Verifying Safety Properties of Inductively Defined Parameterized Systems

Marius Bozga, Radu Iosif

Verifying Safety Properties of Inductively Defined Parameterized Systems

Marius Bozga and Radu Iosif

Univ. Grenoble Alpes, CNRS, Grenoble INP[⋆⋆], Verimag

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 (mutual exclusion, absence of deadlocks). 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.

1 Introduction

A fundamental principle in the design of a distributed system is the separation between *coordination* and *behavior* [19]: the description of the coordinating architecture of a software system states the components it is made of and how they interact, whereas the components define the behavior they encapsulate and specify which part of this behavior is visible in the interface. The architecture then defines the interactions between the interfaces of the components, ignoring the internal aspects of their behavior.

Coordination is either *endogenous*, i.e. making explicit use of synchronization primitives in the code describing the behavior of the components (e.g. semaphores, monitors, barriers, etc.) or *exogenous*, i.e. having global rules describing how the components interact. A commonly perceived advantage of endogenous coordination is that programmers do not have to explicitly build a global coordination model. On the downside, endogenous coordination does not cope well with formal aspects of concurrent/distributed system design, for instance verification, because having a precise description of the structure of interactions is typically needed in order to automatically verify a parameterized system, in which the number of replicated components is finite but the upper bound is not known. More generally, exogenous coordination is a key enabler of the study of coordination mechanisms and their properties, as attested by the development of over a hundred architecture description languages [7,22].

Existing work on verification of parametric distributed systems typically assumes hard-coded architectures, whose structure (but not size) is fixed. For instance, the seminal work of German and Sistla [12] considers cliques, in which every component can interact with every other component, whereas Emerson and Namjoshi [11] and Browne, Clarke and Grumberg [8] consider token-ring architectures, in which each component

[⋆⋆] Institute of Engineering Univ. Grenoble Alpes interacts with its left and right neighbours only. Most early results focus on the decidability and computational complexity of verification problems such as safety (absence of error configurations), depending tightly on the shape of the coordinating architecture [3]. Because decidability can only be obtained at the price of drastic restrictions of the architectural pattern and of the communication model (usually rendez-vous with a bounded number of participants), more recent works go beyond the theoretical aspects and propose practical semi-algorithmic methods, such as *regular model checking* [15,1] or *automata learning* [9]. In such cases the architectural pattern is implicitly determined by the class of language recognizers: word automata encode pipelines or token-rings, whereas tree automata are used to describe hierarchical tree-structured architectures.

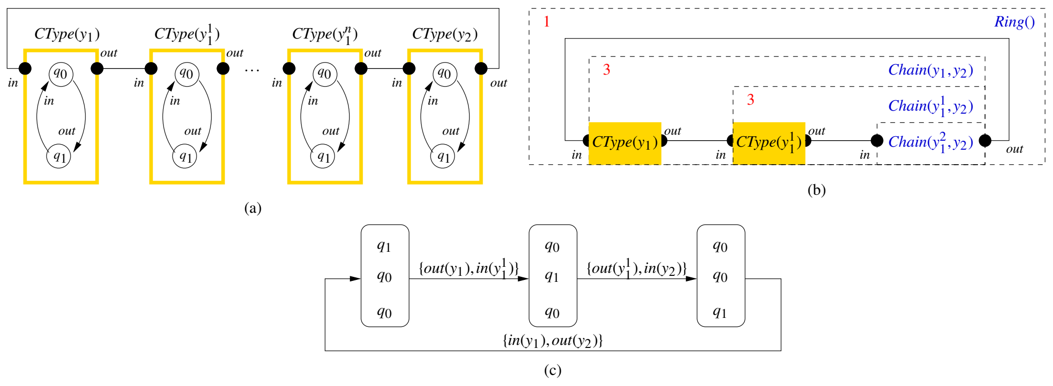
Among the first attempts at specifying architectures by logic is the *interaction logic* of Konnov et al. [18], which is a combination of Presburger arithmetic with monadic uninterpreted function symbols (denoting communication ports), that can describe cliques, stars and token-rings. They use first order logic without successor functions, thus limiting the expressivity of the language and excluding the possibility of describing more structured architectures, such as pipelines, token-rings and tree-structured hierachies. Such architectures can be described by an (undecidable) second-order extension of the interaction logic [21]. Our previous work on verifying safety properties of architectures described using interaction logic(s) considers interpreted successor functions that determine the shape of the architecture: zero successors describe cliques [6], one successor describe linear (pipeline, token-ring) or star architectures (a single controller with many slaves), whereas two or more successor functions describe tree-like architectures [4].

In this paper, we adhere to the exogenous coordination paradigm and define a language for describing the architectures that coordinate the interactions in a distributed system, 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 à priori and (ii) the shape of the structure in which the interactions take place, e.g. a pipeline, ring, star, tree or, more general hypergraph-shaped structures. We use a very simple syntax to describe the interactions between a component and its immediate neighbours, together with a set of inductive definitions that describe unbounded architectures, which follow a common recursive pattern. The motivation behind using inductive definitions is that recursive data structures, such as algebraic datatypes [2] or memory shapes [23] are ubiquitous in programming, hence programmers used to writing inductive specifications of data structures could easily learn to write inductive specifications of distributed component-based systems.

Specifying parameterized component-based systems by inductive definitions is not new. *Network grammars* [24] use context-free grammar rules to describe distributed systems with linear (pipeline, token-ring) architectures obtained by composition of an unbounded number of concurrent processes. Instead, we use predicate symbols of unrestricted arities to describe architectural patterns that are, in general, more complex than trees. Verification of network grammars against safety properties requires the synthesis of *network invariants* [25]. Such network invariants can be computed by rather costly fixpoint iterations [20] or by abstracting the composition of a small bounded number of instances [16]. Instead, our method uses lightweight *structural invariants*, that are shown to be easily inferred and efficient in many practical examples [4].

For starters, let us consider the following specification of a system, consising of components of type *CType* with two interaction ports, namely *in* and *out* and the behav-

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



ior 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). We specify this architecture by means of a predicate $Ring()$ defined inductively by the following rules:

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

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

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

Rule (2) says that the smallest chain consists of two instances of type *CType*, namely $CType(x_1)$ and $CType(x_2)$, such that the *out* port of x_1 connects to the *in* port of x_2, described as $out(x_1) \cdot in(x_2)$, where x_1 and x_2 are the formal arguments of the rule. Rule (3) describes the inductive step, namely that every chain stretching from x_1 to x_2 consists of a component $CType(x_1)$ that interacts with a disjoint chain from y_1 to x_2, where y_1 is an identifier different from every other identifier in the system. Finally, rule (1) closes the ring by connecting the *out* port of the last component y_2 to the *in* port of the first component y_1, written as $out(y_2) \cdot in(y_1)$. We refer to Fig. 1 for an illustration of the unfoldings of this set of recursive definitions. Any system such as the one 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 in red. Each application of rule (2) creates a fresh variable, denoted here as y_1^1, y_1^2, etc.

Having defined a language for specification of architectures, equipped with a formal semantics that describes an architecture as an abstract operator on finite-state behaviors, we move on to the *parametric safety problem*, which is checking that the behavior of every distributed system generated by an unfolding of a set of inductive definitions stays clear of a set of unsafe configurations. For instance, the behavior generated by the composition of three instances of type *CType* is depicted in Fig. 1c and the safety property we check for is that in each state there is at least one enabled transition.

Our method for proving safety relies on automatic invariant synthesis. Like in our previous work [6,4], we use structural invariants that can be derived directly from the behavioral term and the recursive rewriting rules describing the system. The verification method uses the invariant inference procedure to generate a $\mathsf{WS}\kappa\mathsf{S}$ formula that is unsatisfiable only if every system described by the given inductive definitions is safe. Since $\mathsf{WS}\kappa\mathsf{S}$ is a decidable fragment of monadic second-order logic, we use existing tools, such as MONA [13] for proving (parametric) safety. We have implemented the invariant synthesis in a prototype tool and experimented our method on 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(out) a main controller (star), etc. For space reasons, the proofs of the technical results are given in [5].

2 Behaviors and Architectures

This section introduces the preliminary definitions of a (finite-state) behavior and a bounded architecture, before defining behavioral types, that are the first ingredient of a formal definition of parametric component-based systems. Given sets A and B, we denote by $A \mapsto B$ the set of total functions from A into B. Partial mappings from A to B are denoted as $f : A \rightharpoonup B$, where $\mathrm{dom}(f) \stackrel{\mathsf{def}}{=} \{a \in A \mid f(a) \text{ is defined}\}$ is the domain and $\mathrm{rng}(f) \stackrel{\mathsf{def}}{=} \{f(a) \mid a \in \mathrm{dom}(f)\}$ is the range of f.

Let $\mathbb{P} = \{a, b, \ldots\}$ and $\mathbb{S} = \{s, t, \ldots\}$ be countably infinite sets of *ports* and *states*, respectively. A *configuration* $\sigma \subseteq \mathbb{S}$ is a finite set of states. A *behavior* is a tuple $\mathsf{B} = \langle \mathsf{P}, \mathsf{S}, \iota, \rightarrow \rangle$, where $\mathsf{P} \subseteq \mathbb{P}$ and $\mathsf{S} \subseteq \mathbb{S}$ are finite sets of ports and states, respectively, $\iota \subseteq \mathsf{S}$ denotes the initial configuration and $\rightarrow \subseteq 2^{\mathsf{S}} \times 2^{\mathsf{P}} \times 2^{\mathsf{S}}$ is a set of *transitions* denoted as $\sigma \xrightarrow{\pi} \tau$, for some configurations $\sigma, \tau \subseteq \mathsf{S}$ and some set of ports $\pi \subseteq \mathsf{P}$. We assume the existence of an *idling* transition $\sigma \xrightarrow{\emptyset} \sigma$, for each configuration $\sigma \subseteq \mathsf{S}$ and denote by P_B, S_B, ι_B and $\rightarrow_\mathsf{B}$ the ports, states, initial configuration and transitions of B, respectively. An *execution path* of B is a sequence of transitions $\sigma_1 \xrightarrow{\pi_1}_\mathsf{B} \sigma_2 \xrightarrow{\pi_2}_\mathsf{B} \ldots$ A configuration $\sigma \subseteq \mathsf{S}$ is *reachable* in B iff B has a finite execution path starting with ι and leading to σ; B is *safe* w.r.t. a set of configurations E iff no configuration from E is reachable in B.

