Local Reasoning about Parameterized Reconfigurable Distributed Systems

Emma Ahrens, Marius Bozga, Radu Iosif, Joost-Pieter Katoen

Reasoning about Distributed Reconfigurable Systems

EMMA AHRENS, RWTH Aachen University, Germany
MARIUS BOZGA, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France
RADU IOSIF, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France
JOOST-PIETER KATOEN, RWTH Aachen University, Germany

This paper presents a Hoare-style calculus for formal reasoning about reconfiguration programs of distributed systems. Such programs create and delete components and/or interactions (connectors) while the system components change state according to their internal behaviour. Our proof calculus uses a resource logic, in the spirit of Separation Logic, to give local specifications of reconfiguration actions. Moreover, distributed systems with an unbounded number of components are described using inductively defined predicates. The correctness of reconfiguration programs relies on havoc invariants, that are assertions about the ongoing interactions in a part of the system that is not affected by the structural change caused by the reconfiguration. We present a proof system for such invariants in an assume/rely-guarantee style. We illustrate the feasibility of our approach by proving the correctness of real-life distributed systems with reconfigurable (self-adjustable) tree architectures.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: dynamic reconfiguration, parameterized systems, local reasoning

ACM Reference Format:
Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. 2022. Reasoning about Distributed Reconfigurable Systems. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 130 (October 2022), 30 pages. https://doi.org/10.1145/3563293

ACKNOWLEDGMENTS

The authors wish to acknowledge the support of the French National Research Agency project Non-Aggregative Resource COmpositions (NARCO) under grant number ANR-21-CE48-0011.

1 INTRODUCTION

The relevance of dynamic reconfiguration. Dynamic reconfigurable distributed systems are used increasingly as critical parts of the infrastructure of our digital society, e.g. datacenters, e-banking and social networking. In order to address maintenance (e.g., replacement of faulty and obsolete network nodes by new ones) and data traffic issues (e.g., managing the traffic inside a datacenter [Noormohammadpour and Raghavendra 2018]), the distributed systems community has recently put massive effort in designing algorithms for *reconfigurable systems*, whose network topologies change

Authors' addresses: Emma Ahrens, Software Modeling and Verification (MOVES), RWTH Aachen University, Aachen, D-52056, Germany, Emma.Ahrens@inst1.edu; Marius Bozga, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, 38000, France, Marius.Bozga@univ-grenoble-alpes.fr; Radu Iosif, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, 38000, France, Radu.Iosif@univ-grenoble-alpes.fr; Joost-Pieter Katoen, Software Modeling and Verification (MOVES), RWTH Aachen University, Aachen, D-52056, Germany, katoen@cs.rwth-aachen.de.

2475-1421/2022/10-ART130
https://doi.org/10.1145/3563293

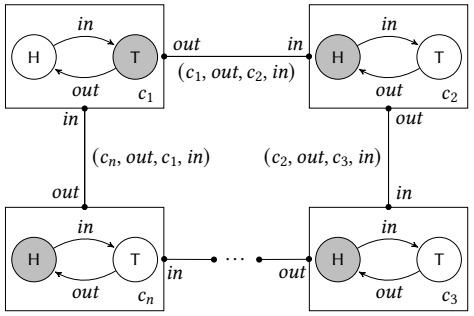
at runtime [Foerster and Schmid 2019]. This development provides new impulses to distributed algorithm design [Michail et al. 2020] and has given rise to self-adjustable network architectures whose topology reconfigurations are akin to amendments of dynamic data structures such as splay trees [Peres et al. 2019]. However, reconfiguration is an important source of bugs, that may result in e.g., denial of services[1]. *This paper introduces a logical framework for reasoning about the safety properties of such systems, in order to prove e.g., absence of deadlocks or critical section violations.*

Modeling distributed systems. In this paper we model distributed systems at the level of abstraction commonly used in component-based design of large heterogeneous systems [Magee and Kramer 1996]. We rely on a clean separation of (a finite-state abstraction of) the *behaviour* from the *coordination* of behaviors, described by complex graphs of components (nodes) and interactions (edges). Mastering the complexity of a distributed system requires a deep understanding of the coordination mechanisms. We distinguish between *endogenous* coordination, that explicitly uses synchronization primitives in the code describing the behavior of the components (e.g. semaphores, monitors, compare-and-swap, etc.) and *exogenous* coordination, that defines global rules describing how the components interact. These two orthogonal paradigms play different roles in the design of a system: exogenous coordination is used during high-level model building, whereas endogenous coordination is considered at a later stage of development, to implement the model using low-level synchronization primitives.

