Specification and Safety Verification of Parametric Hierarchical Distributed Systems

Marius Bozga, Radu Iosif

Specification and Safety Verification of Parametric Hierarchical Distributed Systems

Marius Bozga[0000−0003−4412−5684] and Radu Iosif[0000−0003−3204−3294]

Univ. Grenoble Alpes, CNRS, Grenoble INP**, VERIMAG, 38000 Grenoble, France
{marius.bozga,radu.iosif}@univ-grenoble-alpes.fr

Abstract. We introduce a term algebra as a new formal specification language for the coordinating architectures of distributed systems consisting of a finite yet unbounded number of components. The language allows to describe infinite sets of systems whose coordination between components share the same pattern, using inductive definitions similar to the ones used to describe algebraic data types or recursive data structures. Further, we give a verification method for the parametric systems described in this language, relying on the automatic synthesis of structural invariants that enable proving general safety properties (absence of deadlocks and critical section violations). The invariants are defined using the $\mathsf{WS}\kappa\mathsf{S}$ fragment of the monadic second order logic, known to be decidable by a classical automata-logic connection. This reduces the safety verification problem to checking satisfiability of a $\mathsf{WS}\kappa\mathsf{S}$ formula. We implemented the invariant synthesis method into a prototype tool and carried out verification experiments on a number of non-trivial models specified using the term algebra.

1 Introduction

The separation between behavior and coordination is a fundamental principle in the design of large-scale distributed systems [16]. By *behavior* we mean a set of traces of observable events. A *component* is a representation of a behavior, by means of a (finite) state machine, whose actions are labeled by events. The *architecture* of the system defines the interactions between components, as sets of events that must occur simultaneously in several components. For instance, Fig. 1a shows a token-ring systems, whose components are depicted in yellow boxes (behaviors are modeled by the finite-state machines within the boxes) and whose architecture is the set of connections between components (depicted with solid lines). Such high-level models of real-life distributed systems are suitable for reasoning about correctness in the early stages of system design, when details related to network reliability or the implementation of coordination mechanisms, by means of low-level synchronization mechanisms (e.g. semaphores, monitors, compare-and-swap, etc.) are abstracted away.

** Institute of Engineering Univ. Grenoble Alpes

This modular view of a distributed system is key to scalable design methods that exploit a conceptual hierarchy, in which each module is split into sub-modules. For instance, a ring is a chain whose final output port is connected to the initial input port, whereas a chain consists of a (head) component linked to a separate (tail) chain (Fig. 1b). Furthermore, system designers are accustomed to the use of predefined *architectural patterns*, that define the interactions between (unboundedly large) sets of modules (e.g. crowds, rings, pipelines, stars, trees, etc.). In this context, the contribution of the paper is three-fold.

1. We introduce a formal language to describe the coordinating architectures of distributed systems parameterized by (i) the number of components of each type that are active in the system, e.g. a system with n readers and m writers, in which n and m are not known a priori and (ii) the pattern in which the interactions occur (e.g. a pipeline, ring, star or more general hypergraph structures). The language uses predicate symbols to hierarchically break the architecture into sub-modules. The interpretation of these predicate symbols is defined inductively by rewriting rules consisting of terms that contain predicate atoms, in a way that recalls the usual definitions of algebraic datatypes [2] or heaps [18].

2. We tackle the *parametric safety problem* for systems described in this language, which is checking that the reachable states of every jnstance stays clear of a set of global error configurations, such as deadlocks or critical section violations. We synthesize invariants directly from the syntactic description of the system, generate $\mathsf{WS}\kappa\mathsf{S}$ formulæ [19] that are unsatisfiable only if every system described by the given inductive definitions is safe and use off-the-shelf $\mathsf{WS}\kappa\mathsf{S}$ solvers [11] for proving safety automatically. The invariant synthesis method models the set of executions of a parametric system as a boolean (1-safe) Petri net of unbounded size and computes structural invariants (trap invariants, linear invariants) of this Petri net.

3. We implemented the invariant synthesis in a prototype tool and experimented with a number of parametric component-based systems with non-trivial architectural patterns, such as trees with root links, trees with linked leaves, token-rings with or without a main controller, etc.

Example 1. Let us consider a distributed system consising of components of type C, having two interaction ports, namely *in* and *out* and whose behavior is described by a finite state machine with transitions $q_0 \xrightarrow{out} q_1$ and $q_1 \xrightarrow{in} q_0$. These components are arranged in a ring, such that the *out* port of a component is connected to the *in* port of its right neighbour, with the exception of the last component, whose *out* port connects to the *in* port of the first component (Fig. 1a). The connections (interactions) in the system are described by the predicate $Ring()$, defined inductively by the rules below:

$$Ring() \leftarrow \nu y_1\ \nu y_2\ .\ \langle out(y_2) \cdot in(y_1) \rangle (Chain(y_1, y_2)) \tag{1}$$

$$Chain(x_1, x_2) \leftarrow \langle out(x_1) \cdot in(x_2) \rangle (Comp(x_1), Comp(x_2)) \tag{2}$$

$$Chain(x_1, x_2) \leftarrow \nu y_1\ .\ \langle out(x_1) \cdot in(y_1) \rangle (Comp(x_1), Chain(y_1, x_2)) \tag{3}$$

$$Comp(x) \leftarrow \mathsf{C}(x) \tag{4}$$

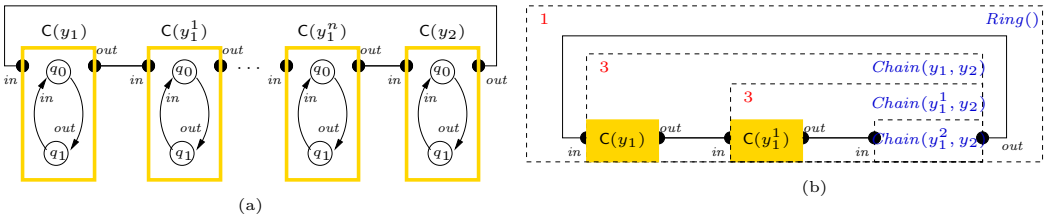

Fig. 1. Recursive Specification of a Token-Ring System

Rule (1) states that a $\mathit{Ring}()$ consist of a $\mathit{Chain}(y_1, y_2)$, where y_1 and y_2 are the indices of the first and last components[1], respectively. The last *out* port is connected to the first *in* port, written as $\mathit{out}(y_2) \cdot \mathit{in}(y_1)$. Rule (2) states that the least $\mathit{Chain}(x_1, x_2)$ consists of two instances of type C, namely $\mathsf{C}(x_1)$ and $\mathsf{C}(x_2)$, and the *out* port of x_1 connects to the *in* port of x_2, described as $\mathit{out}(x_1) \cdot \mathit{in}(x_2)$. Rule (3) gives the inductive step, namely that every $\mathit{Chain}(x_1, x_2)$ consists of a component $\mathsf{C}(x_1)$ that interacts with a disjoint chain from y_1 to x_2. Here the binder νy_1 makes sure the value of y_1 is different from the value of every other variable in the system. Since this binder is used in a recursive rule, each identifier in a subsequent unfolding of $\mathit{Chain}(y_1, x_2)$ is guaranteed to be unique. Last, rule (4) is used to instantiate (i.e. create new) components of type C. In principle, this rule is not necessary, as any occurrence of a predicate $\mathit{Comp}(y)$ can be replaced by a component $\mathsf{C}(y)$, however it is considered for technical reasons related to our invariant-based verification approach.

Fig. 1b shows the unfoldings of this set of recursive definitions. The system depicted in Fig. 1a is obtained by an application of rule (1), followed by n applications of rule (3), ending with an application of rule (2). The first two applications of (3) following the application of (1) are depicted in Fig. 1b, with rule labels annotated (each application of rule (3) creates a fresh variable, denoted by $y_1^1, \ldots, y_1^n$, respectively). ■

2 Preliminaries

This section introduces preliminary definitions used throughout the paper. Given sets A and B, we denote by $A \to B$ the set of total functions $f : A \to B$. If $\mathbf{a} = \langle a_1, \ldots, a_n \rangle$ is a tuple of values from A, then $f(\mathbf{a}) \stackrel{\text{def}}{=} \langle f(a_1), \ldots, f(a_n) \rangle$. By $\mathbf{a} \cdot \mathbf{b}$ we denote the concatenation of tuples $\mathbf{a}$ and $\mathbf{b}$. For two positive integers $k, \ell \in \mathbb{N}$, we denote by $[k, \ell]$ the set $\{k, k+1, \ldots, \ell\}$, assumed to be empty if $k > \ell$. The cardinality of a finite set A is denoted by $|A|$.

Trees Trees play a key role in the definition of parametric distributed systems from the following section (§3). Let $\kappa \geq 1$ be an integer constant, fixed throughout this paper, and let $[1, \kappa]^*$ denote the set of finite sequences of integers between

[1] First and last are understood here in the order of unfolding of the rewriting rules.

1 and κ, called *nodes* in the following. A *κ-ary tree* $\mathcal{T}$ is a partial function mapping $[1,\kappa]^*$ to a set of labels. The domain of $\mathcal{T}$, denoted $\mathrm{nodes}(\mathcal{T})$, is such that $wi \in \mathrm{nodes}(\mathcal{T})$ for some $i \in [1,\kappa]$ only if $w \in \mathrm{nodes}(\mathcal{T})$ and $wj \in \mathrm{nodes}(\mathcal{T})$ for all $j \in [1, i-1]$. The *root* of $\mathcal{T}$ is the empty sequence ϵ, the *children* of a node $w \in \mathrm{nodes}(\mathcal{T})$ are $\{wi \in \mathrm{nodes}(\mathcal{T}) \mid i \in [1,\kappa]\}$ and the *parent* of a node wi, $i \in [1,\kappa]$, is w. The *leaves* of $\mathcal{T}$ are $\mathrm{leaves}(\mathcal{T}) \stackrel{\mathsf{def}}{=} \{w \in \mathrm{nodes}(\mathcal{T}) \mid w.1 \notin \mathrm{nodes}(\mathcal{T})\}$. The *subtree* of $\mathcal{T}$ rooted at w is defined by $\mathrm{nodes}(\mathcal{T}{\downarrow_w}) \stackrel{\mathsf{def}}{=} \{w' \mid ww' \in \mathrm{nodes}(\mathcal{T})\}$ and $\mathcal{T}{\downarrow_w}\,(w') \stackrel{\mathsf{def}}{=} \mathcal{T}(ww')$, for all $w' \in \mathrm{nodes}(\mathcal{T}{\downarrow_w})$.

