, in which case no definition is required.

4.4 Type conflicts

Abstraction of data terms is done by abstracting first the parameters and variables that appear inside the terms, and then by propagating the abstraction to the function symbols according to the rules specified above. The abstraction may raise some type conflicts. We list below the different conflicts and how they are resolved:

6. There is an assignment in which a parameter of sort $\mathcal{P}(D)$ gets a term d of sort D. Then d is replaced by $\{d\}$.
7. There is an assignment in which a parameter or an argument of a function of sort $\mathcal{P}(abs\_D)$ gets a term d of sort D. Then d is replaced by $alpha(\{d\})$
8. There is an assignment in which a parameter of sort $\mathcal{P}(D)$ gets a term d of sort $\mathcal{P}(abs\_D)$. Then d is replaced by $gamma(d)$.
9. If the data term C_a of a condition is abstracted then it is replaced by $gamma(C_a)$.

4.5 From LPEs to *Modal*-LPEs

The *Abstractor* replaces the data terms of the LPEs by their abstract counterparts, producing *Modal*-LPEs. The user can select the parameters and variables to abstract, then the abstraction is propagated over the data terms of the specification, with the rules that we presented in the

previous sections. Let us reconsider the example of the bounded buffer, that was explained in Section 3.2:

$$Buffer(l : List) = \sum_{b:Bit} write(b) \cdot Buffer(add(b, l)) \lhd lt(len(l), MAX) \rhd \delta$$
$$+ read(head(l)) \cdot Buffer(tail(l)) \lhd not(isEmpty(l)) \rhd \delta$$

Recall that the concrete specification has the following signature:

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

If the user selects the parameter l to be abstracted then the propagation of the abstraction will yield the following signature[4]:

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

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. 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.

4.6 From *Modal*-LPEs to LPEs$_{may/must}$

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 *ActNames* (or *ActN* for short) be the set of action labels of a *Modal*-LPE. We define $ActNames_{may/must} = \{a\_may \mid a \in ActNames\} \cup \{a\_must \mid a \in ActNames\}$. 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

[4] The complete output of the *Abstractor* for this example is given in the next section.

homomorphisms is:

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

(*MLPE to LPE (H)*)

The patterns are derived from the semantics of *Modal*-LPEs presented in Section 3.2. 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. The pattern for Galois Connections and lifted homomorphisms is:

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

(*MLPE to LPE (GC)*)

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

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

In [37], we characterized the relationship between *Modal*-LPEs and $\text{LPE}_{may/must}$, and between LPEs and *Modal*-LPEs. Basically, we proved that if the safety conditions (cf. Section 5) between concrete and abstract data hold, then the result of the abstract process is a sound approximation of the original system. Therefore, one can infer properties expressed in μ-calculus from the abstract system to the concrete (cf. Section 6).

5 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.

The example of Section 4.5 can be continued, by defining *abs_List* to be the domain with three values {*empty, one, more*}. These values indicate whether the list is empty, has a single element or more elements. All information about the value of the stored elements is removed. Then, the user has to provide the mapping $H : List \rightarrow abs\_List$,[5] as for example:

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

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

- $abs\_add(b, empty) = \{one\}$, $abs\_add(b, one) = \{more\}$ and $abs\_add(b, more) = \{more\}$
- $abs\_len(empty) = \{0\}$, $abs\_len(one) = \{1\}$ and $abs\_len(more) = \{2, 3, \ldots, maxLength\}$
- $abs\_head(l) = \{b_0, b_1\}$
- $abs\_tail(one) = \{empty\}$ and $abs\_tail(more) = \{one, more\}$

The *Loader* can *export* the signatures of the functions that are needed to complete the specification. We recall that the functions needed to manipulate sets are automatically generated by the tool. It can also be used to import user-provided abstract definitions, or to perform automatically the pointwise abstraction of sorts and functions.