Here we focus on *exogenous coordination* of distributed systems, consisting of an unbounded number of interconnected components, with a flexible topology, i.e. not fixed *à priori*. We abstract from low-level coordination mechanisms between processes such as semaphores, compare-and-swap operations and the like. Components behave according to a small set of finite-state abstractions of sequential programs, whose transitions are labeled with events. They communicate via interactions (handshaking) modeled as sets of events that occur simultaneously in multiple components. Despite their apparent simplicity, these models capture key aspects of distributed computing, such as message delays and transient faults due to packet loss. Moreover, the explicit graph representation of the network is essential for the modeling of dynamic reconfiguration actions.

Programming reconfiguration. The study of dynamic reconfiguration has led to the development of a big variety of formalisms and approaches to specify the changes to the structure of a system using e.g., graph-based, logical or process-algebraic formalisms (see [Bradbury et al. 2004] and [Butting et al. 2017] for surveys). With respect to existing work, we consider a simple yet general imperative reconfiguration language, encompassing four primitive reconfiguration actions (creation and deletion of components and interactions) as well as non-deterministic reconfiguration triggers (constraints) evaluated on small parts of the structure and the state of the system. These features exist, in very similar forms, in the vast majority of existing graph-based reconfiguration formalisms e.g., using explicit reconfiguration scripts as in CommUnity [Wermelinger et al. 2001], reconfiguration controllers expressed as production rules in graph-grammars [Le Metayer 1998], guarded reconfiguration actions in Dr-Bip [El-Ballouli et al. 2021] and graph rewriting rules in Reo [Krause et al. 2011], to cite only a few. In our model, the primitive reconfiguration actions are executed sequentially, but interleave with the firing of interactions i.e., the normal execution of the system. Sequential reconfiguration is not a major restriction, as the majority of reconfiguration languages rely on a centralized management [Bradbury et al. 2004]. Nevertheless, for the sake of simplicity, most existing reconfiguration languages avoid the fine-grain interleaving of reconfiguration and execution steps i.e., they freeze the system's execution during reconfiguration. Our choice of allowing this type of interleaving is more realistic and closer to real-life implementation. Finally,

[1]E.g., Google reports a cloud failure caused by reconfiguration: https://status.cloud.google.com/incident/appengine/19007



Listing (1) Delete Component (wrong version)

```
1 with x, y, z : ⟨x.out, y.in⟩ ∗ y@H ∗ ⟨y.out, z.in⟩
2 do
3   disconnect(y.out, z.in);
4   disconnect(x.out, y.in);
5   delete(y);
6   connect(x.out, z.in);
7 od
```

Listing (2) Delete Component (correct version)

```
1 with x, y, z : ⟨x.out, y.in⟩ ∗ y@H ∗ ⟨y.out, z.in⟩
2 do
3   disconnect(x.out, y.in);
4   disconnect(y.out, z.in);
5   delete(y);
6   connect(x.out, z.in);
7 od
```

Fig. 1. Reconfiguration of a Parametric Token Ring System

our language supports open reconfigurations, in which the number of possible configurations is unbounded [Butting et al. 2017], via non-deterministic choice and iteration.

An illustrative example. We illustrate the setting by a token ring example, consisting of a finite but unbounded number of components, indexed from 1 to n, connected via an unidirectional ring (Fig. 1). A token may be passed from a component i in state T (it has a token) to its neighbour, with index $(i \bmod n) + 1$, which must be in state H (it has a hole instead of a token). As result of this interaction, the i-th component moves to state H while the $(i \bmod n) + 1$ component moves to state T. Note that token passing interactions are possible as long as at least two components are in different states; if all the components are in the same state at the same time, the ring is in a *deadlock* configuration.