The invariant synthesis method uses the restriction of monadic second order logic to trees of branching degree κ and quantification over finite sets only. Let $\mathbb{V}_1 = \{x, y, z, \ldots\}$ and $\mathbb{V}_2 = \{X, Y, Z, \ldots\}$ be countably infinite sets of first and second order variables, respectively. The formulæ of the $\mathsf{WS}\kappa\mathsf{S}$ logic are defined inductively by the syntax:

$$\begin{array}{ll} \tau ::= \epsilon \mid x \in \mathbb{V}_1 \mid \tau_1.i,\ i \in [1,\kappa] & \text{terms} \\ \phi ::= \tau = \tau \mid X(\tau) \mid \phi \wedge \phi \mid \neg\phi \mid \exists x\ .\ \phi \mid \exists X\ .\ \phi & \text{formulæ} \end{array}$$

As usual, we write $\phi_1 \vee \phi_2 \stackrel{\mathsf{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2 \stackrel{\mathsf{def}}{=} \neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2 \stackrel{\mathsf{def}}{=} (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, $\forall x\ .\ \phi \stackrel{\mathsf{def}}{=} \neg\exists x\ .\ \neg\phi$ and $\forall X\ .\ \phi \stackrel{\mathsf{def}}{=} \neg\exists X\ .\ \neg\phi$. $\mathsf{WS}\kappa\mathsf{S}$ formulæ are interpreted over an infinite tree, where first order variables $x \in \mathbb{V}_1$ range over individual nodes $n \in [1,\kappa]^*$, second order variables $X \in \mathbb{V}_2$ range over *finite sets* of nodes, ϵ is a constant symbol interpreted as the root of the tree and, for all $i \in [1,\kappa]$, the notation $.i$ is is interpreted as the function mapping each $w \in [1,\kappa]^*$ into wi. Given a valuation $\nu : \mathbb{V}_1 \cup \mathbb{V}_2 \rightarrow [1,\kappa]^* \cup 2^{[1,\kappa]^*}$, such that $\nu(x) \in [1,\kappa]^*$, for each $x \in \mathbb{V}_1$ and $\nu(X) \subseteq [1,\kappa]^*$ ($\nu(X)$ is finite), for each $X \in \mathbb{V}_2$, the satisfaction relation $\models$ is defined inductively, as usual [14]. A valuation ν is a *model* of a formula ϕ iff $\nu \models \phi$. A formula is *satisfiable* if and only if it has a model.

Petri nets A *Petri net* (PN) is a tuple $\mathsf{N} = \langle S, T, E\rangle$, where S is a set of *places*, T is a set of *transitions*, $S \cap T = \emptyset$, and $E \subseteq (S \times T) \cup (T \times S)$ is a set of *edges*. Given $x, y \in S \cup T$, we write $E(x,y) \stackrel{\mathsf{def}}{=} 1$ if $(x,y) \in E$ and $E(x,y) \stackrel{\mathsf{def}}{=} 0$, otherwise. Let ${}^\bullet x \stackrel{\mathsf{def}}{=} \{y \in S \cup T \mid E(y,x) = 1\}$, $x^\bullet \stackrel{\mathsf{def}}{=} \{y \in S \cup T \mid E(x,y) = 1\}$ and lift these definitions to sets of nodes. A *marking* of N is a function $\mathrm{m} : S \rightarrow \mathbb{N}$. A transition t is *enabled* in m if and only if $\mathrm{m}(s) > 0$ for each place $s \in {}^\bullet t$. We write $\mathrm{m} \xrightarrow{t} \mathrm{m}'$ whenever t is enabled in m and $\mathrm{m}'(s) = \mathrm{m}(s) - E(s,t) + E(t,s)$, for all $s \in S$ and $t \in T$. A sequence of transitions $\sigma = t_1, \ldots, t_n$ is a *firing sequence*, written $\mathrm{m} \xrightarrow{\sigma} \mathrm{m}'$ if and only if either (i) $n = 0$ and $\mathrm{m} = \mathrm{m}'$, or (ii) $n \geq 1$ and there exist markings $\mathrm{m}_1, \ldots, \mathrm{m}_{n-1}$ such that $\mathrm{m} \xrightarrow{t_1} \mathrm{m}_1 \ldots \mathrm{m}_{n-1} \xrightarrow{t_n} \mathrm{m}'$.

A *marked Petri net* is a pair $\mathcal{N} = (\mathsf{N}, \mathrm{m}_0)$, where m_0 is the *initial marking* of N. A marking m is *reachable* in $\mathcal{N}$ if there exists a firing sequence σ such that $\mathrm{m}_0 \xrightarrow{\sigma} \mathrm{m}$. We denote by $\mathcal{R}(\mathcal{N})$ the set of reachable markings of $\mathcal{N}$. A marked PN $\mathcal{N}$ is *boolean* if $\mathrm{m}(s) \leq 1$, for each $s \in S$ and $\mathrm{m} \in \mathcal{R}(\mathcal{N})$. All marked PNs considered in the following will be boolean and we shall silently blur the distinction between a marking $\mathrm{m} : S \rightarrow \{0,1\}$ and the set $\{s \in S \mid \mathrm{m}(s) = 1\}$.

Given a set of markings $\mathcal{E}$, a marked PN $\mathcal{N}$ is *safe w.r.t.* $\mathcal{E}$ if and only if $\mathcal{R}(\mathcal{N}) \cap \mathcal{E} = \emptyset$. A set of markings $\mathcal{M}$ is an *inductive invariant* of $\mathcal{N} = (\mathsf{N}, \mathrm{m}_0)$ if

and only if $\mathrm{m}_0 \in \mathcal{M}$ and for each $\mathrm{m} \xrightarrow{t} \mathrm{m}'$ such that $\mathrm{m} \in \mathcal{M}$, we have $\mathrm{m}' \in \mathcal{M}$. It is known that $\mathcal{R}(\mathcal{N})$ is the least inductive invariant of $\mathcal{N}$, thus $\mathcal{N}$ is safe w.r.t $\mathcal{E}$ if it has an inductive invariant $\mathcal{M}$ disjoint from $\mathcal{E}$.

Components In this paper we are concerned with systems consisting of an unbounded number of components that are replicas of a fairly small set of patterns, called component types. Let $\mathbb{P} = \{a, b, \ldots\}$ and $\mathbb{S} = \{s, t, \ldots\}$ be countably infinite sets of *ports* and *states*, respectively. An injective function P (resp. S) mapping tree nodes to ports (resp. states) is called a *port type* (resp. *state type*). A *component type* is a tuple $\mathcal{B} = \langle \mathcal{P}, \mathcal{S}, \mathcal{I}, \Delta \rangle$, where $\mathcal{P} \subseteq [1, \kappa]^* \rightarrow \mathbb{P}$ and $\mathcal{S} \subseteq [1, \kappa]^* \rightarrow \mathbb{S}$ are finite sets of port and state types, $\mathcal{I} \in \mathcal{S}$ is the *initial* state type, and Δ is a finite set of *transition rules* $S \xrightarrow{P} T$, where $S, T \in \mathcal{S}$ and $P \in \mathcal{P}$. In addition, we require that (i) the elements of $\mathcal{P}$ (resp. $\mathcal{S}$) have pairwise disjoint ranges and (ii) for any two transition rules $S_1 \xrightarrow{P_1} S_1', S_2 \xrightarrow{P_2} S_2'$, if $P_1 = P_2$ then $S_1 = S_2$ and $S_1' = S_2'$. For a transition rule $S \xrightarrow{P} S' \in \Delta$, let ${}^\bullet P \stackrel{\text{def}}{=} S$ and $P^\bullet \stackrel{\text{def}}{=} S'$ denote the pre- and post-state type of the unique transition rule whose label is the port type P.

The replicas of a component type are indexed (distinguished) by tree nodes[2]. Given a component type $\mathcal{B} = \langle \mathcal{P}, \mathcal{S}, \mathcal{I}, \Delta \rangle$ and a tree node $w \in [1, \kappa]^*$, we define the *component* $\mathcal{B}(w) \stackrel{\text{def}}{=} \langle \{P(w) \mid P \in \mathcal{P}\}, \{S(w) \mid S \in \mathcal{S}\}, \mathcal{I}(w), \{S(w) \xrightarrow{P(w)} S'(w) \mid S \xrightarrow{P} S' \in \Delta\} \rangle$. Note that the sets of ports $\{P(w) \mid P \in \mathcal{P}\}$ (resp. states $\{S(w) \mid S \in \mathcal{S}\}$) of different replicas of the same component type are disjoint, because the port (state) types are required to have disjoint ranges. We slightly abuse notation by writing ${}^\bullet(P(w)) \stackrel{\text{def}}{=} {}^\bullet(P)(w)$ and $(P(w))^\bullet \stackrel{\text{def}}{=} (P)^\bullet(w)$ (we omit brackets when they are clear from the context). We consider below a set $\mathbb{B}$ of component types, with pairwise disjoint sets of port and state types.

Architectures The coordination in a system is defined by architectures. An *interaction* $\pi \in 2^{\mathbb{P}}$ is a finite set of ports. An *architecture* $\gamma \subseteq 2^{\mathbb{P}}$ is a finite set of interactions. Given component types $\mathcal{B}_i = \langle \mathcal{P}_i, \mathcal{S}_i, \mathcal{I}_i, \Delta_i \rangle \in \mathbb{B}$ and tree nodes $w_i \in [1, \kappa]^*$, the *behavior* of the system consisting of the components $\mathcal{B}_i(w_i)$, $i = 1, \ldots, n$, coordinated by the architecture γ is defined by the marked PN $\gamma(\mathcal{B}_1(w_1), \ldots, \mathcal{B}_n(w_n)) \stackrel{\text{def}}{=} (\langle S, \gamma, E \rangle, \mathrm{m}_0)$, where $S \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \{S(w_i) \mid S \in \mathcal{S}_i\}$ is the set of places, for each interaction $\pi \in \gamma$, the edges to (from) π are given by ${}^\bullet\pi \stackrel{\text{def}}{=} \{{}^\bullet p \mid p \in \pi\}$ ($\pi^\bullet \stackrel{\text{def}}{=} \{p^\bullet \mid p \in \pi\}$) and the initial marking is $\mathrm{m}_0 \stackrel{\text{def}}{=} \{\mathcal{I}_i(w_i) \mid i \in [1, n]\}$.

Example 2. Fig. 2 shows the marked PN that defines the behavior of the system from Fig. 1a. The tree node $1 \ldots 1$ (i times) is represented by its value i in the unary encoding. The interaction $\{in(n), out(1)\}$ is duplicated, for readability. The initially marked places are surrounded by dashed circles. ■

[2] We identify components by tree nodes in preparation of the ground for the verification technique from §4. However, these definitions can be given in general, for any countably infinite set of identifiers.