Given two behaviors $\mathsf{B}_i = \langle \mathsf{P}_i, \mathsf{S}_i, \iota_i, \rightarrow_i \rangle$, for $i = 1, 2$, such that $\mathsf{S}_1 \cap \mathsf{S}_2 = \emptyset$ and $\mathsf{P}_1 \cap \mathsf{P}_2 = \emptyset$, we define their *product* as $\mathsf{B}_1 \parallel \mathsf{B}_2 \stackrel{\mathsf{def}}{=} \langle \mathsf{P}_1 \cup \mathsf{P}_2, \mathsf{S}_1 \cup \mathsf{S}_2, \iota_1 \cup \iota_2, \rightarrow_{\mathsf{B}_1 \parallel \mathsf{B}_2} \rangle$, where $\rightarrow_{\mathsf{B}_1 \parallel \mathsf{B}_2}$ is the smallest set of transitions defined by the rule (4). Intuitively, the product of two behaviors consists of any transition that belongs to either one of the two behaviors or a combined transition using the ports of both transitions in a joint action[1]. Since $\parallel$ is commutative and associative, we write $\mathsf{B}_1 \parallel \ldots \parallel \mathsf{B}_n$ instead of $(\mathsf{B}_1 \parallel \mathsf{B}_2) \parallel \ldots \parallel \mathsf{B}_n$.

$$\frac{\sigma_i \xrightarrow{\pi_i}_i \tau_i,\ i = 1,2}{(\sigma_1 \cup \sigma_2) \xrightarrow{\pi_1 \cup \pi_2}_{\mathsf{B}_1 \parallel \mathsf{B}_2} (\tau_1 \cup \tau_2)} \quad (4) \qquad \frac{\sigma \xrightarrow{\pi}_{\mathsf{B}_1 \parallel \ldots \parallel \mathsf{B}_n} \tau,\ \pi \in \gamma}{\sigma \xrightarrow{\pi}_{\gamma(\mathsf{B}_1, \ldots, \mathsf{B}_n)} \tau} \quad (5)$$

The product of behaviors (4) is, in general, too permissive and allows unsafe executions. We refine this operator to achieve a desired level of safety, by means of *architectures*, a central notion in the rest of this paper, defined below:

Definition 1. *An* interaction $\pi \subseteq \mathbb{P}$ *is a finite set of ports. An* architecture $\gamma \subseteq 2^{\mathbb{P}}$ *is a finite set of interactions.*

[1] In particular, each transition $\sigma_1 \xrightarrow{\pi_1}_1 \tau_1$ induces a transition $(\sigma_1 \cup \sigma_2) \xrightarrow{\pi_1} (\tau_1 \cup \sigma_2)$ due to the idling transition $\sigma_2 \xrightarrow{\emptyset} \sigma_2$.

Just as the product of behaviors (4), an architecture can be viewed as a commutative and associative operator, whose application to the set of behaviors $\{\mathsf{B}_i = \langle \mathsf{P}_i, \mathsf{S}_i, \iota_i, \rightarrow_i \rangle\}_{i=1}^n$ is the behavior $\gamma(\mathsf{B}_1, \ldots, \mathsf{B}_n) \overset{\text{def}}{=} \langle \bigcup_{i=1} \mathsf{P}_i, \bigcup_{i=1}^n \mathsf{S}_n, \bigcup_{i=1}^n \iota_i, \rightarrow_{\gamma(\mathsf{B}_1,\ldots,\mathsf{B}_n)} \rangle$, where $\rightarrow_{\gamma(\mathsf{B}_1,\ldots,\mathsf{B}_n)}$ is the least set of transitions defined by the rule (5). The architecture γ simply restricts the transitions of the product $\mathsf{B}_1 \parallel \ldots \parallel \mathsf{B}_n$ to the ones labeled with an interaction from γ. Note that the arity of γ is not fixed, i.e. $\gamma(\mathsf{B}_1, \ldots, \mathsf{B}_n)$ is defined, for all $n \geq 1$.

In the rest of this paper, we are concerned with systems consisting of an unbounded number of replicated behaviors, that belong to a fairly small number of patterns, called component types. Let $\mathbb{I} = \{i, j, \ldots\}$ be a countably infinite set of *identifiers*. A *component type* is a tuple $\mathcal{B} = \langle \mathcal{P}_\mathcal{B}, \mathcal{S}_\mathcal{B}, \mathcal{I}_\mathcal{B}, \Delta_\mathcal{B} \rangle$, where $\mathcal{P}_\mathcal{B} \subseteq \mathbb{I} \mapsto \mathbb{P}$ and $\mathcal{S}_\mathcal{B} \subseteq \mathbb{I} \mapsto \mathbb{S}$ are finite sets of total functions mapping identifiers to ports and states, respectively, $\mathcal{I}_\mathcal{B} \in \mathcal{S}$ denotes initial states and $\Delta_\mathcal{B} \subseteq (\mathbb{I} \mapsto \mathcal{S}) \times (\mathbb{I} \mapsto \mathbb{P}) \times (\mathbb{I} \mapsto \mathcal{S})$ is a finite set of *transition rules* of the form $S \xrightarrow{P} T$. In addition, we require that, for any $P, Q \in \mathcal{P}$ $[S, T \in \mathcal{S}]$ and $i, j \in \mathbb{I}$, such that $P(i) = Q(j)$ $[S(i) = T(j)]$, we have $P = Q$ $[S = T]$ and $i = j$, i.e. all elements of $\mathcal{P}_\mathcal{B}$ $[\mathcal{S}_\mathcal{B}]$ are injective functions with pairwise disjoint ranges.

Given a component type $\mathcal{B} = \langle \mathcal{P}, \mathcal{S}, \mathcal{I}, \Delta \rangle$ and an identifier $i \in \mathbb{I}$, the behavior $\mathcal{B}(i) \overset{\text{def}}{=} \langle \{P(i) \mid P \in \mathcal{P}\}, \{S(i) \mid S \in \mathcal{S}\}, \{\mathcal{I}(i)\}, \{\{S(i)\} \xrightarrow{\{P(i)\}} \{T(i)\} \mid S \xrightarrow{P} T \in \Delta\} \rangle$ is called the i-th *instance* of $\mathcal{B}$. Note that $\mathsf{P}_{\mathcal{B}(i)} \cap \mathsf{P}_{\mathcal{B}(j)} = \emptyset$ and $\mathsf{S}_{\mathcal{B}(i)} \cap \mathsf{S}_{\mathcal{B}(j)} = \emptyset$, for any $i \neq j \in \mathbb{I}$.

In the rest of this paper, we consider a fixed set $\mathbb{B}$ of component types, such that $\mathcal{P}_{\mathcal{B}_1} \cap \mathcal{P}_{\mathcal{B}_2} = \emptyset$ and $\mathcal{S}_{\mathcal{B}_1} \cap \mathcal{S}_{\mathcal{B}_2} = \emptyset$, for any $\mathcal{B}_1, \mathcal{B}_2 \in \mathbb{B}$.

3 A Term Algebra of Behaviors

In this section we introduce a recursive term algebra for describing the behaviors resulting from the composition of an unbounded number of component type instances. Let $\mathbb{V}_1$ be a countably infinite set of first-order variables and $\mathbb{A}$ be a countably infinite set of *predicates*, where $\#(\mathsf{A}) \geq 0$ denotes the arity of $\mathsf{A} \in \mathbb{A}$. The following syntax generates *behavioral terms* inductively, starting with the b non-terminal:

$$P \in \mathcal{P},\ x \in \mathbb{V}_1,\ i \in \mathbb{I},\ \mathcal{B} \in \mathbb{B},\ \mathsf{A} \in \mathbb{A}$$

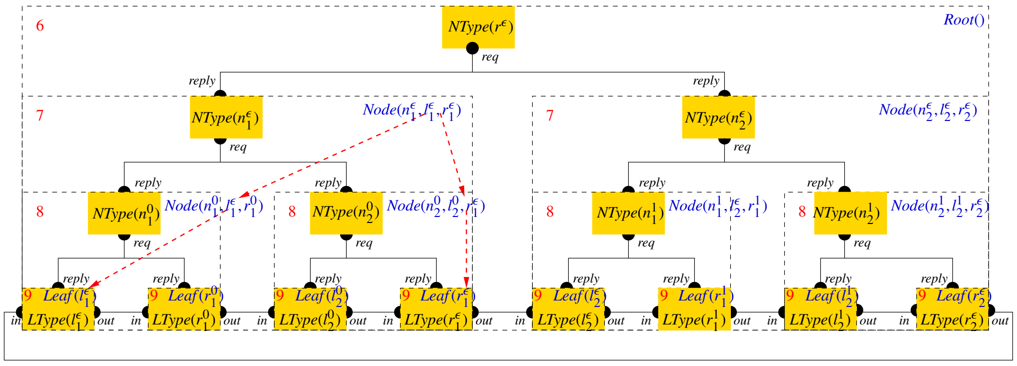
$$\xi ::= x \mid i \qquad \Gamma ::= P(\xi) \mid \Gamma_1 \cdot \Gamma_2 \mid \Gamma_1 + \Gamma_2 \qquad \text{architecture specifications}$$

$$\mathsf{b} ::= \mathcal{B}(\xi) \mid \langle \Gamma \rangle(\mathsf{b}_1, \ldots, \mathsf{b}_n) \mid \nu x \,.\, \mathsf{b}_1 \mid \mathsf{A}(\xi_1, \ldots, \xi_{\#(\mathsf{A})}) \quad \text{behavioral terms}$$

A variable x occurring in a behavioral term b is said to be *free* if it does not occur in the scope of some subterm of the form $\nu x \,.\, \mathsf{b}_1$ and *bound* otherwise. In the following, we assume that all bound variables occurring in a term are pairwise distinct and distinct from the free variables. Note that this assumption loses no generality because terms obtained by α-conversion (renaming of bound variables) are assumed to be equivalent. A term b is said to be *closed* if $\mathrm{fv}(\mathsf{b}) = \emptyset$, *predicate-less* if no predicates from $\mathbb{A}$ occur in b and *ground* if no variable, either free or bound, occurs in b. A term $\mathcal{B}(\xi)$ is called an *instance atom* and a term $\mathsf{A}(\xi_1, \ldots, \xi_n)$ is called a *predicate atom*. We denote by $\mathsf{inst}(\mathsf{b})$ the set of instance atoms of b, by $\#_{\mathsf{pred}}(\mathsf{b})$ the number of occurrences of predicate atoms and by $\mathsf{pred}_j(\mathsf{b})$, $j \in [0, \#_{\mathsf{pred}}(\mathsf{b}) - 1]$, the predicate atom that occurs j-th in b, in some predefined order of the syntax tree nodes of b.