During operation, components can be added to, or removed from the ring. On removing the component with index i, its incoming (from $i-1$, if $i > 1$, or n, if $i = 1$) and outgoing (to $(i \bmod n)+1$) connectors are deleted before the component is deleted, and its left and right neighbours are reconnected in order to re-establish the ring-shaped topology. Consider the program in Listing 1, where the variables x, y and z are assigned indices i, $(i \bmod n) + 1$ and $(i \bmod n) + 2$, respectively (assuming $n > 2$). The program removes first the right connector between y and z (line 3), then removes the left connector between x and y (line 4), before removing the component indexed by y (line 5) and reconnecting the x and z components (line 6). Note that the order of the disconnect commands is crucial: assume that component x is the only one in state T in the entire system. Then the token may move from x to y and is deleted together with the component (line 5). In this case, the resulting ring has no token and the system is in a deadlock configuration. The reconfiguration program in Listing 2 is obtained by swapping lines 3 and 4 from Listing 1. In this case, the deleted component is in state H before the reconfiguration and its left connector is removed before its right one, thus ensuring that the token does not move to the y component (deleted at line 5).

The framework developed in this paper allows to prove that e.g., when applied to a token ring with at least two components in state H and at least one component in state T, the program in Listing 2 yields a system with at least two components in different states, *for any* $n > 2$. Using, e.g. invariant synthesis methods similar to those described in [Abdulla et al. 2007; Bozga et al. 2020;

Bozga and Iosif 2021; Bozga et al. 2021; Chen et al. 2017], an initially correct parametric systems can be proved to be correct *after the application of a sequence of reconfiguration actions*.

The contributions of this paper. Whereas various formalisms for modeling distributed systems support dynamic reconfiguration, the formal verification of system properties under reconfigurations has received scant attention. We provide a *configuration logic* that specifies the safe configurations of a distributed system. This logic is used to build Hoare-style proofs of correctness, by annotating reconfiguration programs (i.e. programs that delete and create interactions or components) with assertions that describe both the topology of the system (i.e. the components and connectors that form its coordinating architecture) and the local states of the components. The annotations of the reconfiguration program are proved to be valid under so-called *havoc invariants*, expressing global properties about the states of the components, that remain, moreover, unchanged under the ongoing interactions in the system. In order to prove these havoc invariants for networks of any size, we develop an induction-based proof system, that uses a parallel composition rule in the style of assume/rely-guarantee reasoning. In contrast with existing formal verification techniques, we do not consider the network topology to be fixed in advance, and allow it to change dynamically, as described by the reconfiguration program. This paper provides the details of our proof systems and the semantics of reconfiguration programs. We illustrate the usability of our approach by proving the correctness of self-adjustable tree architectures [Schmid et al. 2016] and conclude with a list of technical problems relevant for the automation of our method.

Main challenges. Formal reasoning about reconfigurable distributed systems faces two technical challenges. The first issue is the huge complexity of nowadays distributed systems, that requires highly scalable proof techniques, which can only be achieved by *local reasoning*, a key ingredient of other successful proof techniques, based on Separation Logic [O'Hearn et al. 2001]. To this end, atomic reconfiguration commands in our proof system are specified by axioms that only refer to the components directly involved in the action, while framing out the rest of the distributed system. This principle sounds appealing, but is technically challenging, as components from the local specification interfere with components from the frame[2]. To tackle this issue, we assume that frames are invariant under the exchange of messages between components (interactions) and discharge these invariance conditions using cyclic proofs. The inference rules used to write such proof rely on a compositional proof rule, in the spirit of *rely/assume-guarantee* reasoning [Jones 1981; Owicki and Gries 1978], whose assumptions about the environment behavior are automatically synthesized from the formulæ describing the system and the environment.