A *Modal*-LTS, generated from an abstract *Modal*-LPE approximates the original system, if every pair of functions (f, abs_F) satisfies a formal requirement. The user must prove the safety requirements to ensure the soundness of the abstraction. The list of safety conditions is generated by the *Loader* in the format of the μCRL prover [33]. The form of the safety conditions depends also on the type of abstraction. For the example above, choosing the Galois Connection, the following conditions will be generated.

- $\forall\ b, abs\_l :\ lt(alpha(add(b, gamma(\{abs\_l\}))), abs\_add(b, abs\_l))$
- $\forall\ abs\_l :\ lt(alpha(len(gamma(\{abs\_l\}))), abs\_len(abs\_l))$
- $\forall\ abs\_l :\ lt(alpha(head(gamma(\{abs\_l\}))), abs\_head(abs\_l))$
- $\forall\ abs\_l :\ lt(alpha(tail(gamma(\{abs\_l\}))), abs\_tail(abs\_l))$

For the (lifted)-homomorphisms, the safety conditions are reduced to:

- $\forall\ b, l :\ H(add(b, l)) \in abs\_add(b, H(l))$

[5] or $\alpha : \mathcal{P}(List) \rightarrow abs\_List$ depending on the type of abstraction selected by the user.

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

If the safety conditions hold for the function symbols then, by construction, they will hold for the full data terms. Therefore, instead of proving the safety conditions for every guard, action and next-state in a process specification we can prove in general that the abstract data specification satisfies the "safety conditions" and then infer that any particular system does as well. This would allow to reuse abstract specifications of the data into different systems and create libraries of abstractions.

6 Abstract model checking

To integrate the abstract interpretation techniques in the verification methodology we have to provide a 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 case can be defined in a similar way). Typically, the process is as follows:

1. The user gives a *concrete* formula φ to prove in the concrete system M.
2. The arguments of the actions in φ, which are given as concrete sorts, are abstracted, resulting in abs_φ.
3. We check the satisfaction of abs_φ over the abstract model (abs_M, which is described by a *Modal*-LTS).
4. 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 using the regular alternating-free action-based μ-calculus [30]. The logic embeds regular expressions with modal and fixpoint operators. There are three types of formulas, action (α), regular (β) and state formulas (φ), expressed by the following grammars:

$$\alpha ::= \mathsf{T} \mid \mathsf{F} \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid a(\bar{d}) \mid reg - \exp$$
$$\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1 | \beta_2 \mid \beta* \mid \beta+$$
$$\varphi ::= \mathsf{T} \mid \mathsf{F} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\beta]\varphi \mid \langle\beta\rangle\varphi \mid Y \mid \mu Y \cdot \varphi \mid \nu Y \cdot \varphi$$

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 $a \cdot a(\bar{d})$ 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 $\neg\mathsf{T} \cdot \alpha_1 \wedge \alpha_2$ matches all action that match α_1 and $\alpha_2 \cdot \alpha_1 \vee \alpha_2$ matches all action that match α_1 or α_2. Action formulas can also be expressed as regular expressions that match using the standard syntactic rules.

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 is standard. $[\beta]\varphi$ states that all continuations by sequences matching β satisfy $\varphi \cdot \langle\beta\rangle\varphi$ states that there exists at least one β sequence satisfying $\varphi \cdot \mu$ and ν are the minimal and maximal fixpoint operators.

(step ii) As we have shown in section 4, 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 replacing every concrete argument of the actions by its abstract counterpart, i.e, $a(d)$ will be rewritten to $a(H(d))$.

(step iii) Following [24], an abstract formula is interpreted dually over a *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 to two standard 2-valued problems. This allows the use of existing model checking tools such as the evaluator of the CADP toolset [16].