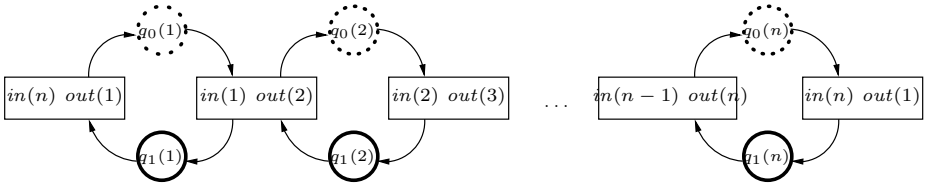

Fig. 2. The Behavior of a Token-Ring System

3 A Term Algebra of Behaviors

In this section we introduce a term algebra for describing the systems resulting from the application of an architecture to an unbounded number of component type instances (see Example 1 for the specification of a token-ring system in this language). Let $\mathbb{A}$ be a countably infinite set of *predicate symbols*, and let $\#(\mathsf{A})$ denote the arity of the predicate symbol $\mathsf{A} \in \mathbb{A}$.

Syntax The following syntax generates *behavioral terms* inductively:

$P \in [1,\kappa]^* \mapsto \mathbb{P},\ x \in \mathbb{V}_1,\ \mathcal{B} \in \mathbb{B},\ \mathsf{A} \in \mathbb{A}$

$I ::= P(x) \mid I_1 \cdot I_2$ interactions

$\Gamma ::= I \mid \Gamma_1 + \Gamma_2$ architectures

$\mathsf{b} ::= \mathcal{B}(x) \mid \langle\Gamma\rangle(\mathsf{b}_1,\ldots,\mathsf{b}_n) \mid \nu x\ .\ \mathsf{b}_1 \mid \mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})})$ behavioral terms

A variable x occurring in a behavioral term b is *free* if it does not occur in the scope of some subterm of the form $\nu x\ .\ \mathsf{b}_1$ and *bound* otherwise. The set of free variables occurring in a term b is denoted by $\mathrm{fv}(\mathsf{b})$. In the following, we assume that all bound variables occurring in a term are pairwise distinct and distinct from the free variables. This assumption loses no generality because terms obtained by α-conversion (renaming of bound variables) are usually viewed as the same term. A term b is *closed* if $\mathrm{fv}(\mathsf{b}) = \emptyset$ and *predicateless* if no predicates from $\mathbb{A}$ occur in b. We denote by $\mathsf{b}[y_1/x_1,\ldots,y_n/x_n]$ the term obtained by substituting the variable $x_i \in \mathrm{fv}(\mathsf{b})$ with y_i, for each $i \in [1,n]$. We write $\mathrm{size}(\mathsf{b})$ for the number of occurrences of symbols in b.

A term $\mathcal{B}(x)$ is called an *instance atom* and a term $\mathsf{A}(x_1,\ldots,x_n)$ is called a *predicate atom*. We denote by $\#_{\mathsf{pred}}(\mathsf{b})$ the number of occurrences of predicate atoms and by $\mathsf{pred}_i(\mathsf{b})$ the predicate atom that occurs i-th in b, for $i \in [1,\#_{\mathsf{pred}}(\mathsf{b})]$, in some linear order of the nodes in the syntax tree of b. The predicate symbols are interpreted as the least sets of predicateless terms inductively defined by a rewriting system:

Definition 1. *A* rewriting system *is a finite set $\mathcal{R}$ of rules of one of the forms:*

$$\mathsf{A}(x) \leftarrow \mathcal{B}(x)$$

$$\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \nu y_1 \ldots \nu y_n\ .\ \langle\Gamma\rangle(\mathsf{A}_1(z_1^1,\ldots,z_{\#(\mathsf{A}_1)}^1),\ldots,\mathsf{A}_m(z_1^m,\ldots,z_{\#(\mathsf{A}_m)}^m))$$

where $m \geq 1$ and $\left\{\{z_1^i,\ldots,z_{\#(\mathsf{A}_i)}^i\}\right\}_{i=1}^m$ is a partition of $\{x_1,\ldots,x_{\#(\mathsf{A})},y_1,\ldots,y_n\}$

For instance, in Example 1, rule (4) is an instantiation rule, whereas (1), (2) and (3) are inductive rules. We write $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow_{\mathcal{R}} \mathsf{b}$ for $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow$

$\mathsf{b} \in \mathcal{R}$. The *size* of $\mathcal{R}$ is $\mathrm{size}(\mathcal{R}) \stackrel{\mathsf{def}}{=} \sum_{\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow_{\mathcal{R}} \mathsf{b}} \mathrm{size}(\mathsf{b})$. Given behavioral terms b_1 and b_2, we denote by $\mathsf{b}_1 \stackrel{r}{\Leftarrow} \mathsf{b}_2$ the rewriting step that obtains b_2 by replacing a predicate atom $\mathsf{A}(y_1, \ldots, y_{\#(\mathsf{A})})$ in b_1 with $\mathsf{b}[y_1/x_1, \ldots, y_{\#(\mathsf{A})}/x_{\#(\mathsf{A})}]$, where $\mathsf{r} = (\mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \mathsf{b})$ is a rule of $\mathcal{R}$ and all bound variables in b are renamed to avoid clashes with the variables from b_1. We write $[\mathsf{b}]_{\mathcal{R}}$ for the set of predicateless terms obtained from b by exhaustively applying the rewriting rules from $\mathcal{R}$ to it.

Semantics Let us consider a given closed behavioral term b and a rewriting system $\mathcal{R}$. First, we define the semantics of a (closed) predicateless behavioral term $\mathsf{t} \in [\mathsf{b}]_{\mathcal{R}}$, as the behavior (i.e. marked PN) resulting from joining the components defined by the instance atoms from t, via the architecture consisting of all the interactions that occur in t. This definition is done in two steps:

(a) we write t in *prenex form* as $\nu x_1 \ldots \nu x_n \,.\, \mathsf{u}$, where u contains no more terms of the form $\nu x \,.\, \mathsf{b}$, by moving all the ν binders upfront. Because all bound variables in t, including those introduced by rewriting, are assumed to be pairwise distinct, this step incurs no name clashes.

(b) we apply the following *flattening* relation exhaustively:

$$\langle \Gamma_1 \rangle(\langle \Gamma_2 \rangle(\mathsf{b}_1, \ldots \mathsf{b}_i), \mathsf{b}_{i+1}, \ldots \mathsf{b}_n) \rightsquigarrow \langle \Gamma_1 + \Gamma_2 \rangle(\mathsf{b}_1, \ldots, \mathsf{b}_n) \tag{5}$$

Example 3. Consider the below rewriting sequence, using rules from Example 1:

$$\begin{array}{l} \mathit{Ring}() \Leftarrow \nu y_1 \nu y_2 \,.\, \langle \mathit{out}(y_2) \cdot \mathit{in}(y_1) \rangle(\mathit{Chain}(y_1, y_2)) \\ \Leftarrow \nu y_1 \nu y_2 \,.\, \langle \mathit{out}(y_2) \cdot \mathit{in}(y_1) \rangle(\nu y_1^1 \,.\, \langle \mathit{out}(y_1) \cdot \mathit{in}(y_1^1) \rangle(\mathit{Comp}(y_1), \mathit{Chain}(y_1^1, y_2))) \\ \Leftarrow \nu y_1 \nu y_2 \,.\, \langle \mathit{out}(y_2) \cdot \mathit{in}(y_1) \rangle(\nu y_1^1 \,.\, \langle \mathit{out}(y_1) \cdot \mathit{in}(y_1^1) \rangle(\mathit{Comp}(y_1), \\ \qquad\qquad \langle \mathit{out}(y_1^1) \cdot \mathit{in}(y_2) \rangle(\mathit{Comp}(y_1^1), \mathit{Comp}(y_2)))) \\ \Leftarrow \ldots \Leftarrow \nu y_1 \nu y_2 \,.\, \langle \mathit{out}(y_2) \cdot \mathit{in}(y_1) \rangle(\nu y_1^1 \,.\, \langle \mathit{out}(y_1) \cdot \mathit{in}(y_1^1) \rangle(\mathsf{C}(y_1), \\ \qquad\qquad \langle \mathit{out}(y_1^1) \cdot \mathit{in}(y_2) \rangle(\mathsf{C}(y_1^1), \mathsf{C}(y_2)))) = \mathsf{t} \end{array}$$

By applying flattening to the last term, we obtain:

$$\mathsf{t}^{\rightsquigarrow} = \nu y_1 \nu y_2 \,.\, \langle \mathit{out}(y_2) \cdot \mathit{in}(y_1) + \mathit{out}(y_1) \cdot \mathit{in}(y_1^1) + \mathit{out}(y_1^1) \cdot \mathit{in}(y_2) \rangle(\mathsf{C}(y_1), \mathsf{C}(y_1^1), \mathsf{C}(y_2)) \quad \blacksquare$$

Note that every chain $\mathsf{t}_1 \rightsquigarrow \mathsf{t}_2 \rightsquigarrow \ldots$ is finite, because the height of terms strictly decreases with flattening. The result of flattening is of the form $\mathsf{t}^{\rightsquigarrow} \stackrel{\mathsf{def}}{=} \langle \Gamma \rangle(\mathcal{B}_1(x_1), \ldots, \mathcal{B}_n(x_n))$, where $\Gamma = \sum_{k=1}^{m} P_{k1}(x_{k1}) \cdot \ldots \cdot P_{kr_k}(x_{kr_k})$ is an architecture description, such that each $P_{k\ell} \in [1, \kappa]^* \mapsto \mathbb{P}$ is a port type, $x_{k\ell} \in \{x_1, \ldots, x_n\}$, for all $k \in [1, m]$ and $\ell \in [1, r_k]$. Given an injective valuation $\nu : \mathbb{V}_1 \to [1, \kappa]^*$ that maps variables into distinct nodes of a tree of branching degree κ, we define the architecture $\Gamma(\nu) \stackrel{\mathsf{def}}{=} \{\{P_{k1}(\nu(x_{k1})), \ldots, P_{kr_k}(\nu(x_{kr_k}))\} \mid k \in [1, m]\}$ and the behavior:

$$\mathsf{B}^{\mathsf{t}}_{\nu} \stackrel{\mathsf{def}}{=} \Gamma(\nu)(\mathcal{B}_1(\nu(x_1)), \ldots, \mathcal{B}_n(\nu(x_n))) \tag{6}$$

The semantics of the behavioral term b in the rewriting system $\mathcal{R}$ is the following set of marked PNs:

$$[\![\mathsf{b}]\!]_{\mathcal{R}} \stackrel{\mathsf{def}}{=} \{\mathsf{B}^{\mathsf{t}}_{\nu} \mid \mathsf{t} \in [\mathsf{b}]_{\mathcal{R}} \,,\ \nu \in \mathbb{V}_1 \to [1, \kappa]^* \text{ injective}\} \tag{7}$$

As a remark, the flattening step is required because applying an architecture to a set of components is a global operation; if an interaction $P_{k1}(x_{k1}) \cdot \ldots \cdot P_{kr_k}(x_{kr_k})$ occurs as a monomial in the architecture description Γ of a subterm

$\mathsf{u} = \langle \Gamma \rangle(\mathsf{t}_1, \ldots, \mathsf{t}_\ell)$ of t and some variable x_{ki} occurs in an instance atom $\mathcal{B}(x_{ki})$ in t but not in u, the interaction would be ignored if we applied $\Gamma(\nu)$ directly to $\mathsf{B}^{\mathsf{t}_1}_\nu, \ldots, \mathsf{B}^{\mathsf{t}_\ell}_\nu$, for some injective valuation ν.