A symbol $\xi \in \mathbb{V}_1 \cup \mathbb{I}$ is *instantiated* in a behavioral term b if $\mathcal{B}(\xi)$ is a subterm of b, for some component type $\mathcal{B}$, and we denote by $\mathrm{inst}(\mathsf{b})$ the set of symbols instantiated in b. Note that a symbol (variable or identifier) may occur in a term with-

Fig. 2: Tree Architecture with Leaves Linked in a Token-Ring



out being instantiated. For example, both identifiers i and j occur within the term $\langle out(i)\cdot in(j)\rangle(CType(j))$, but only j is instantiated by the atom $CType(j)$. A behavioral term b is *well-instantiated* if every identifier occurring in b is instantiated at most once. For example, the following term is not well-instantiated, because i is instantiated twice in $\langle out(i)\cdot in(j)\rangle(CType(i), \langle in(i)\cdot out(j)\rangle(CType(j), CType(i)))$.

A *substitution* is a partial function $\eta : \mathbb{V}_1 \rightharpoonup \mathbb{V}_1 \cup \mathbb{I}$ mapping variables into either variables or identifiers. A substitution η is *ground* if $\mathrm{rng}(\eta) \subseteq \mathbb{I}$. We denote by $[\xi_1/x_1, \ldots, \xi_n/x_n]$ the substitution mapping each $x_i \in \mathbb{V}_1$ into $\xi_i \in \mathbb{V}_1 \cup \mathbb{I}$, for all $i \in [1,n]$, and undefined everywhere else. The application of a substitution η to a behavioral term b is the term $\mathsf{b}\eta$ in which every free occurrence of a variable $x \in \mathrm{fv}(\mathsf{b}) \cap \mathrm{dom}(\eta)$ has been replaced by $\eta(x)$. Note that substitutions only apply to the free variables of the term.

Given a predicate-less behavioral term b and a ground substitution η, such that $\mathrm{fv}(\mathsf{b}) \subseteq \mathrm{dom}(\eta)$, the *ground set* of b is the set $[\mathsf{b}]_\eta$ of ground terms, defined inductively:

$$[\mathcal{B}(x)]_\eta \stackrel{\mathsf{def}}{=} \{\mathcal{B}(\eta(x))\} \qquad [\mathcal{B}(i)]_\eta \stackrel{\mathsf{def}}{=} \{\mathcal{B}(i)\} \qquad [\nu x \,.\, \mathsf{b}_1]_\eta \stackrel{\mathsf{def}}{=} \bigcup_{i\in\mathbb{I}\setminus\mathrm{rng}(\eta)} [\mathsf{b}_1]_{\eta[x\leftarrow i]}$$

$$[\langle\Gamma\rangle(\mathsf{b}_1,\ldots,\mathsf{b}_n)]_\eta \stackrel{\mathsf{def}}{=} \bigcup\{\langle\Gamma\rangle(\mathsf{t}_1,\ldots,\mathsf{t}_n)\eta \mid \forall 1 \le k < \ell \le n \,.\, \mathsf{t}_k \in [\mathsf{b}_k]_\eta \wedge \mathrm{inst}(\mathsf{t}_k) \cap \mathrm{inst}(\mathsf{t}_\ell) = \emptyset\}$$

Whenever b is closed, η can be omitted and its ground set can be written $[\mathsf{b}]$. Note that the definition of the ground set prevents multiple instantiation of the same identifier.

The meaning of a ground architecture specification Γ, built from ports $P(i)$, $P \in \mathcal{P}$, $i \in \mathbb{I}$, using the constructors $+$ and $\cdot$, is the architecture $[\![\Gamma]\!] \subseteq 2^{\mathcal{P}}$, defined inductively:

$$[\![P(i)]\!] \stackrel{\mathsf{def}}{=} \{\{P(i)\}\} \quad [\![\Gamma_1 + \Gamma_2]\!] \stackrel{\mathsf{def}}{=} [\![\Gamma_1]\!] \cup [\![\Gamma_2]\!] \quad [\![\Gamma_1 \cdot \Gamma_2]\!] \stackrel{\mathsf{def}}{=} \{I_1 \cup I_2 \mid I_i \in [\![\Gamma_i]\!],\ i = 1,2\}$$

Note that the $+$ and $\cdot$ constructors are both commutative and associative. Moreover, $\cdot$ distributes over $+$, thus each ground architecture specification can be equivalently written $\Gamma = \sum_{k=1}^{m} \prod_{\ell=1}^{r_k} P_{k\ell}(i_{k\ell})$, where $P_{k\ell} \in \mathcal{P}$ and $i_{k\ell} \in \mathbb{I}$, for all $k \in [1,m]$ and $\ell \in [1,r_k]$.

We extend ground sets from predicate-less terms to terms with predicate occurrences, by recursively replacing predicate subterms by terms given by a set of rewriting rules (called a *rewriting system*) of the form $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \mathsf{b}$, where b is a behavioral term, such that $\mathrm{fv}(\mathsf{b}) \subseteq \{x_1,\ldots,x_{\#(\mathsf{A})}\}$. For conciseness, we write $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow_{\mathcal{R}} \mathsf{b}$ instead of $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \mathsf{b} \in \mathcal{R}$.

Example 1. The following example describes, by the term $Root()$, a tree architecture in which parents communicate with their children and, in addition, all nodes on the frontier communicate via a token-ring. The inner nodes in the tree have component type $NType$, with associated ports *req* and *reply*, whereas the leaves have type *LType*, with associated ports *reply*, *in* and *out*.

$$\begin{aligned}
Root() &\leftarrow \nu r\ \nu n_1\ \nu l_1\ \nu r_1\ \nu n_2\ \nu l_2\ \nu r_2\ . \\
&\quad \langle req(r)\cdot reply(n_1)\cdot reply(n_2) + out(r_1)\cdot in(l_2) + out(r_2)\cdot in(l_1)\rangle \\
&\quad (Ntype(r), Node(n_1,l_1,r_1), Node(n_2,l_2,r_2)) && (6)\\
Node(n,l,r) &\leftarrow \nu n_1\ \nu r_1\ \nu n_2\ \nu l_2\ .\ \langle req(n)\cdot reply(n_1)\cdot reply(n_2) + out(r_1)\cdot in(l_2)\rangle \\
&\quad (NType(n), Node(n_1,l,r_1), Node(n_2,l_2,r)) && (7)\\
Node(n,l,r) &\leftarrow \langle req(n)\cdot reply(l)\cdot reply(r) + out(l)\cdot in(r)\rangle \\
&\quad (NType(n), Leaf(l), Leaf(r)) && (8)\\
Leaf(n) &\leftarrow LType(n) && (9)
\end{aligned}$$

We refer to Fig. 2 for a depiction of the unfolding of the above rewriting rules and of the resulting architecture. The labels of the rewriting rules applied at each rewriting step are marked in red. For readability, we superscript each bound variable introduced by a rule with the node of the rewriting tree where this rule was applied. Each rule (6-8) creates an interaction between the parent node (*req*) and its children (*reply*) and the leaf rules (9) also creates interactions between siblings of the form $\{out(i), in(j)\}$. In addition, the initial rule (6) closes the ring of leaves, via the interactions $\{out(r_2^\epsilon), in(l_1^\epsilon)\}$ and $\{out(r_1^\epsilon), in(l_2^\epsilon)\}$, where the parameters l_1^ϵ and r_2^ϵ are instantiated in the left- and right-most leaves and r_1^ϵ (l_2^ϵ) in the right-most (left-most) leaf of the left (right) subtree. ■

For technical convenience, we place the steps of an rewriting sequence in a tree, whose nodes are labeled by rewriting rules. Formally, a *tree* $\mathcal{T}$ is defined by a set $\mathrm{nodes}(\mathcal{T})$ and a function mapping each node $w \in \mathrm{nodes}(\mathcal{T})$ to its *label*, denoted by $\mathcal{T}(w)$. The set $\mathrm{nodes}(\mathcal{T})$ is a finite subset of $\mathbb{N}^*$, where $\mathbb{N}^*$ is the set of finite sequences of non-negative integers, such that $wi \in \mathrm{nodes}(\mathcal{T})$ for some $i \in \mathbb{N} \setminus \{0\}$ only if $w \in \mathrm{nodes}(\mathcal{T})$ and $wj \in \mathrm{nodes}(\mathcal{T})$ for all $j \in [0, i-1]$. The *root* of $\mathcal{T}$ is the empty sequence ϵ, the *children* of a node $w \in \mathrm{nodes}(\mathcal{T})$ are the nodes $wi \in \mathrm{nodes}(\mathcal{T})$, where $i \in \mathbb{N}$, and the *parent* of a node wi with $i \in \mathbb{N}$ is w (the root ϵ has no parent). The *leaves* of $\mathcal{T}$ are the nodes in $\mathrm{leaves}(\mathcal{T}) \stackrel{\mathsf{def}}{=} \{w \in \mathrm{nodes}(\mathcal{T}) \mid w.0 \notin \mathrm{nodes}(\mathcal{T})\}$. The *subtree* of $\mathcal{T}$ rooted at w is defined as $\mathcal{T}\!\downarrow_w (w') \stackrel{\mathsf{def}}{=} \mathcal{T}(ww')$, for all $w' \in \mathrm{nodes}(\mathcal{T}\!\downarrow_w) \stackrel{\mathsf{def}}{=} \{w' \mid ww' \in \mathrm{nodes}(\mathcal{T})\}$.