To do the translation, we follow the ideas of [7, 17]. 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 $ActNames_{may/must}$ instead of over *ActNames*. For this purpose, we define two recursive operators $\mathcal{T}_{may}$ and $\mathcal{T}_{must}$. See below for 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 $\alpha = \mathsf{T}$ then $\alpha_{may} = T_{may}$. It matches all *may* actions.
- if $\alpha = \mathsf{F}$ 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 time linear in the size of the formula. The difference between this approach and the one used by Godefroid et al. [17] is that instead of generating two different models and using one single formula, we use a single model and two versions of the formula. In general formulas are much smaller than systems and their duplication is less expensive. We present, below, some typical properties, in the abs_φ_{must} form:

Deadlock freedom, with regular expressions:

(P1): ['. *may .*'*] ⟨ '. *must .*'* ⟩ T

The dot . in the regular expressions inside the action formulas matches any character, therefore .* matches any number of occurrences of any character.

Deadlock freedom, with fixed point operators:

(P2): ν X · (⟨ '. *must .*' ⟩ T∧ ['. *may .*'] X)

No execution sequence leads to a:

(P3): ['. *may .*'* . 'a$_{may}$'] F

There exists a sequence leading to a:

(P4): ⟨ '. *must .*'* . 'a$_{must}$' ⟩ T

All sequences lead to a:

(P5): μ X · (⟨ '. *must .*' ⟩ T∧ [¬('a$_{may}$' ∧ '. *may .*')] X)

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

- abs_M satisfies abs_φ_{must}.
- abs_M satisfies abs_φ_{may} but does not satisfy abs_φ_{must}.
- abs_M does not satisfy abs_φ_{may}.

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 to 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 the abstract action formula abs_α. The semantics is given below:

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

We now give an example. Let us consider the system in Fig. 6. 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"*
  $s_0 \models \langle a(d_0)\rangle\mathsf{T}$. The abstract version of the formula is $\langle a(d)\rangle\mathsf{T}$, which trivially holds for S_0. Therefore, we can infer that there exists x such that $H(x) = d$ for which $\langle a(x)\rangle\mathsf{T}$ holds in s_0. In other words, $s_0 \models \langle a(d_0) \vee a(d_1)\rangle\mathsf{T}$ which implies that $s_0 \models \langle a(d_0)\rangle\mathsf{T}$ or $s_0 \models \langle a(d_1)\rangle\mathsf{T}$.
- *"It is not possible to do a transition $b(d_0)$ from the initial state"*
  $s_0 \models [\, b(d_0)\,]\,\mathsf{F}$. The abstract version of the formula is $[\, b(d)\,]\,\mathsf{F}$, which trivially holds for S_0. Therefore, we can infer that for all x such that $H(x) = d$ implies $[\, b(x)\,]\,\mathsf{F}$ holds in s_0. In other words, $s_0 \models [\, b(d_0) \vee b(d_1)\,]\,\mathsf{F}$, which implies that $s_0 \models [\, b(d_0)\,]\,\mathsf{F}$ and $s_0 \models [\, b(d_1)\,]\,\mathsf{F}$.

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. The output of the inference can be described by quantifiers over the actions using a logic such as the one presented in [38].

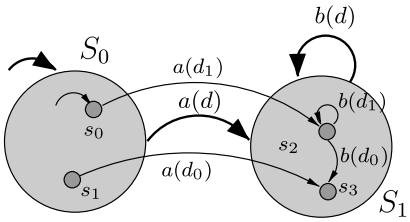


Fig. 6 Example of abstract model checking

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.

An important aspect of abstract model checking refers to spurious counter-examples. Some formulae are not satisfied due to non-realistic scenarios i.e. abstract traces that do not have any corresponding concrete one. In these cases, it is possible to improve the precision of the abstract model in two ways:

– By removing the spurious *may* traces. This gives a model with less possible behaviours.
– By adding extra *must* traces. This gives a model with more necessary behaviours.

Classical theories such as [27] eliminate possible behaviours using refinement by symbolic program execution. Another way to deal with spurious counter-example is to strengthen the formula to prove, in order to discriminate the non-realistic traces, this possibility is studied in [15]. None of these theories are implemented in our tool yet.