4 The Parametric Safety Problem

Having defined a rewriting-based term algebra for the specification of distributed systems, we move on to the problem of verifying that every behavior generated by a given rewriting system $\mathcal{R}$, starting from a given behavioral term b is safe with respect to a given set of error markings. This problem is challenging, because we ask for a proof of safety that holds for the behavior(s) of *every* predicateless rewriting of the behavioral term, i.e. for each $\mathsf{t} \in [\mathsf{b}]_{\mathcal{R}}$. Since, even for token-ring systems with finite-state components, the parametric safety problem is undecidable [9], we resort to a sound but necessarily incomplete solution, that consists in computing inductive invariants.

Structural invariants In contrast with the classical approach to invariant synthesis, based on a fixpoint iteration in an abstract domain [8], we consider a particular class of invariants, that can be obtained directly from the syntactic structure of the marked PN representation of behaviors. For this reason, we call these invariants *structural*. In the following, we define two kinds of such invariants, namely *trap* and *mutex* invariants:

Definition 2. *Given a marked PN* $\mathcal{N} = (\langle S, T, E \rangle, \mathrm{m}_0)$*, a set* $\theta \subseteq S$ *is a:*
1. trap *if* $|\theta \cap \mathrm{m}_0| \geq 1$ *and, for any* $t \in T$*, if* $|\theta \cap {}^\bullet t| \geq 1$ *then* $|\theta \cap t^\bullet| \geq 1$*.*
2. mutex *if* $|\theta \cap \mathrm{m}_0| = 1$ *and, for any* $t \in T$*, we have* $|\theta \cap {}^\bullet t| = |\theta \cap t^\bullet| \leq 1$*.*
The structural invariants of $\mathcal{N}$ *are the trap and mutex invariants, repectively:*
A. $\Theta(\mathcal{N}) \stackrel{\text{def}}{=} \{\mathrm{m}$ *marking of* $\mathcal{N} \mid |\mathrm{m} \cap \theta| \geq 1$*, for each trap* θ *of* $\mathcal{N}\}$
B. $\Omega(\mathcal{N}) \stackrel{\text{def}}{=} \{\mathrm{m}$ *marking of* $\mathcal{N} \mid |\mathrm{m} \cap \theta| = 1$*, for each mutex* θ *of* $\mathcal{N}\}$*.*

Note that, since $\mathcal{N}$ is boolean, each marking can be represented as a set of places. Moreover, it is easy to check that $\Theta(\mathcal{N})$ and $\Omega(\mathcal{N})$ contain the initial marking m_0 and are closed under the transition relation of the net. Thus both sets are inductive invariants of $\mathcal{N}$, that can be used to prove a safety property, by checking the emptiness of the intersection of the above sets with a set $\mathcal{E}$ of error markings.

Our method encodes the families of sets $\{\Theta(\mathcal{N}) \mid \mathcal{N} \in [\![\mathsf{b}]\!]_{\mathcal{R}}\}$ and $\{\Omega(\mathcal{N}) \mid \mathcal{N} \in [\![\mathsf{b}]\!]_{\mathcal{R}}\}$ by formulæ of $\mathsf{WS}\kappa\mathsf{S}$, for a suitable integer constant $\kappa \geq 1$. To prove a parametric safety property given by a $\mathsf{WS}\kappa\mathsf{S}$ encoding of the $\mathcal{E}$ set, a sufficient (but not necessary) condition is that the $\mathsf{WS}\kappa\mathsf{S}$ formula defining the family of sets $\{\Theta(\mathcal{N}) \cap \Omega(\mathcal{N}) \cap \mathcal{E} \mid \mathcal{N} \in [\![\mathsf{b}]\!]_{\mathcal{R}}\}$ is unsatisfiable. Since automata-theoretic decision procedures exist for $\mathsf{WS}\kappa\mathsf{S}$ [19], we rely on existing provers [11] to perform this check.

Rewriting trees The crux of the method is to represent each predicateless behavioral term $\mathsf{t} \in [\mathsf{b}]_{\mathcal{R}}$ by a tree labeled with the rewriting rules from some rewriting sequence $\mathsf{b} \Leftarrow^*_{\mathcal{R}} \mathsf{t}$. As will be shown below, each such *rewriting tree*

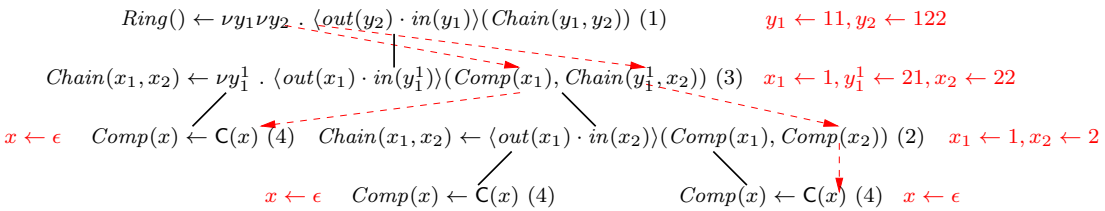

Fig. 3. A Rewriting Tree for the Token-Ring System

(Def. 3) defines an injective valuation $\nu : \mathbb{V}_1 \to [1, \kappa]^*$ of the bound variables from t (Def. 4) that, in turn, induces a behavior $\mathsf{B}^{\mathsf{t}}_{\nu} \in [\![\mathsf{b}]\!]_{\mathcal{R}}$ (7). Since each term $\mathsf{t} \in [\mathsf{b}]_{\mathcal{R}}$ can be represented by a rewriting tree (Prop. 1), it follows that each behavior $\mathcal{N} \in [\![\mathsf{b}]\!]_{\mathcal{R}}$ corresponds to a rewriting tree (up to a permutation of component identifiers). Our encoding uses rewriting trees $\mathcal{T}$ as backbone parameters for the definition of the trap ($\Theta(\mathcal{N}(\mathcal{T}))$) and mutex ($\Omega(\mathcal{N}(\mathcal{T}))$) invariant, respectively, where $\mathcal{N}(\mathcal{T}) = \mathsf{B}^{\mathsf{t}}_{\nu}$ is the behavior induced by $\mathcal{T}$. In fact, any injective valuation of the variables is sufficient for safety checking, provided that the safety properties considered are defined by sets of error markings that are closed under permutations of component identifiers.

To simplify technicalities, we assume the existence of a rule $\mathsf{A}_{\mathsf{b}}(x_1, \ldots, x_n) \leftarrow \mathsf{b}$ in $\mathcal{R}$, where $\mathrm{fv}(\mathsf{b}) = \{x_1, \ldots, x_n\}$ and A_{b} is a predicate symbol of arity n not occurring elsewhere in $\mathcal{R}$. We also assume that the constant κ is greater than the number of predicate atoms that occur in any rule of $\mathcal{R}$.

Definition 3. *Given a rewriting system $\mathcal{R}$ and a behavioral term b, a* rewriting tree *for b is a tree $\mathcal{T} : [1, \kappa]^* \to \mathcal{R}$, such that:*

1. $\mathcal{T}(\epsilon) = (\mathsf{A}_{\mathsf{b}}(x_1, \ldots, x_n) \leftarrow \mathsf{b})$

and, for all nodes $w \in \mathrm{nodes}(\mathcal{T})$, such that $\mathcal{T}(w) = \big(\mathsf{A}_w(x_1, \ldots, x_{\#(\mathsf{A}_w)}) \leftarrow \mathsf{b}_w\big)$, the following hold:

2. *for all $i \in [1, \#_{\mathsf{pred}}(\mathsf{b}_w)]$, if $\mathsf{pred}_i(\mathsf{b}_w) = \mathsf{A}_{wi}(y_1, \ldots, y_{\#(\mathsf{A}_{wi})})$ then $wi \in \mathrm{nodes}(\mathcal{T})$ and $\mathcal{T}(wi) = \big(\mathsf{A}_{wi}(x_1, \ldots, x_{\#(\mathsf{A}_{wi})}) \leftarrow \mathsf{b}_{wi}\big)$, for some rule of the form $\mathsf{A}_{wi}(x_1, \ldots, x_{\#(\mathsf{A}_{wi})}) \leftarrow \mathsf{b}_{wi}$ from $\mathcal{R}$.*
3. *for all $i \geq \#_{\mathsf{pred}}(\mathsf{b}_w)$, we have $wi \notin \mathrm{nodes}(\mathcal{T})$.*

We denote by $\mathbb{T}_{\mathcal{R}}\big(\mathsf{b}\big)$ the set of rewriting trees for b in $\mathcal{R}$.

A rewriting tree $\mathcal{T}$ induces a *characteristic term* $\mathfrak{C}_{[\mathcal{T}]}$, obtained by the application of the rewriting rules labeling the tree nodes in some order of traversal, and a *characteristic valuation* $\nu_{[\mathcal{T}]}$, that maps each variable in the term to the node where it is instantiated.