Definition 2. *Given a rewriting system $\mathcal{R}$ and a closed behavioral term b, a* rewriting tree *for b is a tree $\mathcal{T}$ such that $\mathcal{T}(\epsilon) = (\mathsf{A_b}() \leftarrow \mathsf{b})$, where $\mathsf{A_b}$ is a predicate symbol of zero arity, that does not occur in $\mathcal{R}$ and, for all $w \in \mathrm{dom}(\mathcal{T})$, such that $\mathcal{T}(w) = (\mathsf{A}_w(x_1, \ldots, x_{\#(\mathsf{A}_w)}) \leftarrow_{\mathcal{R}} \mathsf{b}_w)$:*

1. *for all $i \in [0, \#_{\mathsf{pred}}(\mathsf{b}_w) - 1]$, 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 $\mathcal{T}(wi) = \mathsf{A}_{wi}(x_1, \ldots, x_{\#(\mathsf{A}_{wi})}) \leftarrow_{\mathcal{R}} \mathsf{b}_{wi}$, for some behavioral term b_{wi},*
2. *for all $i \geq \#_{\mathsf{pred}}(\mathsf{b}_w)$, we have $wi \notin \mathrm{nodes}(\mathcal{T})$.*

We denote $\mathcal{R}_{\mathsf{b}} \stackrel{\mathsf{def}}{=} \mathcal{R} \cup \{\mathsf{A_b}() \leftarrow \mathsf{b}\}$ and by $\mathbb{T}_{\mathcal{R}}(\mathsf{b})$ the set of rewriting trees for b in $\mathcal{R}$.

Note the addition of a fresh rule $\mathsf{A_b}() \leftarrow \mathsf{b}$ to $\mathcal{R}$, that is required for a uniform labeling of the tree with rules. For instance, Fig. 2 shows a balanced binary rewriting tree, whose root is labeled by rule (6), second and third level nodes are labeled by rules (7) and (8) respectively, and leaves are labeled by rule (9). A rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$ corresponds to a predicate-less *characteristic term* $\mathfrak{C}(\mathcal{T})$ defined inductively on the structure of $\mathcal{T}$:

Definition 3. *Given a behavioral term* b *and a rewriting tree* $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$*, for each* $w \in \mathrm{nodes}(\mathcal{T})$*, such that* $\mathcal{T}(w) = (\mathsf{A}_w(x_1,\ldots,x_{\#(\mathsf{A}_w)}) \leftarrow_{\mathcal{R}} \mathsf{b}_w)$*, we define* $\mathfrak{C}(\mathcal{T}\downarrow_w)$ *as the term obtained by replacing each predicate atom* $\mathsf{pred}_i(\mathsf{b}) = \mathsf{A}_{wi}(y_1,\ldots,y_{\#(\mathsf{A}_{wi})})$ *by the term* $\mathfrak{C}(\mathcal{T}\downarrow_{wi})[y_1/x_1,\ldots,y_{\#(\mathsf{A}_i)}/x_{\#(\mathsf{A}_i)}]$*, for all* $i \in [0,\#_{\mathsf{pred}}(\mathsf{b}_w)-1]$*. We define* $\mathfrak{C}(\mathcal{T}) \overset{\mathsf{def}}{=} \mathfrak{C}(\mathcal{T}\downarrow_\epsilon)$ *and* $\mathfrak{C}_{\mathcal{R}}(\mathsf{b}) \overset{\mathsf{def}}{=} \{\mathfrak{C}(\mathcal{T}) \mid \mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})\}$ *the set of predicate-less terms generated by* $\mathcal{R}$ *from* b*.*

Intuitively, the characteristic term of a rewriting tree is the predicate-less term obtained by replacing all predicate instances by the bodies of their corresponding rewriting rules from the tree. We define the ground set of behavioral term (possibly with predicate atoms) as $[\mathsf{b}]_{\eta,\mathcal{R}} \overset{\mathsf{def}}{=} \bigcup_{\mathsf{t}\in\mathfrak{C}_{\mathcal{R}}(\mathsf{b})} [\mathsf{t}]_\eta$ and avoid mentioning η when b is closed.

Next, we define a semantic operator $[\![.]\!]$ that maps ground behavioral terms to behaviors. The idea is that nested terms, such as e.g. $\langle out(x)\cdot in(y)\rangle(CType(x), \langle in(x)\cdot out(y)\rangle(CType(y)))$ are not be dealt with compositionally; instead, all the (partial) architectures that occur within subterms of a behavioral term are first joined into a top-level architecture that applies, at the same time, to all instances in the term. Formally, we define the following *flattening* relation on behavioral terms:

$$\langle \Gamma_1\rangle(\langle\Gamma_2\rangle(\mathsf{b}_1,\ldots\mathsf{b}_i),\mathsf{b}_{i+1},\ldots\mathsf{b}_n) \rightsquigarrow \langle\Gamma_1+\Gamma_2\rangle(\mathsf{b}_1,\ldots,\mathsf{b}_n) \quad (10)$$

Note that the order of the arguments $\langle\Gamma_2\rangle(\mathsf{b}_1,\ldots\mathsf{b}_i),\mathsf{b}_{i+1},\ldots,\mathsf{b}_n$ of Γ_1 is not important. It is easy to see that every chain $\mathsf{t}_1 \rightsquigarrow \mathsf{t}_2 \rightsquigarrow \ldots$ is finite, because the height of terms strictly decreases with flattening. Moreover, for each behavioral term b, the endpoint of any such chain starting with b is unique (modulo commutativity and associativity of the $\cdot$ and $+$ architecture constructors) and is denoted by $\mathsf{b}^{\rightsquigarrow}$.

We are now in position to define the semantics of a behavioral term b, as a (possibly infinite) set of behaviors. Let η be a ground substitution, such that $\mathrm{fv}(\mathsf{b}) \subseteq \mathrm{dom}(\eta)$, and $\mathcal{R}$ be a rewriting system. First, we define the semantics of a well-instantiated ground term in canonical form $\mathsf{t}^{\rightsquigarrow} = \langle\Gamma\rangle(\mathsf{t}_1,\ldots,\mathsf{t}_n)$, from the ground set of b, namely $\mathsf{t} \in [\mathsf{b}]_{\eta,\mathcal{R}}$. Because the flattenning relation is applied exhaustively to t, it must be the case that $\mathsf{t}_k = \mathcal{B}_k(i_k)$, where $i_k \in \mathbb{I}$, for all $k \in [1,n]$. Then $[\![\mathsf{t}^{\rightsquigarrow}]\!]$ is the behavior $[\![\Gamma]\!](\mathcal{B}_1(i_1),\ldots,\mathcal{B}_n(i_n))$, defined by (5). The semantics is lifted from ground terms to arbitrary behavioral terms:

$$[\![\mathsf{b}]\!]_{\eta,\mathcal{R}} \overset{\mathsf{def}}{=} \bigcup_{\mathsf{t}\in[\mathsf{b}]_{\eta,\mathcal{R}}} [\![\mathsf{t}^{\rightsquigarrow}]\!] \quad (11)$$

We omit writing η when b is closed. For example, Fig. 1d shows the behavior obtained by the following sequence alternating rewriting and flattening steps:

$$Ring() \overset{(1)}{\leftarrow} \nu y_1 \nu y_2\ .\ \langle out(y_2)\cdot in(y_1)\rangle(Chain(y_1,y_2)) \overset{(2)}{\leftarrow}$$
$$\nu y_1 \nu y_2 \nu y_1^1\ .\ \langle out(y_2)\cdot in(y_1) + out(y_1)\cdot in(y_1^1)\rangle(CType(y_1), Chain(y_1^1,y_2)) \overset{(3)}{\leftarrow}$$
$$\nu y_1 \nu y_2 \nu y_1^1\ .\ \langle out(y_2)\cdot in(y_1) + out(y_1)\cdot in(y_1^1) + out(y_1^1)\cdot in(y_2)\rangle(CType(y_1), CType(y_1^1), CType(y_2)).$$

4 The Parametric Safety Problem

Having defined a language for specification of architectures, we move on to the problem of verifying that every behavior generated by a rewriting system, starting with a given behavioral term, is safe with respect to a set of error configurations. This problem is challenging, because we ask for a proof of safety that holds *for every* ground instantiation of some predicate-less rewriting of the behavioral term.

Intuitively, a set of behaviors is said to be *parametric* if each behavior in the set is obtained from the same pattern, by assigning different values to several designated variables, called *parameters*. Formally, a *parametric system* is a tuple $\mathcal{C} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_K, \mathcal{A} \rangle$, where $\mathcal{B}_i \in \mathbb{B}$ are component types and $\mathcal{A}$ maps a tuple $\mathbf{T} = \langle T_1, \ldots, T_N \rangle$ of sets of identifiers $T_1, \ldots, T_N \subseteq \mathbb{I}$, to an architecture, denoted as $\mathcal{A}(\mathbf{T})$. Intuitively, the tuple of sets $\mathbf{T}$ is a *structural parameter* of the system, that defines (i) the architecture which coordinates the instances of $\mathcal{B}_1, \ldots, \mathcal{B}_K$ and (ii) the set of instances belonging to each behavior type. For presentation purposes, we defer the precise definitions to §4.2. The behavior resulting from the application, using the composition rule (5), of the architecture $\mathcal{A}(\mathbf{T})$ to these instances is denoted as $\mathcal{C}(\mathbf{T})$.

The *parametric safety problem* asks whether each behavior $\mathcal{C}(\mathbf{T})$ of a parametric system $\mathcal{C}$ is safe w.r.t. a given set E of configurations. Since, in general, the parametric safety problem is undecidable, we resort to a sound but necessarily incomplete solution, that consists in computing *safety invariants*. Given a behavior B, an invariant I of B is a superset of the set of reachable configurations of B, thus B is safe w.r.t. E if $\mathsf{I} \cap \mathsf{E} = \emptyset$ (the reversed implication is clearly not true in general). Since we consider a parametric system, the challenge is computing a parametric safety invariant, i.e. a pattern that defines an invariant for each behavior $\mathcal{C}(\mathbf{T})$, determined by a choice of $\mathbf{T}$.