7 A case study: *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 confirmation messages may be lost. The sender will attempt to retransmit each packet at most *MAX* times.

The protocol presents 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. We describe, here, the application of the abstract interpretation techniques to remove uninteresting information of this protocol in order to use model checking to verify it.

We base our solution on the μCRL model presented in the paper [18], in which Groote and Van de Pol proved using algebraic methods that the model is branching bisimilar to the desired external behaviour also specified in μCRL. This proof requires a strong and creative human interaction in order to be accomplished.

The figure below shows the different agents that participate in the system. The system contains a sender that gets a file that consists of a list of elements. It delivers the file frame by frame through a channel. The receiver sends an acknowledgement for each frame, when it receives a packet it delivers it to the external receiver client attaching a positive indication I_{fst}, I_{inc} or I_{ok}. The sender, after each frame, waits for the acknowledgements, if the confirmation message does not arrive, it retransmits the packet. If the transmission was successful, i.e., all the acknowledgements have arrived, then the sender informs the sending

client with a positive indication. When the maximum number of retransmissions is exceeded, the transmission is cancelled and I_{nok} is sent to the exterior by both participants. If the last frame or its confirmation are lost the sender cannot know whether the receiver has received the complete list, therefore it sends *"I don't know"* to the sending client, I_{dk}.

sort abs_List
func $empty :\rightarrow abs\_List$
$one :\rightarrow abs\_List$
$more :\rightarrow abs\_List$
$H : List \rightarrow abs\_List$
$eq : abs\_List \times abs\_List \rightarrow Bool$
$abs\_head : abs\_List \rightarrow \mathcal{P}(abs\_D)$
$abs\_tail : abs\_List \rightarrow \mathcal{P}(abs\_List)$
$abs\_last : abs\_List \rightarrow \mathcal{P}(Bool)$
$abs\_indl : abs\_List \rightarrow \mathcal{P}(Bit)$
var $abs\_l :\rightarrow abs\_List$
$l : List$
rew $abs\_head(\widehat{l}) = \{d0, d1, d2\}$
$abs\_tail(empty) = \{empty\}$
$abs\_tail(one) = \{empty\}$
$abs\_tail(more) = \{more, one\}$
$abs\_last(empty) = \{\mathsf{T}\}$
$abs\_last(one) = \{\mathsf{T}\}$
$abs\_last(more) = \{\mathsf{F}\}$
$abs\_indl(empty) = \{e_1\}$
$abs\_indl(one) = \{e_1\}$
$abs\_indl(more) = \{e_0\}$
$eq(abs\_l, abs\_l) = \mathsf{T}$
$eq(empty, one) = \mathsf{F}$
$eq(..., ...) = ...$
$H(emptyList) = empty$
$H(cons(d, emptyList)) = one$
$H(cons(d, cons(d', l))) = more$

The protocol depends on the time behaviour, which is controlled by two timers T_1 and T_2. They determine when the messages are either delivered or lost, the retransmission of the packets and the timeout that makes the participants give up the transmission. The solution of [18] does not deal with the explicit time delays but with some non-deterministic signals (modelled by channels 9 and 10).

We are interested in proving that the external indications delivered by the sender and the receiver are "consistent". For that purpose, we chose an abstraction that abstracts away the data stored in the file to transmit and maps the list to three critical values: *empty, one, more*. *empty* for when the list is empty, *one* when it has only one element, and *more* when it has more than one. We provide a new data specification, which is shown in the μCRL code below.