The second issue is dealing with the non-trivial interplay between reconfigurations and interactions. Reconfigurations change the system by adding/removing components/interactions *while the system is running*, i.e. while state changes occur within components by firing interactions. Although changes to the structure of the distributed system seem, at first sight, orthogonal to the state changes within components, the impact of a reconfiguration can be immense. For instance, deleting a component holding the token in a token-ring network yields a deadlocked system, while adding a component with a token could lead to a data race, in which two components access a shared resource simultaneously. Technically, this means that a frame rule cannot be directly applied to sequentially composed reconfigurations, as e.g. an arbitrary number of interactions may fire between two atomic reconfiguration actions. Instead, we must prove havoc invariance of the intermediate assertions in a sequential composition of reconfiguration actions. As an optimization of the proof technique, such costly checks do not have to be applied along sequential compositions of reconfiguration actions that only decrease/increase the size of the architecture; in such monotonic

[2]Essentially the equivalent of the environment in a compositional proof system for parallel programs.

reconfiguration sequences, invariance of a set of configurations under interaction firing needs only to be checked in the beginning (for decreasing sequences) or in the end (for increasing sequences).

2 A MODEL OF DISTRIBUTED SYSTEMS

A distributed system is modeled as a set of network nodes (called *components*), each running its own copy of the same program (called *behavior*). The components communicate via connectors (called *interactions*) that synchronize transitions in different components. In addition, there is a designated *reconfiguration manager* that has access to the entire network and is allowed to change it by executing a reconfiguration program.

A *configuration* is a snapshot of the system, describing the topology of the network and the current state of each component (for simplicity, the reconfiguration manager is not explicitly represented). The configuration is changed by executing either an interaction (which changes the states of the components involved) or a reconfiguration action (that adds/removes a component/interaction). The global behavior of the system is the sequence of configurations obtained by the interleaving of interactions (also called *havoc* actions) and reconfiguration actions.

2.1 Components, Behavior and Interactions

For a function $f : A \to B$, we denote by $\mathrm{dom}(f)$ its domain and by $f[a \leftarrow b]$ the function that maps a into b and behaves like f for all other elements from the domain of f. By $\mathrm{pow}(A)$ we denote the powerset of a set A. For a relation $R \subseteq A \times A$, we denote by R^* its reflexive and transitive closure. Given sets A and B, we write $A \subseteq_{fin} B$ if A is a finite subset of B and define $A \uplus B \stackrel{\text{def}}{=} A \cup B$ if $A \cap B = \emptyset$ and $A \uplus B$ is undefined, if $A \cap B \neq \emptyset$.

We model a distributed system by a finite set $C \subseteq_{fin} \mathbb{C}$, where $\mathbb{C}$ is a countably infinite universe of *components*. The components in C are said to be *present* (in the system) and those from $\mathbb{C} \setminus C$ are *absent* (from the system).

The present components can be thought of as the nodes of a network, each executing a copy of the same program, called *behavior* in the following. The behavior is described by a finite-state machine $\mathcal{B} = (P, Q, \to)$, where P is a finite set of *ports* i.e., the event alphabet of the machine, Q is a finite set of *states*, and $\to \subseteq Q \times P \times Q$ is a transition relation. We denote transitions as $q \xrightarrow{p} q'$ instead of (q, p, q'), the states q and q' being referred to as the pre- and post-state of the transition.

The network of the distributed system is described by a finite set $\mathcal{I} \subseteq_{fin} \mathbb{C} \times P \times \mathbb{C} \times P$ of *interactions*. Intuitively, an interaction (c_1, p_1, c_2, p_2) connects the port p_1 of component c_1 with the port p_2 of component c_2, provided that c_1 and c_2 are distinct components. Intuitively, an interaction (c_1, p_1, c_2, p_2) can be thought of as a joint execution of transitions labeled with the ports p_1 and p_2 from the components c_1 and c_2, respectively.

Definition 2.1. A *configuration* is a quadruple $\gamma = (C, \mathcal{I}, \varrho, \nu)$, where C and $\mathcal{I}$ describe the present components and the interactions of the system, $\varrho : C \to Q$ is a *state map* associating each present component a state of the common behavior $\mathcal{B} = (P, Q, \to)$ and $\nu : \mathbb{V} \to \mathbb{C}$ is a *store* that maps variables, taken from a countably infinite set $\mathbb{V}$, to components (not necessarily present). We denote by Γ the set of configurations.