In contrast with the classical approach to invariant synthesis based on a fixpoint iteration in an abstract domain [10], we focus on a particular class of invariants that can be obtained directly from the description of the parametric system. These invariants are called *structural* in the following. The structural invariants considered in this paper are mostly inspired by the following notions:

Definition 4. *A* trap θ *of a behavior* $\mathsf{B} = \langle \mathsf{P}, \mathsf{S}, \iota, \rightarrow \rangle$ *is a subset of* S *such that, for any two configurations* σ *and* σ' *of* B, *such that* $\sigma \rightarrow_{\mathsf{B}} \sigma'$, *we have* $\sigma \cap \theta \neq \emptyset$ *only if* $\sigma' \cap \theta \neq \emptyset$. *A trap* θ *is* marked *iff* $\theta \cap \iota \neq \emptyset$. *The* trap invariant *of* B *is the set* $\Theta(\mathsf{B}) \stackrel{\text{def}}{=} \{\sigma \subseteq \mathsf{S} \mid \sigma \cap \theta \neq \emptyset$, *for each marked trap* θ *of* $\mathsf{B}\}$.

To understand why $\Theta(\mathsf{B})$ is an invariant of B, note that $\Theta(\mathsf{B})$ contains the initial configuration of B and is closed under the transition relation $\rightarrow_{\mathsf{B}}$. Since the set of reachable configurations of B is the smallest such set, it follows that $\Theta(\mathsf{B})$ is an over-approximation of the reachable configurations of B, hence an invariant.

4.1 The Weak Sequential Calculus of κ Successors

The structural invariants and the sets of unsafe configurations will be described using a restriction of monadic second order logic (MSO) to trees of branching κ, where $\kappa > 0$ is an integer constant. Let $\mathbb{V}_2 = \{X, Y, Z, \ldots\}$ be a countably infinite set of second order variables. The formulæ of WSκS are defined by the following syntax:

$$\begin{array}{ll} \tau ::= \overline{\epsilon} \mid x \in \mathbb{V}_1 \mid \mathrm{succ}_i(\tau_1),\ i \in [0, \kappa-1] & \text{terms} \\ \phi ::= \tau_1 = \tau_2 \mid X(\tau) \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists x \,.\, \phi_1 \mid \exists X \,.\, \phi_1 & \text{formulæ} \end{array}$$

As usual, we write $\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2 \stackrel{\text{def}}{=} \neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2 \stackrel{\text{def}}{=} \phi_1 \rightarrow \phi_2 \wedge \phi_2 \rightarrow \phi_1$, $\forall x \,.\, \phi \stackrel{\text{def}}{=} \neg\exists x \,.\, \neg\phi$ and $\forall X \,.\, \phi \stackrel{\text{def}}{=} \neg\exists X \,.\, \neg\phi$.

WSκS formulæ are interpreted over an infinite κ-ary tree with nodes $[0, \kappa-1]^*$, where first order variables $x \in \mathbb{V}_1$ range over individual nodes $n \in [0, \kappa-1]^*$, second order variables $X \in \mathbb{V}_2$ range over finite sets of nodes $T \subseteq [0, \kappa-1]^*$, $\overline{\epsilon}$ is a constant symbol interpreted as ϵ and, for all $i \in [0,\kappa-1]$, the function symbol succ_i is interpreted by the total function $n \mapsto ni$. Given a valuation $\nu : \mathbb{V}_1 \cup \mathbb{V}_2 \to [0,\kappa-1]^* \cup 2^{[0,\kappa-1]^*}$, such that $\nu(x) \in [0,\kappa-1]^*$, for each $x \in \mathbb{V}_1$ and $\nu(X) \subseteq [0,\kappa-1]^*$, for each $X \in \mathbb{V}_2$, the satisfaction relation $\nu \models \phi$ is defined inductively on the structure of the formula ϕ, as usual. A valuation ν is a *model* of a formula ϕ if and only if $\nu \models \phi$. A formula is *satisfiable* if and only if it has a model.

4.2 Parametric Systems Defined by Behavioral Terms

We define the parametric component-based system $C = \langle \mathcal{B}_1,\ldots,\mathcal{B}_K,\mathcal{A}\rangle$ corresponding to a given closed behavioral term b and a rewriting system $\mathcal{R}$. To ease the upcoming developments, we shall consider closed behavioral terms b and rewriting systems $\mathcal{R}$ that meet the following:

Assumption 1 *Each bound variable in* b *is instantiated exactly once in each predicate-less term* $\mathsf{t} \in \mathfrak{C}_{\mathcal{R}}(\mathcal{T})$*, for each* $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$*. Moreover, different variables are instantiated in different nodes of* $\mathcal{T}$*.*

We refer the interested reader to [5, Proposition 1] for a proof of the fact that Assumption 1 loses no generality. This allows us to identify the indices of instances with the nodes of a rewriting tree (Definition 2), in order to describe parametric invariants using WSκS. More precisely, we identify the index of a component instantiated by an atom $\mathcal{B}(x)$ of b, with the unique node of the rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$ labeled by that atom. Note that, by Assumption 1, the index of the $\mathcal{B}(x)$ component is uniquely determined by $\mathcal{T}$. Consequently, in the rest of the paper, we shall silently identify $\mathbb{I}$ with $[0,\kappa-1]^*$.

In principle, by fixing a particular interpretation of indices in a ground term $\mathsf{t} \in [\mathsf{b}]_{\mathcal{R}}$, we also restrict the set of behaviors considered, i.e. we consider a strict subset of $[\![\mathsf{b}]\!]_{\mathcal{R}}$ (11). This particular restriction is, however, without consequences for the soundness of the verification method, because ground terms that differ only by a permutation of indices generate behaviors that are bisimilar and have the same safety properties (modulo a permutation of indices). We shall silently assume, from now on, that the set of unsafe configurations E from the specification of a parametric safety problem is closed under permutations of indices. This is the case when the WSκS definition of E does not use successor functions and only compares first order variables for equality (e.g. 15).

Let us consider that $\mathcal{R}_{\mathsf{b}} = \mathcal{R} \cup (\mathsf{A}_{\mathsf{b}}() \leftarrow \mathsf{b})$ consists of the rules $\mathsf{r}_1,\ldots,\mathsf{r}_N$, such that $\mathsf{r}_1 = (\mathsf{A}_{\mathsf{b}}() \leftarrow \mathsf{b})$. 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 the rewriting tree. Note that, with this convention, U_1 is a singleton containing the root of the rewriting tree (Definition 2). We say that a tuple of sets of identifiers $\mathbf{T} = \langle T_1,\ldots,T_N\rangle$ is *parameter-compatible with* $\mathcal{R}$ *and* b iff any valuation ν, such that $\nu(U_i) = T_i$, for all $i \in [1,N]$, is a model of the $\mathit{RTree}(\mathbf{U})$ formula (Fig. 3). Note that this formula is a WSκS encoding of the conditions from Definition 2. The above formulæ depend implicitly on $\mathcal{R}$ and b, which will be silently assumed in the following.

We are now in position to define the parametric system $C = \langle \mathcal{B}_1,\ldots,\mathcal{B}_K,\mathcal{A}\rangle$, corresponding to $\mathcal{R}$ and b. First, let $\mathcal{B}_1,\ldots,\mathcal{B}_K$ be the component types that occur in b and in the rules of $\mathcal{R}$. Second, we define $\mathcal{A}$ as a partial mapping of the sets $T_1,\ldots,T_N \subseteq \mathbb{I}$ to an architecture defined whenever $\mathbf{T} = \langle T_1,\ldots,T_N\rangle$ is parameter-compatible with $\mathcal{R}$ and b. Since, in this case, we have $[U_1 \leftarrow T_1,\ldots,U_N \leftarrow T_N] \models \mathit{RTree}(\mathbf{U})$, the sets $T_1,\ldots,T_N$

Fig. 3: Encoding Rewriting Trees, Instance Sets and Configurations in $\mathsf{WS}\kappa\mathsf{S}$

$$
\begin{aligned}
\mathit{RTree}(\mathbf{U}) &\stackrel{\text{def}}{=} \forall x \,.\, \textstyle\bigwedge_{1 \le i < j \le N} \Big(\neg U_i(x) \vee \neg U_j(x)\Big) \wedge U_1(x) \leftrightarrow x = \overline{\epsilon} \wedge \\
&\quad \forall x \,.\, \textstyle\bigwedge_{\mathsf{r}_i \in \mathcal{R}} \bigwedge_{\ell=0}^{\kappa-1} U_i(\mathrm{succ}_\ell(x)) \rightarrow \bigvee_{\mathsf{r}_j \in \mathcal{R}} U_j(x) \wedge \\
&\quad \forall x \,.\, \textstyle\bigwedge_{\mathsf{r}_i = \left(\mathsf{A}'(x_1,\ldots,x_{\#(\mathsf{A}')}) \leftarrow_{\mathcal{R}} \mathsf{b}'\right)} \bigwedge_{j=0}^{\#_{\mathsf{pred}}(\mathsf{b}')-1} U_i(x) \rightarrow \\
&\qquad\qquad \Big(\textstyle\bigvee_{\substack{\mathsf{r}_\ell = \left(\mathsf{A}''(x_1,\ldots,x_{\#(\mathsf{A}'')}) \leftarrow_{\mathcal{R}} \mathsf{b}''\right) \\ \mathsf{A}''(\xi_1,\ldots,\xi_{\#(\mathsf{A}'')}) = \mathsf{pred}_j(\mathsf{b}')}} U_\ell(\mathrm{succ}_j(x))\Big) \\
\mathit{Inst}(\mathbf{U},\mathbf{Z}) &\stackrel{\text{def}}{=} \forall x. \textstyle\bigwedge_{i=1}^{K} Z_i(x) \leftrightarrow \bigvee_{\substack{\mathsf{r}_j = \left(\mathsf{A}'(x_1,\ldots,x_{\#(\mathsf{A}')}) \leftarrow_{\mathcal{R}_{\mathsf{b}}} \mathsf{b}'\right) \\ \mathcal{B}_i(z) \in \mathsf{inst}(\mathsf{b}')}} U_j(x) \\
\mathit{Config}(\mathbf{X},\mathbf{Z}) &\stackrel{\text{def}}{=} \forall x. \textstyle\bigwedge_{S \neq T \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}} \Big(\neg X_S(x) \vee \neg X_T(x)\Big) \wedge \Big(\bigvee_{S \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}} X_S(x)\Big) \leftrightarrow \Big(\bigvee_{j=1}^{K} Z_j(x)\Big)
\end{aligned}
$$

uniquely determine a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$, such that $T_i \subseteq \mathrm{nodes}(\mathcal{T})$ is the set of nodes labeled by the rule r_i, for all $i \in [1,N]$.