The function *indl* gives different bits when either a list is at its end (it is empty or has only one element) or when there is more than one element, the rest of the functions are standard. The maximum number of retransmissions is abstracted away, which makes

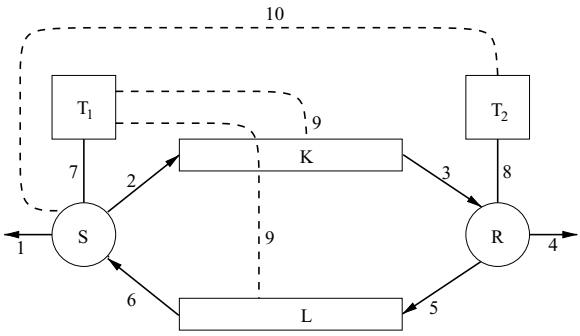


Fig. 7 Overview of the bounded retransmission protocol

the sender non-deterministically choose between resending a lost packet or giving up the transmission.

Once we obtain the abstract *Modal*-LPE, we can use the state space generator of the μCRL toolset to obtain the abstract *Modal*-LTS. The result consists of 446 states and 1016 transitions, from which 448 are *must* transitions and the rest *may*s.

The abstraction we have used allows to reason about the execution of the final part of the protocol without knowing the exact content of data files or the number of retrials. For example the following *safety* property: *"after a positive notification by the receiver, the sender cannot send a negative one"* is *necessarily* satisfied by the abstract system.

(C1): $[\ \mathsf{true}^*\ .\ \text{`}\mathsf{R}(.*,\mathsf{I}_{\mathsf{ok}})\text{'}\ .\ (\neg\ \text{`}\mathsf{S}(.*)\text{'})*\ .\ \text{`}\mathsf{S}(\mathsf{I}_{\mathsf{nok}})\text{'}\ ]\ \mathsf{F}$

The following *liveness* property expresses that: *"After a negative notification by the receiver, there exists a path that leads to a negative or* don't know *notification by the sender"*.

(C2): $[\ \mathsf{T}^*\ .\ \text{`}\mathsf{R}(.*,\mathsf{I}_{\mathsf{nok}})\text{'}\ ]\ \langle\ \mathsf{T}^*\ .\ (\text{`}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ \vee\ \text{`}\mathsf{S}(\mathsf{I}_{\mathsf{nok}})\text{'})\rangle\ \mathsf{T}$

The next property is stronger than the previous, instead of only requesting that there exists a path it states that the expected sender notification is inevitably achieved:

(C3): $[\ \mathsf{T}^*\ .\ \text{`}\mathsf{R}(.*,\mathsf{I}_{\mathsf{nok}})\text{'}\ ]\ \mu\ \mathrm{X}.\ (\langle\ \mathsf{T}\rangle\ \mathsf{T}\wedge\ [\ \neg(\text{`}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ \vee\ \text{`}\mathsf{S}(\mathsf{I}_{\mathsf{nok}})\text{'})\ ]\ \mathrm{X})$

These three properties are *necessarily* satisfied in the abstract system, therefore we can infer its satisfaction in the original one. However, the following property, which states that *"after a positive notification by the receiver there exists a path that leads to a* don't know *notification by the sender"* is not satisfied in the abstract system. The reason is that we have abstracted away the maximum number of retransmissions, therefore if all the acknowledgements are lost the sender can retransmit the frames forever:

(C4): $[\ \mathsf{T}^*\ .\ \text{`}\mathsf{R}(.*,\mathsf{I}_{\mathsf{ok}})\text{'}\ ]\ \langle\ \mathsf{T}^*\ .\ \text{`}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ \rangle\ \mathsf{T}$

C4 is not *necessarily* satisfied but is *possibly* satisfied on the abstract, therefore we cannot conclude anything on the concrete.

Other papers have verified different properties of the protocol using abstract interpretation, we refer among others to [12, 29]. The approach of Manna et al. is based on automatic

predicate abstractions and is limited to the proof of invariants. Dams and Gerth propose a number of creative abstractions in order to prove the satisfaction of safety properties on the sequentiality of the delivered frames.

8 Conclusion and related work

Automated applications are indispensable to apply formal methods to realistic industrial systems. Here we have described a toolkit that helps in using the abstraction techniques theoretically introduced in [34]. The tool described is not the only one dedicated to such tasks.