Definition 4. *Given a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}\big(\mathsf{b}\big)$, the* characteristic term $\mathfrak{C}_{[\mathcal{T}]}$ *and* characteristic valuation $\nu_{[\mathcal{T}]}$ *are defined inductively on the structure of $\mathcal{T}$:*

- *if* $\mathrm{nodes}(\mathcal{T}) = \{\epsilon\}$, *then* $\mathfrak{C}_{[\mathcal{T}]} \stackrel{\text{def}}{=} \mathcal{B}(x)$ *and* $\nu_{[\mathcal{T}]}(x) \stackrel{\text{def}}{=} \epsilon$, *for* $\mathcal{T}(\epsilon) = (\mathsf{A}(x) \leftarrow \mathcal{B}(x))$,
- *else, let $1, \ldots, m$ be the children of the root of $\mathcal{T}$ and let:*

$$\mathfrak{C}_{[\mathcal{T}]} \stackrel{\text{def}}{=} \nu y_1 \ldots \nu y_n \,.\, \langle \Gamma \rangle (\mathfrak{C}_{[\mathcal{T}_1]}[z^1_1/x_1, \ldots, z^1_{\#(\mathsf{A}_1)}/x_{\#(\mathsf{A}_1)}], \ldots, \mathfrak{C}_{[\mathcal{T}_m]}[z^m_1/x_1, \ldots, z^m_{\#(\mathsf{A}_1)}/x_{\#(\mathsf{A}_m)}])$$

where $\nu_{[\mathcal{T}]}(z^i_j) \stackrel{\text{def}}{=} i \cdot \nu_{[\mathcal{T}\downarrow_i]}(x_j)$*, for all* $i \in [1,m]$ *and* $j \in [1, \#(\mathsf{A}_i)]$*, such that*

$$\mathcal{T}(\epsilon) = \big(\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \nu y_1 \ldots \nu y_n \,.\, \langle \Gamma \rangle (\mathsf{A}_1(z^1_1,\ldots,z^1_{\#(\mathsf{A}_1)}),\ldots, \mathsf{A}_m(z^m_1,\ldots,z^m_{\#(\mathsf{A}_m)}))\big)$$

A consequence of the specific form of rewriting rules (Def. 1) is that each (free or bound) variable y from $\mathfrak{C}_{[\mathcal{T}]}$ is mapped by $\nu_{[\mathcal{T}]}$ to a unique node $w \in \text{nodes}(\mathfrak{C}_{[\mathcal{T}]})$, such that $\mathfrak{C}_{[\mathcal{T}]}(w)$ contains an instance atom $\mathcal{B}(x)$, where x is substituted with y along the path from the node u that introduces y (take $u = \epsilon$ if $y \in \text{fv}(\mathfrak{C}_{[\mathcal{T}]})$) to w. For example, Fig. 3 shows the rewriting tree for the rewriting sequence from Example 3; we annotate on the side of the tree the bottom-up definition of the characteristic valuation associating the bound variables y_1 and y_2 with the nodes of the rewriting tree where they are instantiated.

Below we show that the set $[\mathsf{b}]_{\mathcal{R}}$ of predicateless terms obtained by complete rewriting is the same as the set of characteristic terms that correspond to some rewriting tree:

Proposition 1. *Given a behavioral term* b*, we have* $[\mathsf{b}]_{\mathcal{R}} = \{\mathfrak{C}_{[\mathcal{T}]} \mid \mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})\}$.

Its proof is an easy consequence of the confluence of the rewriting system, i.e. the order in which the rules are applied to a term does not change the resulting predicateless term.

4.1 Encoding Invariants and Error Configurations

We begin by building a $\mathsf{WS}\kappa\mathsf{S}$ formula that describes an infinite κ-ary tree whose finite prefix encodes a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$. Let us assume that $\mathcal{R} = \{\mathsf{r}_1,\ldots,\mathsf{r}_N\}$, such that $\mathsf{r}_1 = (\mathsf{A}_{\mathsf{b}}(x_1,\ldots,x_n) \leftarrow \mathsf{b})$. We use a designated tuple of second order variables $\mathbf{U} = \langle U_1,\ldots,U_N\rangle$, where each variable U_i is interpreted as the set of tree nodes labeled with the rule r_i in $\mathcal{T}$. With this convention, the *RTree*($\mathbf{U}$) formula (Fig. 4) defines a rewriting tree:

- line (8) states that the sets $\mathbf{U}$ are pairwise disjoint and that U_1 is a singleton containing the root of the tree (condition 1 of Def. 3).
- line (9) states that the union of the sets $\mathbf{U}$ is prefix-closed, i.e. the parent x of each node $x.\ell$ from some U_i belongs to some U_j, for $i,j \in [1,N]$.
- lines (10) and (11) encode the conditions 2 and 3 of Def. 3, respectively.

Clearly, for each model ν of *RTree*($\mathbf{U}$), there is a unique rewriting tree, denoted $\mathcal{T}^{\mathbf{U}}_{\nu} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$, such that $\text{nodes}(\mathcal{T}^{\mathbf{U}}_{\nu}) = \bigcup_{i=1}^{N} \nu(U_i)$ and $\mathcal{T}^{\mathbf{U}}_{\nu}(w) = \mathsf{r}_i$ iff $w \in \nu(U_i)$, for all $i \in [1,N]$ and $w \in \text{nodes}(\mathcal{T}^{\mathbf{U}}_{\nu})$.

As said, a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$ defines a behavior (i.e. a marked PN) denoted by $\mathcal{N}(\mathcal{T}) \stackrel{\text{def}}{=} \mathsf{B}^{\mathsf{t}}_{\nu}$ (6), where $\mathsf{t} = \mathfrak{C}_{[\mathcal{T}]}$ is the characteristic term and $\nu = \nu_{[\mathcal{T}]}$ is the characteristic valuation of $\mathcal{T}$ (Def. 4). An invariant of $\mathcal{N}(\mathcal{T})$ is a set of markings, i.e. a set of sets of marked places from different components.

Let $\{\mathcal{B}_i \stackrel{\text{def}}{=} \langle \mathcal{P}_i, \mathcal{S}_i, \mathcal{I}_i, \Delta_i\rangle\}_{i=1}^{K}$ be the set of component types that occur in the rules of $\mathcal{R}$ and let $\mathbf{Z} = \langle Z_1,\ldots,Z_K\rangle$ be a tuple of second-order variables, where Z_i is interpreted as the set of identifiers of the components of type $\mathcal{B}_i$.

$$RTree(\mathbf{U}) \stackrel{\text{def}}{=} \forall x \ . \bigwedge_{1 \leq i < j \leq N} \Big(\neg U_i(x) \vee \neg U_j(x) \Big) \wedge U_1(x) \leftrightarrow x = \epsilon \ \wedge \tag{8}$$

$$\forall x \ . \bigwedge_{i : \mathsf{r}_i \in \mathcal{R}} \bigwedge_{\ell=1}^{\kappa} U_i(x.\ell) \rightarrow \bigvee_{\mathsf{r}_j \in \mathcal{R}} U_j(x) \ \wedge \tag{9}$$

$$\forall x \ . \bigwedge_{i : \mathsf{r}_i = \left(\mathsf{A}'(x_1, \ldots, x_{\#(\mathsf{A}')}) \leftarrow \mathsf{b}'\right)} \bigwedge_{j=1}^{\#_{\mathsf{pred}}(\mathsf{b}')} U_i(x) \rightarrow \bigvee_{\substack{j : \mathsf{pred}_j(\mathsf{b}') = \mathsf{A}''(\xi_1, \ldots, \xi_{\#(\mathsf{A}'')}) \\ \ell : \mathsf{r}_\ell = \left(\mathsf{A}''(x_1, \ldots, x_{\#(\mathsf{A}'')}) \leftarrow \mathsf{b}''\right)}} U_\ell(x.j) \ \wedge \tag{10}$$

$$\forall x \ . \bigwedge_{i : \mathsf{r}_i = \left(\mathsf{A}'(x_1, \ldots, x_{\#(\mathsf{A}')}) \leftarrow \mathsf{b}'\right)} \bigwedge_{j=\#_{\mathsf{pred}}(\mathsf{b}')+1}^{\kappa} U_i(x) \rightarrow \bigwedge_{\ell=1}^{N} \neg U_\ell(x.j) \tag{11}$$

Fig. 4. The Definition of Rewriting Trees

We encode the markings of $\mathcal{N}(\mathcal{T})$ by a $\mathsf{WS}\kappa\mathsf{S}$ formula using a tuple of second-order variables $\mathbf{X} = \langle X_S \mid S \in \bigcup_{i=1}^{K} \mathcal{S}_i \rangle$, where each X_S is interpreted as the set of identifiers of the components currently in state $S(w)$ and define the set $\sigma_\nu^{\mathbf{X}} \stackrel{\text{def}}{=} \{S(w) \mid S \in \bigcup_{i=1}^{K} \mathcal{S}_i, \ w \in \nu(X_S)\}$. The following formula constrains the set represented by $\mathbf{X}$ to be a marking of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$:

$$\mathit{mark}(\mathbf{X}, \mathbf{Z}) \stackrel{\text{def}}{=} \forall x. \bigwedge_{S \neq S' \in \bigcup_{j=1}^{K} \mathcal{S}_j} \big(\neg X_S(x) \vee \neg X_{S'}(x) \big) \wedge \bigvee_{S \in \bigcup_{j=1}^{K} \mathcal{S}_j} X_S(x) \leftrightarrow \bigvee_{j=1}^{K} Z_j(x)$$

$$\mathit{inst}(\mathbf{Z}, \mathbf{U}) \stackrel{\text{def}}{=} \forall x. \bigwedge_{i=1}^{K} Z_i(x) \leftrightarrow \bigvee_{j : \mathsf{r}_j = \left(\mathsf{A}'(y) \leftarrow \mathcal{B}_i(y)\right)} U_j(x)$$

Intuitively, $\mathit{mark}(\mathbf{X}, \mathbf{Z})$ states that no component can be in two different states (first conjunct) and each component is an instance of some component type (second conjunct). The formula $\mathit{inst}(\mathbf{Z}, \mathbf{U})$ above relates the instance indices to the nodes of the rewriting tree where the corresponding instance atoms occur, assuming that the sets $\mathbf{U}$ are constrained by $\mathit{RTree}(\mathbf{U})$. Then, for each model ν of $\mathit{mark}(\mathbf{X}, \mathbf{Z}) \wedge \mathit{inst}(\mathbf{Z}, \mathbf{U}) \wedge \mathit{RTree}(\mathbf{U})$, the set $\sigma_\nu^{\mathbf{X}}$ is a marking of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.

We proceed with the encoding of invariants and error states, by assuming the existence of a *flow formula*, that defines the pre- and post-sets of the transitions from a behavior $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$, formally described next (§4.2). In the following, the primed copy of the tuple $\mathbf{X}$ is denoted as $\mathbf{X}'$.