Further, let $\mathsf{t} \in [\mathfrak{C}(\mathcal{T})]$ be unique ground term defined in the following way: for each instance atom $\mathcal{B}_i(x)$ that occurs in b, the variable x is substituted with the unique node of $\mathcal{T}$ where this atom occurs. This substitution determines the sets of instances for each behavioral type $\mathcal{B}_1,\ldots,\mathcal{B}_K$, encoded by the second order variables $\mathbf{Z} = \langle Z_1,\ldots,Z_K\rangle$, in the $\mathit{Inst}(\mathbf{U},\mathbf{Z})$ formula (Fig. 3). Note that, by Assumption 1, there is at most one node $w \in \mathrm{nodes}(\mathcal{T})$ such that $\mathcal{T}(w) = (\mathsf{A}_w(x_1,\ldots,x_{\#(\mathsf{A}_w)}) \leftarrow \mathsf{b}_W)$ and $\mathcal{B}_i(x) \in \mathsf{inst}(\mathsf{b})$. Moreover, each such node contains at most one instance atom, thus different instance atoms are assigned different identifiers. Finally, the architecture $\mathcal{A}(\mathbf{T})$ is the union of the ground architectures that occur in t, formally $\mathcal{A}(\mathbf{T}) \stackrel{\text{def}}{=} [\![\Gamma]\!]$, where $\mathsf{t}^{\rightsquigarrow} = \langle\Gamma\rangle(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ is the canonical form of t obtained by exhaustive application of the flattening relation (10).

4.3 Trap Invariants for Behavioral Terms

Let $C = \langle\mathcal{B}_1,\ldots,\mathcal{B}_K,\mathcal{A}\rangle$ be the parametric system corresponding to the given behavioral term b and the rewriting system $\mathcal{R}$. The sets of configurations of C are represented by tuples of second order variables $\mathbf{X} \stackrel{\text{def}}{=} \langle X_S \mid S \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}\rangle$ and $\mathbf{Y} \stackrel{\text{def}}{=} \langle Y_S \mid S \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}\rangle$, where a variable X_S (respectively Y_S) encodes the set of indices $i \in \mathbb{I}$ such that the instance $\mathcal{B}_j(i)$ is in state $S(i)$, for all $j \in [1,K]$. For a mapping $\nu : \mathbf{X} \rightarrow 2^{\mathbb{I}}$, we define $\nu(\mathbf{X}) \stackrel{\text{def}}{=} \langle \nu(X_S) \mid S \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}\rangle$. The *Config* formula (Fig. 3) ensures that $\nu(\mathbf{X})$ defines a configuration σ, for each satisfying valuation ν, by requiring that the sets assigned to $\mathbf{X}$ are a partition of the set of indices of the instances from the system, assigned to $\mathbf{Z}$. If $\nu \models \mathit{Config}(\mathbf{X},\mathbf{Z})$, we write $\nu(\mathbf{X}) \triangleright \sigma$ iff $\sigma = \{S(i) \mid S \in \mathcal{S}_{\mathcal{B}_j},\ i \in \nu(X_S),\ j \in [1,K]\}$.

For the time being, we assume the existence of a $\mathsf{WS}\kappa\mathsf{S}$ formula satisfying the condition below, the definition of which will be given in §4.3:

$$\nu \models \mathit{Flow}(\mathbf{X},\mathbf{Y},\mathbf{U}) \iff \nu(\mathbf{X}) \triangleright {}^{\bullet}\pi \text{ and } \nu(\mathbf{Y}) \triangleright \pi^{\bullet}, \text{ for some } \pi \in \mathcal{A}(\nu(\mathbf{U})) \qquad (12)$$

Intuitively, *Flow* is satisfied by any valuation that assigns $\mathbf{X}$ and $\mathbf{Y}$ sets of identifiers defining the pre- and post-configurations of an interaction from the architecture defined by the valuation of $\mathbf{U}$. With these definitions, the following formula translates the conditions of Definition 4, describing (parametric) traps:

$$Trap(\mathbf{X},\mathbf{U}) \stackrel{\text{def}}{=} \forall \mathbf{Y}^1 \forall \mathbf{Y}^2 \,.\, Flow(\mathbf{Y}^1,\mathbf{Y}^2,\mathbf{U}) \wedge inter(\mathbf{X},\mathbf{Y}^1) \rightarrow inter(\mathbf{X},\mathbf{Y}^2)$$
$$inter(\mathbf{X},\mathbf{Y}) \stackrel{\text{def}}{=} \exists x. \bigvee_{j=1}^{K} \bigvee_{S \in \mathcal{S}_{\mathcal{B}_j}} X_S(x) \wedge Y_S(x)$$

where $\mathbf{Y}^i$ is the copy of the tuple $\mathbf{Y}$ with variables superscripted by i, for $i = 1,2$. The set of configurations defined by the formula below is the trap invariant (Definition 4) of C, for each parameter-compatible interpretation of $\mathbf{U}$:

$$\begin{aligned} TrapInv(\mathbf{X},\mathbf{U}) \stackrel{\text{def}}{=} {} & \exists \mathbf{Z} \,.\, Inst(\mathbf{U},\mathbf{Z}) \wedge Config(\mathbf{X},\mathbf{Z}) \;\wedge \\ & \forall \mathbf{Y}^1 \forall \mathbf{Y}^2 \,.\, Init(\mathbf{Y}^1,\mathbf{Z}) \wedge Trap(\mathbf{Y}^2,\mathbf{U}) \wedge inter(\mathbf{Y}^1,\mathbf{Y}^2) \rightarrow inter(\mathbf{X},\mathbf{Y}^2) \end{aligned} \tag{13}$$
$$Init(\mathbf{X},\mathbf{Z}) \stackrel{\text{def}}{=} \bigwedge_{j=1}^{K} \forall x \,.\, Z_j(x) \leftrightarrow X_{\mathcal{I}_{\mathcal{B}_j}}(x)$$

where the formula *Init* defines the initial configuration of the parametric system, in which each instance is in the initial state of its component type. The following lemma proves that, assuming the existence of a formula *Flow* satisfying the condition (16), the formula *TrapInv* correctly defines the (parametric) trap invariant of the parametric system corresponding to $\mathcal{R}$ and b:

Lemma 1. *Let $T_1,\ldots,T_N \subseteq \mathbb{I}$ be finite sets such that $[U_1 \leftarrow T_1,\ldots,U_N \leftarrow T_N] \models RTree(\mathbf{U})$. Then $\Theta(C(\mathbf{T})) = \{\sigma \mid \nu(\mathbf{X}) \rhd \sigma,\ \nu[U_1 \leftarrow T_1,\ldots,U_N \leftarrow T_N] \models TrapInv(\mathbf{X},\mathbf{U})\}$.*

Assuming that the E set is encoded by a formula *Bad*, the parametric safety problem has a positive answer if the following formula is unsatisfiable:

$$Safe(\mathbf{U}) \stackrel{\text{def}}{=} RTree(\mathbf{U}) \wedge \exists \mathbf{X} \,.\, TrapInv(\mathbf{X},\mathbf{U}) \wedge Bad(\mathbf{X},\mathbf{U}) \tag{14}$$

As a typical example of a set of unsafe states, we consider the following definition of deadlock configurations, i.e. configurations in which no interaction can be fired:

$$DeadLock(\mathbf{X},\mathbf{U}) \stackrel{\text{def}}{=} \forall \mathbf{Y}^1 \forall \mathbf{Y}^2 \,.\, Flow(\mathbf{Y}^1,\mathbf{Y}^2,\mathbf{U}) \rightarrow \exists x. \bigvee_{j=1}^{K} \bigvee_{S \in \mathcal{S}_{\mathcal{B}_j}} Y^1_S(x) \wedge \neg X_S(x) \tag{15}$$

4.4 The Flow of a Behavioral Term

To complete the definition of trap invariants using $\mathsf{WS}\kappa\mathsf{S}$, we are left with defining the $Flow(\mathbf{X},\mathbf{Y},\mathbf{U})$ formula (12), that holds whenever $(\mathbf{X},\mathbf{Y})$ encodes the pairs of pre- and post-configurations of some interaction from $C(\mathbf{T})$, when $\mathbf{U}$ are interpreted by the sets of identifiers $\mathbf{T}$. We recall that $\mathcal{R}_\mathsf{b} = \{\mathsf{r}_1,\ldots,\mathsf{r}_N\}$ and that we have assumed the rules in $\mathcal{R}_\mathsf{b}$ to be of the form $\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \nu y_1 \ldots \nu y_m \,.\, \langle \Gamma \rangle(\mathsf{t}_1,\ldots,\mathsf{t}_n)$, where each t_i is an atom and at most one t_i is an instance atom. Moreover, assuming $\Gamma = \sum_{i=1}^{k} \prod_{j=1}^{h_i} P_{ij}(x_{ij})$, we denote $\mathsf{Inter}(\mathsf{r}) \stackrel{\text{def}}{=} \{\{P_{ij}(x_{ij}) \mid j \in [1,h_i]\} \mid i \in [1,k]\}$ the set of interactions occurring in r.