The existing tool closest to ours is αSpin [15], 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 [21] implements the same method of abstraction, furthermore it provides algorithms for *program slicing* and data dependency analysis in order to automatically find suitable variables to abstract. Bandera generates PROMELA code from simple Java programs.

FeaVer [23] and abC [13] 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 [22], BeBop [2] and SLAM [3] use *predicate abstraction*. We refer to [11] for an extended overview of tools and techniques for abstract model checking.

All the enumerated tools only generate over-approximations, therefore they 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, another feature that is not provided by any other tool is the possibility of abstracting action labels.

In [32], the tool was used to attempt to improve the performance of distributed algorithms for model checking and state space reduction. The idea is to introduce a new distribution policy of state spaces over workers. This policy reduces the number of transitions between states located at different workers. This in turn is expected to reduce the communication costs of the distributed algorithms. We have used the automatic abstraction mechanism of the tool to compute a small approximation of the state space, starting from some high level description of the system. Based on this approximation, the connectivity of concrete states is predicted. This information is used to distribute states with expected connectivity to the same worker.

The tool implements a simple automatic abstraction approach, as *variable hiding*, and facilitates the use of creative abstractions. More work is needed to automate the task of selecting suitable abstractions and of providing correct abstract domains. As presented at the end of the previous section, an interesting aspect that should be studied deeper is how to deal with spurious counter-examples.

References

1. Ball T, Majumdar R, Millstein T, Rajamani SK (2001) Automatic predicate abstraction of C programs. In: Proceedings of Conference on Programming Language Design and Implementation (PLDI), ACM, pp 203–213
2. Ball T, Rajamani SK (2000) BeBop: a symbolic model checker for Boolean programs. In: Proceedings of SPIN model checking and software verification, LNCS, vol 1885, Springer, pp 113–130