Definition 5. *$\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ is a* flow formula *for b and $\mathcal{R}$ if, for each model ν of $\mathit{RTree}(\mathbf{U})$, we have $\nu \models \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ if and only if ${}^\bullet t = \sigma_\nu^{\mathbf{X}}$ and $t^\bullet = \sigma_\nu^{\mathbf{X}'}$, for some transition t of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.*

Given a flow formula Φ, the parametric trap invariant $\mathit{TrapInv}^\Phi(\mathbf{X}, \mathbf{U})$ is defined by the formula below:

$$\mathit{trap}^\Phi(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \ \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \wedge \mathit{inter}(\mathbf{X}, \mathbf{Y}^1) \rightarrow \mathit{inter}(\mathbf{X}, \mathbf{Y}^2)$$

$$\mathit{TrapInv}^\Phi(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \exists \mathbf{Z} \ . \ \mathit{mark}(\mathbf{X}, \mathbf{Z}) \wedge \forall \mathbf{Y}^1, \mathbf{Y}^2. \ \mathit{init}(\mathbf{Y}^1, \mathbf{Z}) \wedge \mathit{inter}(\mathbf{Y}^1, \mathbf{Y}^2) \wedge \mathit{trap}^\Phi(\mathbf{Y}^2, \mathbf{U}) \rightarrow \mathit{inter}(\mathbf{X}, \mathbf{Y}^2)$$

where the tuples $\mathbf{Y}^i \stackrel{\text{def}}{=} \langle Y^i_S \mid S \in \bigcup_{j=1}^K \mathcal{S}_j \rangle$, for $i = 1, 2$, are distinct copies of $\mathbf{X}$. The auxiliary formula $init(\mathbf{X}, \mathbf{Z}) \stackrel{\text{def}}{=} mark(\mathbf{X}, \mathbf{Z}) \wedge \bigwedge_{j=1}^K \forall x \ . \ Z_j(x) \leftrightarrow X_{\mathcal{I}_j}(x)$ states that $\mathbf{X}$ represents the initial marking of the behavior, whereas $inter(\mathbf{X}, \mathbf{Y}) \stackrel{\text{def}}{=} \exists x. \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} X_S(x) \wedge Y_S(x)$ means that the sets of places encoded by $\mathbf{X}$ and $\mathbf{Y}$, respectively, have a non-empty intersection. Intuitively, the formula $trap^{\Phi}(\mathbf{X}, \mathbf{U})$ defines the traps (point 1 of Def. 2), whereas $TrapInv^{\Phi}(\mathbf{X}, \mathbf{U})$ defines the set of markings that intersect with the initial marking and with each trap of the behavior, i.e. the trap invariant (point A of Def. 2.

Mutexes and mutex invariants (points 2 and B of Def. 2) are defined by the formulæ:

$$mutex^{\Phi}(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \ \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \rightarrow \bigwedge \begin{array}{c} \neg inter(\mathbf{X}, \mathbf{Y}^1) \leftrightarrow \neg inter(\mathbf{X}, \mathbf{Y}^2) \\ single(\mathbf{X}, \mathbf{Y}^1) \leftrightarrow single(\mathbf{X}, \mathbf{Y}^2) \end{array}$$

$$MutexInv^{\Phi}(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \exists \mathbf{Z} \ . \ mark(\mathbf{X}, \mathbf{Z}) \wedge \forall \mathbf{Y}^1, \mathbf{Y}^2. \ init(\mathbf{Y}^1, \mathbf{Z}) \wedge single(\mathbf{Y}^1, \mathbf{Y}^2) \wedge mutex^{\Phi}(\mathbf{Y}^2, \mathbf{U}) \rightarrow single(\mathbf{X}, \mathbf{Y}^2)$$

where $single(\mathbf{X}, \mathbf{Y}) \stackrel{\text{def}}{=} \exists_1 x \ . \ \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} X_S(x) \wedge Y_S(x)$ states that the intersection of the sets of places defined by $\mathbf{X}$ and $\mathbf{Y}$ is a singleton[3]. The following lemma states the correctness of the encoding:

Lemma 1. *Given a flow formula $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ and a model ν of $RTree(\mathbf{U}) \wedge inst(\mathbf{Z}, \mathbf{U})$, we have:*

1. $\Theta(\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})) = \{\sigma_\mu^{\mathbf{X}} \mid \mu \models TrapInv^{\Phi}(\mathbf{X}, \mathbf{U}), \ \mu(\mathbf{U} \cdot \mathbf{Z}) = \nu(\mathbf{U} \cdot \mathbf{Z})\}$,
2. $\Omega(\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})) = \{\sigma_\mu^{\mathbf{X}} \mid \mu \models MutexInv^{\Phi}(\mathbf{X}, \mathbf{U}), \ \mu(\mathbf{U} \cdot \mathbf{Z}) = \nu(\mathbf{U} \cdot \mathbf{Z})\}$.

In our examples (§5) we consider two kinds of error sets, defined as:

$$DeadLock^{\Phi}(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \ \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \rightarrow \exists x. \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} Y^1_S(x) \wedge \neg X_S(x)$$

$$CriticalSection^{\Xi}(\mathbf{X}, \mathbf{U}) \stackrel{\text{def}}{=} \exists x \exists y \ . \ \bigvee_{S, S' \in \ \Xi} X_S(x) \wedge X_{S'}(y) \wedge \neg x = y$$

Intuitively, $DeadLock^{\Phi}(\mathbf{X}, \mathbf{U})$ defined the deadlock markings, in which no transition of the behavior is enabled and $CriticalSection^{\Xi}(\mathbf{X}, \mathbf{U})$ states that no two components are at the same time in a state from a given set $\Xi \subseteq \bigcup_{i=1}^K \mathcal{S}_i$ of state types. It is worth mentioning that these sets of error markings are closed under permutations of component indices[4], which makes them suitable for safety checking using our encoding of trap and mutex invariants (with variables mapped to the nodes of the rewriting tree where they occur instantiated, as in Fig. 3).

4.2 The Flow of a Behavioral Term

The previous definition of structural invariants relied on the existence of a flow formula $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$, stating that $\sigma_\nu^{\mathbf{X}}$ and $\sigma_\nu^{\mathbf{X}'}$ are the pre- and post-sets of some transition from the behavior (i.e. marked PN) $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$, whenever ν is a model of $RTree(\mathbf{U}) \wedge \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ (Def. 5). In this section, we describe the flow formula. In the following, we assume w.l.o.g. that the inductive rules in $\mathcal{R}$ are of the form:

$$\mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \leftarrow \nu y_1 \ldots \nu y_m \ . \Big\langle \sum_{k=1}^{m} P_{k1}(x_{k1}) \cdot \ldots \cdot P_{kr_k}(x_{kr_k}) \Big\rangle (\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

[3] $\exists_1 x \ . \ \phi(x)$ is a shorthand for $\exists x \ . \ \phi(x) \wedge \forall x \forall y \ . \ \phi(x) \wedge \phi(y) \rightarrow x = y$.

[4] This is the case for every $\mathsf{WS}\kappa\mathsf{S}$ formula consisting of equality and membership atoms, without successor functions.

$$\Phi(\mathbf{X},\mathbf{X}',\mathbf{U}) \stackrel{\text{def}}{=} \bigvee_{\ell=1}^{N} \bigvee_{\pi\in\mathsf{Inter}(\mathsf{r}_\ell)} \Psi_{\ell,\pi}(\mathbf{X},\mathbf{X}',\mathbf{U}) \tag{12}$$

$$\Psi_{\ell,\{P_1(x_1),\ldots,P_n(x_n)\}}(\mathbf{X},\mathbf{X}',\mathbf{U}) \stackrel{\text{def}}{=} \exists y_0 \ldots \exists y_n \ . \ U_\ell(y_0) \ \wedge \tag{13}$$

$$\bigwedge_{i=1}^{n} \Big(\bigvee_{\mathsf{r}'=\left(\mathsf{A}'(y_i)\leftarrow\mathcal{B}(y_i)\right)} Path^{x_i,y_i}_{\mathsf{r}_\ell,\mathsf{r}'}(y_0,y_i,\mathbf{U}) \Big) \ \wedge \tag{14}$$

$$\forall x. \bigwedge_{S\in\bigcup_{j=1}^{K}\mathcal{S}_j} \Big[\Big(X_S(x) \leftrightarrow \bigvee_{\bullet P_k=S} x = y_k \Big) \wedge \Big(X'_S(x) \leftrightarrow \bigvee_{P_k\bullet=S} x = y_k \Big) \Big] \tag{15}$$

Fig. 5. Definition of the Flow Formula for the Rewriting System $\mathcal{R} = \{\mathsf{r}_1, \ldots, \mathsf{r}_N\}$.

if necessary, by applying the flattening relation (5) to each rule of $\mathcal{R}$. We denote by $\mathsf{Inter}(\mathsf{r}) \stackrel{\text{def}}{=} \big\{\{P_{ki}(x_{ki}) \mid i \in [1, r_k]\} \mid k \in [1, m]\big\}$ the set of sets of port atoms $P_{ki}(x_{ki})$, corresponding to the interactions (i.e. the monomials from the architecture) from r.

The flow formula $\Phi(\mathbf{X},\mathbf{X}',\mathbf{U})$ is given in Fig. 5. Essentially, the formula (12) is split into a disjunction of formulæ $\Psi_{\ell,\{P_1(x_1),\ldots,P_n(x_n)\}}$ (13), one for each rule $\mathsf{r}_\ell \in \mathcal{R}$ and each set of port atoms $\{P_1(x_1), \ldots, P_n(x_n)\}$, denoting an interaction (monomial) from r_ℓ. To understand the formulæ (13), recall that each of the variables $x_1, \ldots, x_n$ is mapped to the (unique) node of the rewriting tree containing an instance atom $\mathcal{B}_i(x_i)$. In order to find this node, we must track the variable x_i from the node labeled by the rule r, to the node where this instance atom occurs. This is done by the $Path^{x_i,y_i}_{\mathsf{r}_\ell,\mathsf{r}'}(y_0,y_i,\mathbf{U})$ formulæ (14), that holds whenever $\mathcal{T}^{\mathbf{U}}_\nu$ is a rewriting tree, uniquely encoded by the interpretation ν of the $\mathbf{U}$ variables, and y_0, y_i are mapped to the source and the destination of a path from a node $w \in \text{nodes}(\mathcal{T}^{\mathbf{U}}_\nu)$, with label $\mathcal{T}^{\mathbf{U}}_\nu(w) = \mathsf{r}_\ell$ to a node $w' \in \text{nodes}(\mathcal{T}^{\mathbf{U}}_\nu)$, with label $\mathcal{T}^{\mathbf{U}}_\nu(w') = \mathsf{r}'$, such that x_i and y_i are variables that occur in the bodies of r and r', respectively, mapped to the same variable in the characteristic term of $\mathcal{T}^{\mathbf{U}}_\nu$ (Def. 4). We describe these paths by an automaton and define $Path^{x_i,y_i}_{\mathsf{r}_\ell,\mathsf{r}'}(y_0,y_i,\mathbf{U})$, by translating this automaton into a $\mathsf{WS}\kappa\mathsf{S}$ formula.