Example 2.2. For instance, the configuration $(C, \mathcal{I}, \varrho, \nu)$ of the token ring system, depicted in Fig. 1 (left) has present components $C = \{c_1, \ldots, c_n\}$, interactions $\mathcal{I} = \{(c_i, out, c_{(i \bmod n)+1}, in) \mid i \in [1, n]\}$ and state map given by $\varrho(c_1) = \mathsf{T}$ and $\varrho(c_i) = \mathsf{H}$, for $i \in [2, n]$. The store ν is arbitrary. ■

Given a configuration $(C, \mathcal{I}, \varrho, \nu)$, an interaction $(c_1, p_1, c_2, p_2) \in \mathcal{I}$ is *loose* if and only if $c_i \notin C$, for some $i = 1, 2$. A configuration is *loose* if and only if it contains a loose interaction. Interactions (resp. configurations) that are not loose are said to be *tight*. In particular, loose configurations are

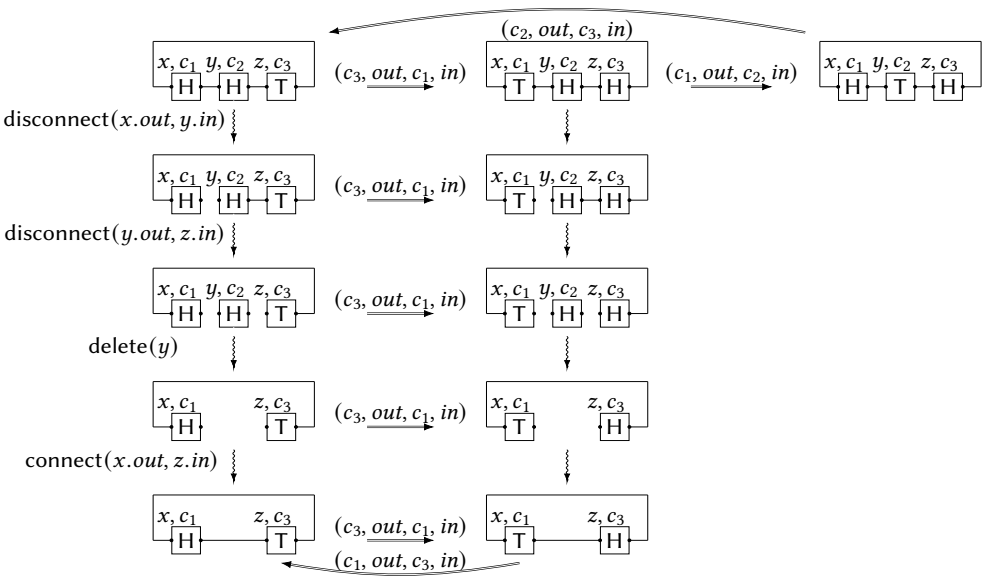


Fig. 2. Havoc and Reconfigurations of a Token Ring

useful for the definition of a composition operation, as the union of disjoint sets of components and interactions, respectively:

Definition 2.3. The composition of two configurations $\gamma_i = (C_i, \mathcal{I}_i, \varrho_i, \nu)$, for $i = 1, 2$, is defined as $\gamma_1 \bullet \gamma_2 \stackrel{\text{def}}{=} (C_1 \uplus C_2, \mathcal{I}_1 \uplus \mathcal{I}_2, \varrho_1 \cup \varrho_2, \nu)$. The composition $\gamma_1 \bullet \gamma_2$ is undefined if either $C_1 \uplus C_2$ or $\mathcal{I}_1 \uplus \mathcal{I}_2$ is undefined[3]. A composition $\gamma_1 \bullet \gamma_2$ is *trivial* if $C_i = \mathcal{I}_i = \varrho_i = \emptyset$, for some $i = 1, 2$. A configuration γ_2 is a *subconfiguration* of γ_1, denoted $\gamma_2 \sqsubseteq \gamma_1$, if and only if there exists a configuration $\gamma_3 \in \Gamma$, such that $\gamma_1 = \gamma_2 \bullet \gamma_3$.