Assumption 2 *For any component type $\mathcal{B} = \langle \mathcal{P},\mathcal{S},\mathcal{I},\Delta \rangle$ and any two transition rules $S_1 \xrightarrow{P_1}_{\mathcal{B}} T_1, S_2 \xrightarrow{P_2}_{\mathcal{B}} T_2$, if $P_1 = P_2$ then $S_1 = S_2$ and $T_1 = T_2$. For a transition rule $S \xrightarrow{P} T \in \Delta_{\mathcal{B}}$, let ${}^\bullet P \stackrel{\text{def}}{=} S$ and $P^\bullet \stackrel{\text{def}}{=} T$ denote the pre- and post-state of the unique transition rule whose label is P.*

Fig. 4: Definition of the *Flow* Formula

$$Flow(\mathbf{X},\mathbf{Y},\mathbf{U}) \overset{\mathsf{def}}{=} \bigvee_{1 \le i \le N} \bigvee_{\pi \in \mathsf{Inter}(\mathsf{r}_i)} IFlow_{i,\pi}(\mathbf{X},\mathbf{Y},\mathbf{U}) \tag{16}$$

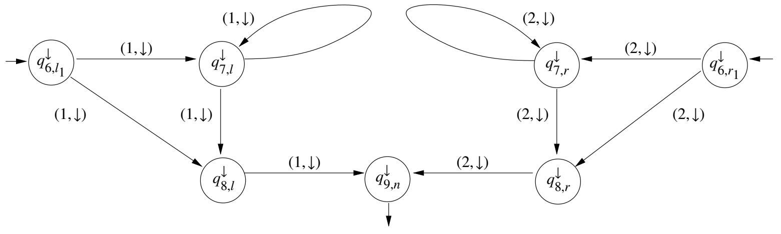
$$\begin{aligned} IFlow_{\ell,\{P_1(x_1),\ldots,P_n(x_n)\}}(\mathbf{X},\mathbf{Y},\mathbf{U}) \overset{\mathsf{def}}{=} {} & \exists y_0 \ldots \exists y_n \,.\, U_\ell(y_0) \wedge \\ & \bigwedge_{i=1}^{n} \Big(\bigvee_{\substack{\mathsf{r}' = (\mathsf{A}'(x_1,\ldots,x_{\#(\mathsf{A}')}) \leftarrow_{\mathcal{R}_\mathsf{b}} \mathsf{b}') \\ \mathcal{B}(y_i) \in \mathsf{inst}(\mathsf{b}')}} Path^{x_i,y_i}_{\mathsf{r}_\ell,\mathsf{r}'}(y_0,y_i,\mathbf{U}) \Big) \wedge \\ \forall x. & \bigwedge_{S \in \bigcup_{j=1}^{K} \mathcal{S}_{\mathcal{B}_j}} \Big[\Big(X_S(x) \leftrightarrow \bigvee_{{}^\bullet P_k = S} x = y_k \Big) \wedge \Big(Y_S(x) \leftrightarrow \bigvee_{P_k{}^\bullet = S} x = y_k \Big) \Big] \end{aligned} \tag{17}$$

The above assumption can be lifted at the cost of cluttering the following presentation. The *Flow* formula (16) is defined in Fig. 4. Essentially, *Flow* is split into a disjunction of $IFlow_{\ell,\{P_1(x_1),\ldots,P_n(x_n)\}}$ formulæ (17), one for each set of ports $\{P_1(x_1),\ldots,P_n(x_n)\}$ that denotes an interaction of the rule r_ℓ, for all $\ell \in [1,N]$. To understand the formulæ (17), recall that each of the variables $x_1,\ldots,x_n$ is interpreted as the (unique) node of the rewriting tree containing an instance atom $\mathcal{B}_i(x_i)$. In order to find this node, we track the variable x_i from the current node y_0, labeled by the rule r_ℓ, to the node y_i, where this instance atom occurs. This is done by the $Path^{z,u}_{\mathsf{r},\mathsf{r}'}(x,y,\mathbf{U})$ formula, that holds iff $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathsf{b})$ is a rewriting tree, uniquely encoded by the interpretation of the $\mathbf{U}$ variables, and x,y are mapped to the endpoints of a path from a node $w \in \mathrm{nodes}(\mathcal{T})$, with label $\mathcal{T}(w) = \mathsf{r}$ to a node $w' \in \mathrm{nodes}(\mathcal{T})$, with label $\mathcal{T}(w') = \mathsf{r}'$, such that z and u are variables that occur in the bodies of r and r', respectively, mapped to the same identifier (node) in any ground term from the set $[\mathfrak{C}(\mathcal{T})]$. Note that, by the definition of ground sets, two different variables are mapped to the same identifier only if they are replaced by the same variable, when $\mathfrak{C}(\mathcal{T})$ is built from the labels of $\mathcal{T}$ (Definition 3).

We encode sets of paths in a rewriting tree by a finite automaton and use a classical result from automata theory to define $Path^{z,u}_{\mathsf{r},\mathsf{r}'}(x,y,\mathbf{U})$ by turning the finite automaton into a $\mathsf{WS}\kappa\mathsf{S}$ formula. But first, let us define paths in a tree formally. Given a tree $\mathcal{T}$, with $\mathrm{nodes}(\mathcal{T}) \subseteq [0,\kappa-1]^*$, a *path* is a finite sequence of nodes $\rho = n_1,\ldots,n_\ell$ 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 $\alpha_i \in [0,\kappa-1]$. The path is determined by the source node and the sequence $(\alpha_1,d_1)\ldots(\alpha_{\ell-1},d_{\ell-1})$ of *directions* $(\alpha_i,d_i) \in [0,\kappa-1]\times\{\uparrow,\downarrow\}$, with the following meaning: $d_i = \uparrow$ if $n_{i+1}\alpha_i = n_i$ and $d_i = \downarrow$ if $n_{i+1} = n_i\alpha_i$.

A *path automaton* is a tuple $A = (Q,I,F,\delta)$, where Q is a set of states, $I,F \subseteq Q$ are the initial and final states, respectively, and $\delta \subseteq Q \times [0,\kappa-1] \times \{\uparrow,\downarrow\} \times Q$ is a set of transitions of the form $q \xrightarrow{(\alpha,d)} q'$, with $\alpha \in [0,\kappa-1]$ being a direction and $d \in \{\uparrow,\downarrow\}$ indicates whether the automaton moves up or down in the tree. A run of A over the path $\omega = (\alpha_1,d_1)\ldots(\alpha_{n-1},d_{n-1})$ 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 iff $q_n \in F$ and the *language* of A is the set of paths over which A has an accepting run, denoted $\mathcal{L}(A)$.

Fig. 5: Path Automata Recognizing the Instantiation Paths from Example 1



A path automaton $A = (Q, I, F, \delta)$ corresponds, in the sense of Lemma 2 below, to the following $\mathsf{WS}\kappa\mathsf{S}$ formula, that can be effectively built from the description of A:

$$\Phi_A(x,y,\overline{\mathbf{X}}) \stackrel{\mathsf{def}}{=} \bigwedge_{1\le i\ne j\le N} \forall z.\big(\neg\overline{X}_i(z)\vee\neg\overline{X}_j(z)\big) \wedge \bigvee_{q_i\in I}\overline{X}_i(x) \;\wedge\; \bigvee_{q_j\in F}\overline{X}_j(y)\;\wedge$$
$$\bigwedge_{i=1}^{N} \forall z \,.\, z\ne y\wedge \overline{X}_i(z)\rightarrow\Big(\bigvee_{q_i\xrightarrow{(\alpha,\downarrow)}q_j}\overline{X}_j(\mathrm{succ}_\alpha(z))\vee\bigvee_{q_i\xrightarrow{(\alpha,\uparrow)}q_j}\exists z' \,.\, \mathrm{succ}_\alpha(z')=z\wedge\overline{X}_j(z')\Big)$$
$$\bigwedge_{i=1}^{N} \forall z \,.\, z\ne x\wedge\overline{X}_j(z)\rightarrow\Big(\bigvee_{q_i\xrightarrow{(\alpha,\downarrow)}q_j}\exists z' \,.\, \mathrm{succ}_\alpha(z')=z\wedge\overline{X}_i(z')\vee\bigvee_{q_i\xrightarrow{(\alpha,\uparrow)}q_j}\overline{X}_i(\mathrm{succ}_\alpha(z))\Big)$$

where $Q = \{q_1,\ldots,q_L\}$ and $\overline{\mathbf{X}} = \langle\overline{X}_1,\ldots,\overline{X}_L\rangle$ 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 encode the facts that $\overline{\mathbf{X}}$ are disjoint (no tree node is labeled by more than one state during the run), the run starts in an initial state with node x and ends in a final state with node y. The fourth conjunct 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 is the reversed flow condition on the path, needed to ensure that $\overline{\mathbf{X}}$ do not contain useless nodes, being thus symmetric to the fourth. The following lemma is adapted from folklore automata-logic connection results[2] [17, §2.10]:

Lemma 2. *Given a tree $\mathcal{T}$ with* $\mathrm{nodes}(\mathcal{T}) \subseteq [0,\kappa-1]^*$ *and a path* $\omega \in ([0,\kappa-1]\times\{\uparrow,\downarrow\})^*$ *from w_1 to w_2 in $\mathcal{T}$, we have $\omega \in \mathcal{L}(A)$ iff $[x \leftarrow w_1, y \leftarrow w_2] \models \exists\overline{\mathbf{X}} \,.\, \Phi_A(x,y,\overline{\mathbf{X}})$.*

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. For example, the paths that track the instantiations of the variables l_1^ϵ and r_1^ϵ in the rewriting tree for the term *Root*() generated by the rewriting system from Example 1 are depicted in red in Fig. 2. To this end, we define a path automaton that tracks the instantiation of variables from the rewriting system $\mathcal{R}_{\mathsf{b}}$. 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 $A^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2} \stackrel{\mathsf{def}}{=} (Q, I^{z_1}_{\mathsf{r}_1}, F^{z_2}_{\mathsf{r}_2}, \delta)$ as follows. We associate a state $q^d_{\mathsf{r},z}$ to each rule $\mathsf{r} = (\mathsf{A}(x_1,\ldots,x_{\#\mathsf{A}}) \leftarrow_{\mathcal{R}_\mathsf{b}} \mathsf{b}')$, each variable z occurring (free or bound) in b' and each direction $d \in \{\uparrow,\downarrow\}$. The sets of initial and final states are $I^{z_1}_{\mathsf{r}_1} \stackrel{\mathsf{def}}{=} \{q^d_{\mathsf{r}_1,z_1} \mid d =\uparrow,\downarrow\}$ and $F^{z_2}_{\mathsf{r}_2} \stackrel{\mathsf{def}}{=} \{q^{\downarrow}_{\mathsf{r}_2,z_2}\}$. The transition relation consists of the triples $q^{\downarrow}_{\mathsf{r}_1,y_j} \xrightarrow{(\alpha,\downarrow)} q^{\downarrow}_{\mathsf{r}_2,x_j}$, $q^{\uparrow}_{\mathsf{r}_2,x_j} \xrightarrow{(\alpha,\uparrow)} q^{\uparrow}_{\mathsf{r}_1,y_j}$ and $q^{\uparrow}_{\mathsf{r}_2,x_j} \xrightarrow{(\alpha,\uparrow)} q^{\downarrow}_{\mathsf{r}_1,y_j}$, for any two distinct rules $\mathsf{r}_i = (\mathsf{A}_j(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow_{\mathcal{R}_\mathsf{b}} \mathsf{b}_i)$, $i = 1,2,$