But first, let us define paths in a tree formally. Given a κ-ary tree $\mathcal{T}$, a *path* is a finite sequence of nodes $\rho = n_1, \ldots, n_\ell \in \text{nodes}(\mathcal{T})$, such that, for all $i \in [1, \ell-1]$, n_{i+1} is either the parent ($n_i = n_{i+1}\alpha_i$) or a child ($n_{i+1} = n_i\alpha_i$) of n_i, for some $\alpha_i \in [1, \kappa]$. The path is determined by the source node and the sequence $(\alpha_1, d_1) \ldots (\alpha_{\ell-1}, d_{\ell-1})$ of *directions* $(\alpha_i, d_i) \in [1, \kappa] \times \{\uparrow, \downarrow\}$, with the following meaning: $d_i = \uparrow$ if $n_{i+1}\alpha_i = n_i$ and $d_i = \downarrow$ if $n_{i+1} = n_i\alpha_i$. Given two distinct nodes $w_1, w_2 \in \text{nodes}(\mathcal{T})$, there is a unique minimal path from w_1 to w_2, labeled by a sequence denoted as $\rho(w_1, w_2)$. This path climbs from w_1 to the greatest common prefix w of w_1 and w_2, before descending from w to w_2.

A *path automaton* is a tuple $A = (Q, I, F, \delta)$, where Q is a set of states, $I, F \subseteq Q$ are the sets of initial and final states, respectively, and $\delta \subseteq Q \times [1, \kappa] \times \{\uparrow, \downarrow\} \times Q$ is a set of transitions $q \xrightarrow{(\alpha,d)} q'$, with $\alpha \in [1, \kappa]$ being a direction and $d \in \{\uparrow, \downarrow\}$ indicating whether the automaton moves up or down in the tree. A run of A over $\omega = (\alpha_1, d_1) \ldots (\alpha_{n-1}, d_{n-1})$ is a sequence of states $q_1, \ldots, q_n \in Q$ such that $q_1 \in I$ and $q_i \xrightarrow{(\alpha_i,d_i)} q_{i+1} \in \delta$, for all $i \in [1, n-1]$. The run is accepting if and only if $q_n \in F$. The *language* $\mathcal{L}(A)$ of A is the set of sequences over which A has an accepting run.

A path automaton $A = (Q, I, F, \delta)$ corresponds (Lemma 2) to the formula in Figure 6, that can be effectively built from the description of A. Here $Q = \{q_1, \ldots, q_L\}$ is the

$$\Delta_A(x, y, \mathbf{Y}) \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \neq j \leq L} \forall z. \Big(\neg Y_i(z) \vee \neg Y_j(z) \Big) \wedge \bigvee_{q_i \in I} Y_i(x) \ \wedge \ \bigvee_{q_j \in F} Y_j(y) \ \wedge \quad (16)$$

$$\bigwedge_{i=1}^{L} \forall z \ . \ z \neq y \wedge Y_i(z) \rightarrow \bigvee_{j : q_i \xrightarrow{(\alpha,\downarrow)} q_j} Y_j(z.\alpha) \vee \bigvee_{j : q_i \xrightarrow{(\alpha,\uparrow)} q_j} \exists z' \ . \ z'.\alpha = z \wedge Y_j(z') \ \wedge \quad (17)$$

$$\bigwedge_{j=1}^{L} \forall z \ . \ z \neq x \wedge Y_j(z) \rightarrow \bigvee_{i : q_i \xrightarrow{(\alpha,\downarrow)} q_j} \exists z' \ . \ z'.\alpha = z \wedge Y_i(z') \vee \bigvee_{i : q_i \xrightarrow{(\alpha,\uparrow)} q_j} Y_i(z.\alpha) \quad (18)$$

Fig. 6. Definition of the Path Automaton formula $\Delta_A(x, y, \mathbf{Y})$

set of states of A and $\mathbf{Y} = \langle Y_1, \ldots, Y_L \rangle$ are second order variables interpreted as the sets of tree nodes labeled by the automaton with $q_1, \ldots, q_L$, respectively. Intuitively, the first three conjuncts of the above formula (16) encode the facts that $\mathbf{Y}$ are disjoint (no tree node is labeled by more than one state during the run) and that the run starts in an initial state with node x and ends in a final state with node y. The fourth conjunct (17) states that, for every non-final node on the path, if the automaton visits that node by state q_i, then either the node has a $(\alpha, \downarrow)$-child or a $(\alpha, \uparrow)$-parent visited by state q_j, where $q_i \xrightarrow{(\alpha,\downarrow)} q_j$ and $q_i \xrightarrow{(\alpha,\uparrow)} q_j$ are transitions of the automaton. The fifth conjunct (18) is the reversed flow condition on the path, needed to ensure that the sets $\mathbf{Y}$ do not contain useless nodes, being thus symmetric to the fourth. The following result stems from the classical automata-logic connection[5] [14, §2.10]:

Lemma 2. *Given a tree $\mathcal{T}$ with* $\text{nodes}(\mathcal{T}) \subseteq [1, \kappa]^*$ *and a sequence $\omega \in ([1, \kappa] \times \{\uparrow, \downarrow\})^*$ from $w_1 \in \text{nodes}(\mathcal{T})$ to $w_2 \in \text{nodes}(\mathcal{T})$, for each valuation ν such that $\nu(x) = w_1$ and $\nu(y) = w_2$, we have $\omega \in \mathcal{L}(A) \iff \nu \models \exists \mathbf{Y} \ . \ \Delta_A(x, y, \mathbf{Y})$.*

Our purpose is to define path automata that recognize the paths between the node where a bound variable is introduced and the node where the variable is instantiated, in a given rewriting tree. This automaton is directly inferred from the syntax of the rules in $\mathcal{R}$. For each pair of rules $\mathsf{r}_1, \mathsf{r}_2 \in \mathcal{R}$ and variables $z_1, z_2 \in \mathbb{V}_1$ that occur in the bodies of r_1 and r_2, respectively, we define the path automaton $A^{z_1, z_2}_{\mathsf{r}_1, \mathsf{r}_2} \stackrel{\text{def}}{=} (Q, I^{z_1}_{\mathsf{r}_1}, F^{z_2}_{\mathsf{r}_2}, \delta)$:

– We associate a state $q^d_{\mathsf{r},z}$ to each rule $\mathsf{r} = \big(\mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \overline{\mathsf{b}}\big)$, each variable z occurring (free or bound) in $\overline{\mathsf{b}}$ and each direction $d \in \{\uparrow, \downarrow\}$. The intuition is that the automaton visits the state $q^d_{\mathsf{r},z}$ while going up or down, as indicated by the direction d, currently tracking variable z in rule r.

[5] A similar conversion of tree walking automata to MSO has been described in [12].

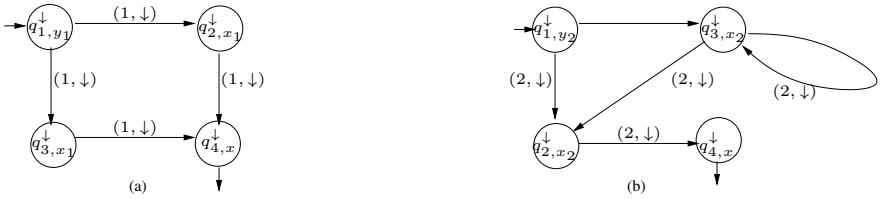

Fig. 7. Path Automata Recognizing the Instantiation Paths from Example 1

- The sets of initial and final states are $I^{z_1}_{\mathsf{r}_1} \stackrel{\text{def}}{=} \{q^{d}_{\mathsf{r}_1,z_1} \mid d = \uparrow,\downarrow\}$ and $F^{z_2}_{\mathsf{r}_2} \stackrel{\text{def}}{=} \{q^{\downarrow}_{\mathsf{r}_2,z_2}\}$. In other words, the automaton starts to track z_1 in r_1, moving either up or down and it ends tracking z_2 in r_2, while moving down.
- The transitions are $q^{\downarrow}_{\mathsf{r}_1,y_j} \xrightarrow{(\alpha,\downarrow)} q^{\downarrow}_{\mathsf{r}_2,x_j}$, $q^{\uparrow}_{\mathsf{r}_2,x_j} \xrightarrow{(\alpha,\uparrow)} q^{\uparrow}_{\mathsf{r}_1,y_j}$ and $q^{\uparrow}_{\mathsf{r}_2,x_j} \xrightarrow{(\alpha,\uparrow)} q^{\downarrow}_{\mathsf{r}_1,y_j}$, for any two distinct rules $\mathsf{r}_i = (\mathsf{A}_j(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \mathsf{b}_i)$, $i = 1,2$, all $\alpha \in [1, \#_{\mathsf{pred}}(\mathsf{b}_1)]$, such that $\mathsf{pred}_{\alpha}(\mathsf{b}_1) = \mathsf{A}_2(y_1,\ldots,y_{\#(\mathsf{A}_2)})$ and all $j \in [1,\#(\mathsf{A}_2)]$. Intuitively, if r_1 labels the parent of the node labeled by r_2 in the rewriting tree, the automaton can move either: (i) down from tracking y_j in r_1 to tracking x_j in r_2, (ii) up from tracking x_j in r_2 to tracking y_j in r_1, or (iii) change direction from moving up tracking x_j in r_2 to moving down tracking y_j in r_1. In particular, the last case might be needed to accept a path that only goes up in the tree.

Note that a run of a path automaton $A^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}$ may have at most one change of direction, by a rule of the form $q^{\uparrow}_{\mathsf{r}_2,x_j} \xrightarrow{(\alpha,\uparrow)} q^{\downarrow}_{\mathsf{r}_1,y_j}$.