Note that a tight configuration may be the result of composing two loose configurations, whereas the composition of tight configurations is always tight. The example below shows that, in most cases, a non-trivial decomposition of a tight configuration necessarily involves loose configurations.

Example 2.4. Let $\gamma_i = (C_i, \mathcal{I}_i, \varrho_i, \nu)$, where $C_i = \{c_i\}$, $\mathcal{I}_i = \{(c_i, out, c_{(i \bmod 3)+1}, in)\}$, for all $i \in [1, 3]$, $\varrho_1(c_1) = \varrho_2(c_2) = \mathsf{H}$ and $\varrho_3(c_3) = \mathsf{T}$. Then $\gamma \stackrel{\text{def}}{=} \gamma_1 \bullet \gamma_2 \bullet \gamma_3$ is the configuration from the top-left corner of Fig. 2, where the store ν is arbitrary. Note that γ_1, γ_2, and γ_3 are loose, respectively, but γ is tight. Moreover, the only way of decomposing γ into two tight subconfigurations γ_1' and γ_2' is taking $\gamma_1' \stackrel{\text{def}}{=} \gamma$ and $\gamma_2' \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \nu)$, or viceversa. ■

A configuration is changed by two types of *actions*: (a) *havoc* actions change the local states of the components by executing interactions (that trigger simultaneous transitions in different components), without changing the structure or the store, and (b) *reconfiguration* actions that change the structure, store and possibly the state map of a configuration. We refer to Fig. 2 for a depiction of havoc and reconfiguration actions. Each havoc action is the result of executing a sequence of interactions (horizontally depicted using straight double arrows), whereas each reconfiguration action (vertically depicted using snake-shaped arrows) corresponds to a statement in a reconfiguration program. The two types of actions may interleave, yielding a transition graph

[3]Since $\mathrm{dom}(\varrho_i) \subseteq C_i$, for $i = 1, 2$ and $C_1 \cap C_2 = \emptyset$, the disjointness condition is not necessary for state maps.

with a finite but unbounded (parametric) or even infinite (obtained by iteratively adding new components) set of vertices (configurations).

Formally, an *action* is a function $f : \Gamma \to \mathrm{pow}(\Gamma)^\top$, where $\mathrm{pow}(\Gamma)^\top \stackrel{\text{def}}{=} \mathrm{pow}(\Gamma) \cup \{\top\}$. The complete lattice $(\mathrm{pow}(\Gamma), \subseteq, \cup, \cap)$ is extended with a greatest element $\top$, with the conventions $S \cup \top \stackrel{\text{def}}{=} \top$ and $S \cap \top \stackrel{\text{def}}{=} S$, for each $S \in \mathrm{pow}(\Gamma)$. We consider that an action f is *disabled* in a configuration γ if and only if $f(\gamma) = \emptyset$ and that it *faults* in γ if and only if $f(\gamma) = \top$. Actions are naturally lifted to sets of configurations as $f(S) \stackrel{\text{def}}{=} \bigcup_{\gamma \in S} f(\gamma)$, for each $S \subseteq \Gamma$.

Definition 2.5. The *havoc* action $\mathfrak{h} : \Gamma \to \mathrm{pow}(\Gamma)$ is defined as $\mathfrak{h}(\gamma) \stackrel{\text{def}}{=} \{\gamma' \mid \gamma \Rightarrow^* \gamma'\}$, where $\Rightarrow^*$ is the reflexive and transitive closure of the relation $\Rightarrow \subseteq \Gamma \times \Gamma$, defined by the following rule:

$$\text{(Havoc)}\ \frac{(c_1, p_1, c_2, p_2) \in \mathcal{I} \quad c_1, c_2 \in C \quad \varrho(c_i) = q_i \quad q_i \xrightarrow{p_i} q_i', \quad \text{for all } i = 1, 2}{(C, \mathcal{I}, \varrho, \nu) \Rightarrow (C, \mathcal{I}, \varrho[c_1 \leftarrow q_1'][c_2 \leftarrow q_2'], \nu)}$$