[2] A similar conversion of tree walking automata to MSO has been described in [14].

all $\alpha \in [0, \#_{\mathsf{pred}}(\mathsf{b}_1)]$, such that $\mathsf{pred}_\alpha(\mathsf{b}_1) = \mathsf{A}_2(y_1, \ldots, y_{\#(\mathsf{A}_2)})$ and all $j \in [1, \#(\mathsf{A}_2)]$. For instance, the path automata that recognize the instantiation paths for the variables l_1^ϵ and r_1^ϵ in the rewriting tree for the term *Root*() generated by the rewriting system from Example 1 are depicted in Fig. 5. The initial states are $q^{\downarrow}_{6,l_1}$ and $q^{\downarrow}_{6,r_1}$, respectively, and the final state is $q^{\downarrow}_{9,n}$ in both cases, where the labels of the rules of the rewriting system are the ones from Example 1. We define the $\mathit{Path}^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}$ formula following the below lemma, proving the correctness of the automata construction:

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 $\mathcal{T}(w_i) = \mathsf{r}_i = \big(\mathsf{A}_i(x_{i,1}, \ldots, x_{i,\#(\mathsf{A}_i)}) \leftarrow_{\mathcal{R}_\mathsf{b}} \mathsf{b}_i\big)$, 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 mapped to the same identifier in any ground term $\mathsf{t} \in [\mathfrak{C}(\mathcal{T})]$,*
2. *$A^{x_{1,k_1},\, x_{2,k_2}}_{\mathsf{r}_1,\mathsf{r}_2}$ accepts the sequence of directions labeling the path from w_1 to w_2 in $\mathcal{T}$.*

$$\mathit{Path}^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}(x,y,\mathbf{U}) \stackrel{\mathsf{def}}{=} \exists \overline{X}_1 \ldots \exists \overline{X}_L \,.\, \Phi_{A^{z_1,z_2}_{\mathsf{r}_1,\mathsf{r}_2}}(x,y,\overline{\mathbf{X}}) \wedge \Psi(\overline{\mathbf{X}},\mathbf{U})$$
$$\Psi(\overline{\mathbf{X}},\mathbf{U}) \stackrel{\mathsf{def}}{=} \bigwedge_{d=\uparrow,\downarrow} \bigwedge_{\mathsf{r}_i=\left(\mathsf{A}'(x_1,\ldots,x_{\#(\mathsf{A}')})\leftarrow_{\mathcal{R}_\mathsf{b}}\mathsf{b}'\right)} \bigwedge_{z\in\mathrm{fv}(\mathsf{b}')} \forall x \,.\, \overline{X}^{d}_{\mathsf{r},z}(x) \rightarrow U_i(x)$$

The formula Ψ 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. The lemma below proves that the definition (16) of the formula *Flow* meets condition (12):

Lemma 4. *For any valuation $\nu : \mathbf{X} \cup \mathbf{Y} \cup \mathbf{U} \cup \mathbf{Z} \to 2^{\mathbb{I}}$, such that $\nu \models \mathit{RTree}(\mathbf{U}) \wedge \mathit{Inst}(\mathbf{U},\mathbf{Z}) \wedge \mathit{Config}(\mathbf{X},\mathbf{Z}) \wedge \mathit{Config}(\mathbf{Y},\mathbf{Z})$, the following are equivalent:*

1. *$\nu \models \mathit{Flow}(\mathbf{X},\mathbf{Y},\mathbf{U})$,*
2. *$\nu(\mathbf{X}) \rhd {}^\bullet\pi$ and $\nu(\mathbf{Y}) \rhd \pi^\bullet$, for some interaction $\pi \in \mathcal{A}(\nu(\mathbf{U}))$.*

Together with Lemma 1, this ensures that the trap invariant of the parametric system corresponding to $\mathcal{R}$ and b is defined in WSκS, by the *TrapInv* formula (13). Hence the verification of safety properties (such as absence of deadlocks) is reduced to checking the satisfiability of the *Safe* formula (14), leading to the following result:

Theorem 1. *Given a closed behavioral term b, a rewriting system $\mathcal{R}$, a formula $\mathit{Bad}(\mathbf{X},\mathbf{U})$ and a tuple of sets $T_1, \ldots, T_N \subseteq \mathbb{I}$, that are parameter-compatible with $\mathcal{R}$ and b, the behavior $C(\mathbf{T})$ is safe w.r.t the set of configurations $\mathsf{E} \stackrel{\mathsf{def}}{=} \{\sigma \mid \nu[U_1 \leftarrow T_1, \ldots, U_N \leftarrow T_N](\mathbf{X}) \rhd \sigma,\ \nu[U_1 \leftarrow T_1, \ldots, U_N \leftarrow T_N] \models \mathit{Bad}(\mathbf{X},\mathbf{U})\}$ if $\mathit{Safe}(\mathbf{U})$ is unsatisfiable.*

5 Experimental Evaluation

We implemented the trap invariant synthesis in a prototype tool[3] that generates the WSκS formula corresponding to the (sufficient) deadlock freedom condition (14) from a given behavioral term and a rewriting system. Our test cases are hand-crafted examples of common architectures encountered in practice (e.g. pipelines and stars), textbook examples (dining philosophers) and several hierarchical tree-shaped architectures with rather complex architectural patterns (trees with root links or leaves linked in a ring).

[3] Available online at `https://github.com/raduiosif/rtab`.

The table below shows the results of checking deadlock freedom of several test cases. The 2nd column gives the number of states $n_1 \times \ldots \times n_K$, where n_i is the number of states in the i-th component type and K is the number of component types from the system. The number of rewriting rules and interactions in the specification are given in the 3rd and 4th columns, respectively. The 5th column reports the result of the satisfiability check (14) using the Mona v1.4-18 tool [13] and the 6th column shows the runing times (in seconds) on an Debian AMD64 2GHz machine with 16GB of RAM. The 7th and 8th columns report the type of invariant (trap or 1-invariant) used to prove deadlock freedom and the 9th column gives the type of WSκS logic, for $\kappa \in \{1,2\}$.

benchmark	#states/comp.	#rules	#inter.	deadlock	time (sec)	trap-inv	1-inv	κ
ring	2×2	3	3	✓	0.01	✓	-	1
star	2×2	3	4	✓	0.01	✓	-	1
star-ring	$2\times3\times3$	3	9	✓	0.03	✓	-	1
alt-philo-sym	3×2	3	9	×	0.70	✓	✓	1
alt-philo-asym	3×2	3	9	✓	0.67	✓	✓	1
sync-philo	2×2	3	6	✓	0.03	✓	-	1
tree-dfs	$2\times6\times2$	4	6	✓	0.07	✓	-	2
tree-back-root	2×2	3	5	✓	0.03	✓	-	2
tree-linked-leaves	$2\times2\times4\times3$	4	10	✓	0.27	✓	-	2

The ring, star and ring-star test cases correspond to a simple token-ring, a star with one master (coordinator) and $n \geq 2$ slaves and a star with n slaves linked in a token-ring.

The alt-philo-sym and alt-philo-asym examples correspond to the dining philosophers in which the philosophers pick their left and right forks separately, with all symmetric philosophers and one asymetric philosopher, respectively. The sync-philo example models the dining philosophers in which every philosopher picks her forks simultaneously. It is known that alt-philo-sym reaches a deadlock configuration, whereas alt-philo-asym and sync-philo are deadlock free. Moreover, the alt-philo-asym system cannot be the proved deadlock free using trap invariants only [4, Proposition 1]. Following the solution from [4], we used the structural information given by the *Flow* formula (16) to synthethize 1-*invariants*, i.e. inductive sets of configurations that contain exactly one active state at the time[4].

The tree-dfs example models a binary tree architecture traversed by a token in depth-first order, while the (i) tree-back-root and (ii) tree-linked-leaves (Example 1) go beyond trees, modeling hierarchical systems with parent-children communication on top of which (i) the nodes communicate with the root and (ii) the leaves are linked in a token-ring, respectively.

6 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.

One of the drawbacks that prevented us from tackling more real-life examples is the lack of support for broadcast communication (i.e. interactions that involve an unbounded number of participants). We plan on adding support for broadcast in our behavioral term algebra and develop further the invariant synthesis method to take broadcast into account, as future work.

[4] We refer the reader to [4, Definition 1] for a formal definition of 1-invariants.

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.: Verifying safety properties of inductively defined parameterized systems. Tech. Rep. 2008.04160, arXiv (2020)
6. 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)
7. Bradbury, J.S.: Organizing definitions and formalisms for dynamic software architectures. Tech. rep., In Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems Newport (2004)
8. Browne, M., Clarke, E., Grumberg, O.: Reasoning about networks with many identical finite state processes. Information and Computation 81(1), 13 – 31 (1989)
9. 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)
10. 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)
11. 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)
12. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
13. 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)
14. 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)
15. 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)
16. Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network invariants in action. In: Brim, L., Jancar, P., Kretínský, M., Kucera, A. (eds.) CONCUR 2002 - Concurrency Theory, 13th International Conference. LNCS, vol. 2421, pp. 101–115. Springer (2002)
17. Khoussainov, B., Nerode, A.: Automata Theory and its Applications. Springer (2001)

18. 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)
19. Kramer, J., Magee, J.: Analysing dynamic change in distributed software architectures. IEE Proceedings - Software 145(5), 146–154 (1998)
20. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 346–357. ACM Press (1997)
21. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: Modeling architecture styles. J. Log. Algebr. Meth. Program. 86(1), 2–29 (2017)
22. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
23. 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)
24. Shtadler, Z., Grumberg, O.: Network grammars, communication behaviors and automatic verification. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems, International Workshop. LNCS, vol. 407, pp. 151–165. Springer (1989)
25. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems, International Workshop. LNCS, vol. 407, pp. 68–80. Springer (1989)