Example 4. The paths that track the instantiations of the variables y_1 and y_2 in a rewriting tree for the term $Ring()$ are depicted in dashed lines in Fig. 3. The path automata that recognize these paths are given in Fig. 7a (y_1) and Fig. 7b (y_2). The initial states are $q^{\downarrow}_{1,y_1}$ and $q^{\downarrow}_{1,y_2}$, respectively, and the final state is $q^{\downarrow}_{4,x}$ in both cases, where the labels (1-4) of the rewriting rules are the ones from Example 1. ■

The lemma below shows that these automata recognize exactly the labels of the minimal paths between two nodes:

Lemma 3. *Let $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$ be a rewriting tree and $w_i \in \mathrm{nodes}(\mathcal{T})$ be nodes labeled with the rules $\mathcal{T}(w_i) = (\mathsf{A}_i(x_{i,1},\ldots,x_{i,\#(\mathsf{A}_i)}) \leftarrow \mathsf{b}_i) = \mathsf{r}_i$, for $i = 1,2$. Then, for all $k_i \in [1,\#(\mathsf{A}_i)]$, $i = 1,2$, the following are equivalent:*

1. *x_{1,k_1} and x_{2,k_2} are substituted by the same variable during the construction of $\mathfrak{C}_{[\mathcal{T}]}$ (Def. 4),*
2. *$\rho(w_1,w_2) \in \mathcal{L}(A^{x_{1,k_1},\ x_{2,k_2}}_{\mathsf{r}_1,\mathsf{r}_2})$.*

The path automata $A^{x_{1,k_1},\ x_{2,k_2}}_{\mathsf{r}_1,\mathsf{r}_2}$ are used to define the $Path^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}$ formulæ:

$$Path^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}(x,y,\mathbf{U}) \stackrel{\text{def}}{=} \exists \mathbf{Y} \ .\ \Delta_{A^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}}(x,y,\mathbf{Y}) \wedge \Upsilon(\mathbf{Y},\mathbf{U})$$

$$\Upsilon(\mathbf{Y},\mathbf{U}) \stackrel{\text{def}}{=} \bigwedge_{d=\uparrow,\downarrow} \bigwedge_{i:\mathsf{r}_i=(\mathsf{A}'(x_1,\ldots,x_{\#(\mathsf{A}')})\leftarrow \mathsf{b}')} \bigwedge_{z\in \mathrm{fv}(\mathsf{b}')} \forall x \ .\ Y^{d}_{\mathsf{r},z}(x) \rightarrow U_i(x)$$

The formula $\Upsilon(\mathbf{Y},\mathbf{U})$ above states that all nodes labeled with a state $q^{d}_{\mathsf{r},z}$ during the run must be also labeled with r in the rewriting tree given as input to the path automaton. The lemma below proves that $\Phi(\mathbf{X},\mathbf{X}',\mathbf{U})$ (12) is indeed a flow formula (Def. 5):

Lemma 4. *For each model ν of RTree($\mathbf{U}$), we have $\nu \models \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ if and only if $\sigma_\nu^{\mathbf{X}} = {}^\bullet t$ and $\sigma_\nu^{\mathbf{X}'} = t^\bullet$ for some transition t of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.*

Together with Lemma 1, this ensures that the trap and mutex invariant of the parametric system described by b and $\mathcal{R}$ are defined by the $TrapInv^\Phi(\mathbf{X}, \mathbf{U})$ and $MutexInv^\Phi(\mathbf{X}, \mathbf{U})$ formulæ, respectively. Hence a sufficient condition that proves a safety property of the parametric system described by b and $\mathcal{R}$ is the unsatisfiability of a WSκS formula, obtained from the syntax of b and $\mathcal{R}$:

Theorem 1. *Let b be closed behavioral term, $\mathcal{R}$ be a rewriting system and $\mathcal{E}(\mathbf{X}, \mathbf{U})$ be a* WSκS *formula. The behavior $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$ is safe w.r.t. the set $\{\sigma_\mu^{\mathbf{X}} \mid \mu \models \mathcal{E}(\mathbf{X}, \mathbf{U}),\ \mu(\mathbf{U}) = \nu(\mathbf{U})\}$, for any valuation ν, if the formula $RTree(\mathbf{U}) \wedge TrapInv^\Phi(\mathbf{X}, \mathbf{U}) \wedge MutexInv^\Phi(\mathbf{X}, \mathbf{U}) \wedge \mathcal{E}(\mathbf{X}, \mathbf{U})$ is unsatisfiable.*

In particular, we have experimented with error sets defined by the $DeadLock^\Phi(\mathbf{X}, \mathbf{U})$ and $CriticalSection^\Xi(\mathbf{X}, \mathbf{U})$ formulæ, for some critical section given by $\Xi \in \bigcup_{i=1}^K \mathcal{S}_i$.

5 Experimental Evaluation

We implemented the structural invariant synthesis in a prototype tool[6]. Table 1 shows the results of checking deadlock freedom in all test cases and absence of critical section violations, for those test cases where a critical section was defined (otherwise marked n/a). The 2nd column gives the number of states in the system, in the form $n_1 \times \ldots \times n_K$, where n_i is the number of states in the i-th component type and K is the number of component types. The number of rewriting rules and interactions in the specification are given in the 3rd and 4th columns, respectively. The 5th and 7th columns report the results of the satisfiability check (✓means that the formula is unsatisfiable and $\times$ means that a counterexample has been found, in which case safety could not be proved using our method) for deadlock freedom and absence of critical section violations, using the MONA v1.4-18 tool [11]. The 6th and 8th columns show the total running times (in seconds) on an iMac 3,4 GHz with 32 GB of RAM, respectively (∞ means that MONA has run out of memory). The 9th column gives the branching degree $\kappa \in \{1, 2\}$ of trees in the WSκS logic. Note that star and token ring systems require $\kappa = 1$, whereas the tree-structured systems require $\kappa = 2$.

The test cases we consider are grouped according to the architectural pattern. *Token rings* (Fig. 1a) consist of instances of the same component type, such that the *out* port of a component is connected to the *in* port of the next component in the ring. *Dining philosophers* are special cases of token rings, consisting of alternating *philosopher* and *fork* instances. *Stars* consists of a single controller (master) sending requests and receiving replies from one or more slaves connected to it. Concerning *trees*, the tree-dfs example models a binary tree architecture traversed by a token in depth-first order, while tree-back-root and tree-linked-leaves(-generic) go beyond trees, modeling hierarchical systems with parent-children communication on top of which the nodes communicate with the root and the leaves are linked in a token-ring, respectively. These examples could not have been described using first order logic, as in [4]. The verification problems considered could be solved in less than 1 second, with the exception of the critical section violations for the tree-linked-leaves(-generic) examples, that require mutex, in addition to trap invariants. In particular, in the examples marked with -generic, the initial state of the components is arbitrary. Consequently, all these examples violate the critical section initially.

[6] `https://github.com/raduiosif/rtab`

Table 1. Experimental Results

Example	#states	#rules	#interaction types	deadlock freedom	time (secs)	critical section	time (secs)	κ
token-ring	2×2	3	3	✓	0.66	✓	0.63	1
token-ring-generic	2×2	5	4	✓	0.75	×	0.72	1
sync-philo	2×2	3	6	✓	0.69	✓	0.67	1
alt-philo-sym	3×2	3	9	×	0.75	✓	0.77	1
alt-philo-asym	3×2	3	9	✓	0.84	✓	0.78	1
alt-philo-generic	3×2	4	12	✓	0.91	✓	0.87	1
star	2×2	3	4	✓	0.58	n/a	-	1
star-ring	$2 \times 3 \times 3$	3	9	✓	0.75	✓	0.76	1
star-ring-generic	$2 \times 3 \times 3$	5	12	✓	0.84	×	0.88	1
tree-dfs	$2 \times 6 \times 2$	4	6	✓	0.70	n/a	-	2
tree-back-root	2×2	3	5	✓	0.60	n/a	-	2
tree-linked-leaves	$2 \times 2 \times 4 \times 3$	4	10	✓	1.05	✓	1.21	2
tree-linked-leaves-generic	$2 \times 2 \times 4 \times 3$	7	16	✓	1.31	×	1.73	2

6 Related Work

Traditionally, verification of unbounded networks of parallel processes considers known architectural patterns, typically cliques or rings [10,6]. Because the price for decidability is drastic restrictions on the shape of architectures [3], more recent works propose practical semi-algorithms, e.g. *regular model checking* [13,1] or *automata learning* [7]. Here the architectural pattern is implicitly determined by the class of language recognizers: word automata encode pipelines or rings, whereas tree automata describe trees. A first attempt at specifying architectures by logic is the *interaction logic* of Konnov et al. [15], which is a combination of Presburger arithmetic with monadic uninterpreted function symbols, that can describe cliques, stars and rings. More structured architectures (pipelines and trees) can be described using a second-order extension [17]. As such, these interaction logics are undecidable and have no support for automated verification. Recently, interaction logics that support the verification of safety properties by structural invariant synthesis have been developed. These logics use fragments of first order logic with interpreted function symbols that implicitly determine the shape of the architecture [5,4].

7 Conclusions and Future Work

We present a formal language for the specification of distributed systems parameterized by the number of replicated components and by the shape of the coordinating architecture. The language uses inductive definitions to describe systems of unbounded size. We propose a verification method for safety properties based on the synthesis of structural invariants able to prove deadlock freedom for a number of non-trivial models.

References

1. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis

of Systems, 13th International Conference, TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer (2007)
2. Barrett, C.W., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisf. Boolean Model. Comput. **3**(1-2), 21–46 (2007)
3. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015)
4. Bozga, M., Esparza, J., Iosif, R., Sifakis, J., Welzel, C.: Structural invariants for the verification of systems with parameterized architectures. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020. LNCS, vol. 12078, pp. 228–246. Springer (2020)
5. Bozga, M., Iosif, R., Sifakis, J.: Checking deadlock-freedom of parametric component-based systems. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019. LNCS, vol. 11428, pp. 3–20. Springer (2019)
6. Browne, M., Clarke, E., Grumberg, O.: Reasoning about networks with many identical finite state processes. Information and Computation **81**(1), 13 – 31 (1989)
7. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017. pp. 76–83. IEEE (2017)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 269–282. ACM Press, New York, NY (1979)
9. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 85–94. ACM Press (1995)
10. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
11. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95. LNCS, vol. 1019, pp. 89–110. Springer (1995)
12. Iosif, R., Rogalewicz, A., Simácek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction. LNCS, vol. 7898, pp. 21–38. Springer (2013)
13. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theoretical Computer Science **256**(1), 93–112 (2001)
14. Khoussainov, B., Nerode, A.: Automata Theory and its Applications. Springer (2001)
15. Konnov, I.V., Kotek, T., Wang, Q., Veith, H., Bliudze, S., Sifakis, J.: Parameterized systems in BIP: design and model checking. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016. LIPIcs, vol. 59, pp. 30:1–30:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)

16. Kramer, J., Magee, J.: Analysing dynamic change in distributed software architectures. IEE Proceedings - Software **145**(5), 146–154 (1998)
17. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: Modeling architecture styles. J. Log. Algebr. Meth. Program. **86**(1), 2–29 (2017)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002). pp. 55–74. IEEE Computer Society (2002)
19. Thatcher, J., Wright, J.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathematical systems theory **2**, 57–81 (2005)