3. Ball T, Rajamani SK (2001) Automatically validating temporal safety properties of interfaces. In: Proceedings of SPIN model checking and software verification, LNCS, vol 2057. Springer, pp 103–122
4. Bergstra JA, Klop JW (1985) Algebra of communicating processes with abstraction. Theor Comput Sci 37:77–121
5. Blom S, Fokkink W, Groote JF, van Langevelde I, Lisser B, van de Pol JC (2001) μCRL: a toolset for analysing algebraic specifications. In: Proceedings of Computer Aided Verification (CAV), LNCS, vol 2102. Springer, pp 250–254
6. Blom S, Groote JF, van Langevelde I, Lisser B, van de Pol JC (2003) New developments around the μCRL tool set. ENTCS 80
7. Bruns G, Godefroid P (1999) Model checking partial state spaces with 3-valued temporal logics. In: Proceedings of Computer Aided Verification (CAV), LNCS, vol 1877. Springer, pp 274–287
8. Clarke EM, Grumberg O, Long DE (1992) Model checking and abstraction. J ACM, pp 343–354
9. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixed points. J ACM 238–252
10. Dams D (1996) Abstract interpretation and partition refinement for model checking. PhD thesis, Eindhoven University of Technology
11. Dams D (2002) Abstraction in software model checking: principles and practice (tutorial overview and bibliography). In: Proceedings of SPIN model checking and software verification, LNCS, vol 2318, Springer, pp 14–21
12. Dams D, Gerth R (2000) The bounded retransmission protocol revisited. ENTCS 9
13. Dams D, Hesse W, Holzmann G (2002) Abstracting C with abC. In: Proceedings of the Computer Aided Verification (CAV), LNCS, vol 2404, Springer, pp 515–520
14. Long DE (1993) Model checking, abstraction, and compositional verification. PhD thesis, Carnegie Mellon University
15. Gallardo MM, Martínez J, Merino P, Pimentel E (2004) αSPIN: a tool for abstract model checking. Int J Softw Tools for Technol Transf (STTT) 5(2–3):165–184
16. Garavel H, Lang F, Mateescu R (2002) An overview of CADP 2001. Eur Assoc Softw Sci Technol Newsl 4:13–24
17. Godefroid P, Huth M, Jagadeesan R (2001) Abstraction-based model checking using modal transition systems. In: Proceedings of the concurrency theory (CONCUR), LNCS, vol 2154, Springer, pp 426–440
18. Groote JF, van de Pol JC (1996) A bounded retransmission protocol for large data packets. In: Proceedings of the Algebraic Methodology and Software Technology (AMAST), LNCS, vol 1101. Springer, pp 536–550
19. Groote JF, Ponse A (1994) The syntax and semantics of μCRL. In: Algebra of communicating processes, workshops in computing. pp 26–62
20. Groote JF, Ponse A, Usenko Y (2001) Linearization in parallel pCRL. J Logic Algebraic Programm 48(1–2):39–70
21. Hatcliff J, Dwyer M, Pasareanu C, Robby (2002) Foundations of the Bandera abstraction tools. In: Proceedings of the essence of computation, LNCS, vol 2566, Springer, pp 172–203
22. Havelund K, Skakkebaek J (1999) Applying model checking in Java verification. In: Proceedings of the SPIN model checking and software verification, LNCS, vol 1680, Springer, pp 216–232
23. Holzmann GJ, Smith MH (1999) A practical method for verifying event-driven software. In: Proceedings of International Conference on Software Engineering (ICSE). ACM, pp 597–607
24. Huth M, Jagadeesan R, Schmidt D (2001) Modal transition systems: a foundation for three-valued program analysis. In: Proceedings of the programming languages and systems (ESOP), LNCS, vol 2028, Springer, pp 155–169
25. Jones ND, Nielson F (1995) Abstract interpretation: a semantics-based tool for program analysis. In: Handbook of logic in computer science. Oxford Science Publications, pp 527–636
26. Kozen D (1982) Results on the propositional μ-calculus. In: Proceedings of the International Conference on Automata, Languages and Programming (ICALP), LNCS, vol 140. Springer, pp 348–359
27. Kroening D, Groce A, Clarke EM (2004) Counterexample guided abstraction refinement via program execution. In: Proceedings of the international conference on formal engineering methods (ICFEM), LNCS, vol 3380. Springer, pp 224–238
28. Larsen KG, Thomsen B (1988) A modal process logic. In: Proceedings of the logic in computer science (LICS). IEEE, pp 203–210
29. Manna Z, Colon M, Finkbeiner B, Sipma H, Uribe TE (1997) Abstraction and modular verification of infinite-state reactive systems. In: Proceedings of the requirements targeting software and systems engineering (RTSE), LNCS, vol 1526. Springer, pp 273–292
30. Mateescu R (1998) Verification des proprietes temporelles des programmes paralleles. PhD thesis, Institut National Polytechnique de Grenoble

31. Ore O (1944) Galois connexions. Trans. Am Math Soc 55:493–513
32. Orzan S, van de Pol JC, Valero Espada M (2005) A state space distribution policy based on abstract interpretation. ENTCS 128:35–45
33. van de Pol JC (2001) A prover for the μCRL toolset with applications. Technical Report SEN-R0106, CWI
34. van de Pol JC, Valero Espada M (2004) Modal abstraction in μCRL. In: Proceedings of the Algebraic Methodology and Software Technology (AMAST), LNCS, vol 3116, Springer, pp 409–425
35. Schmidt D (2002) Structure-preserving binary relations for program abstraction. In: Proceedings of the essence of computation, LNCS, vol 2566, Springer, pp 245–268
36. Usenko Y (2002) Linearization in μCRL. PhD thesis, Eindhoven University of Technology
37. Valero M (2005) Modal abstraction and replication of processes with data. PhD thesis, Free University Amsterdam
38. Willemse T (2003) Semantics and verification in process algebras with data and timing. PhD thesis, Eindhoven University of Technology