Note that the havoc action is the result of executing any sequence of tight interactions, whereas loose interactions are simply ignored. The above definition can be generalized to multi-party interactions $(c_1, p_1, \ldots, c_n, p_n)$ with $n \geq 1$ pairwise distinct participant components $c_1, \ldots, c_n$, that fire simultaneously transitions of the behavior labeled with the ports $p_1, \ldots, p_n$, respectively. In particular, the interactions of arity $n = 1$ correspond to the local (silent) actions performed independently by a single component. To keep the presentation simple, we refrain from considering such generalizations, for the time being.

Example 2.6. Let $\gamma_i = (\{c_1, c_2, c_3\}, \{(c_i, \mathit{out}, c_{i \bmod 3+1}, \mathit{in}) \mid i \in [1, 3]\}, \varrho_i, \nu)$, for $i \in [1, 3]$ be the top-most configurations from Fig. 2, where $\varrho_1(c_1) = \varrho_1(c_2) = \mathsf{H}$, $\varrho_1(c_3) = \mathsf{T}$, $\varrho_2(c_1) = \mathsf{T}$, $\varrho_2(c_2) = \varrho_2(c_3) = \mathsf{H}$, $\varrho_3(c_1) = \varrho_3(c_3) = \mathsf{H}$, $\varrho_3(c_2) = \mathsf{T}$ and $\nu(x) = c_1$, $\nu(y) = c_2$, $\nu(z) = c_3$. Then $\mathfrak{h}(\gamma_i) = \{\gamma_1, \gamma_2, \gamma_3\}$, for all $i \in [1, 3]$. ■

2.2 The Expressiveness of the Model

Before moving on to the formal definitions of a logic describing sets of configurations (§3), a reconfiguration language and a proof system for reconfiguration programs (§4), we discuss the expressive power of the *components-behavior-interactions* model of distributed systems introduced so far, namely *what kinds of distributed algorithms can be described in our model*?

The component model (a finite-state machine representing a behavior encapsulated within an interface consisting of a set of ports) is reminiscent of I/O automata [Lynch and Tuttle 1989] and interface automata [de Alfaro and Henzinger 2001], whereas the interaction model is reminiscent of the Behavior-Interaction-Priorities (BIP) framework [Basu et al. 2006] and the extensive body of work on verification of parameterized networks (see [Bloem et al. 2015] for a survey). This model is widely used to describe multi-core and distributed concurrency: multi-party communication protocols such as mutual exclusion, flooding/notification of crowd, deadlock problems (dining philosophers/cryptographers) etc. are described using finite-state machines with rendez-vous [Bozga et al. 2020].

In addition to synchronous interactions, this model is also capable of describing asynchronous communication, via bounded message channels modeled using additional components[4]. Furthermore, *transient faults* (process delays, message losses, etc.) can be modeled as well, by nondeterministic transitions e.g., a channel component might chose to nondeterministically lose

[4]The number of messages in transit depends on the number of states in the behavior; unbounded message queues would require an extension of the model to infinite-state behaviors.

a message. In particular, having a single finite-state machine that describes the behavior of all components is not a limitation, because finitely many behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_m$ can be represented by state machines with disjoint transition graphs, the state map distinguishing between different behavior types – if $\varrho(c) = q$ and q is a state of $\mathcal{B}_i$, the value of $\varrho(c)$ can never change to a state of a different behavior $\mathcal{B}_j$, as the result of a havoc action.

On the other hand, the current model cannot describe complex distributed algorithms, such as leader election [Chang and Roberts 1979; Dolev et al. 1982], spanning tree [Kruskal 1956; Prim 1957], topological linearization [Gall et al. 2014], Byzantine consensus [Lamport et al. 1982] or Paxos parliament [Lamport 1998], due to the following limitations:

- Finite-state behavior is oblivious of the identity of the components (processes) in distributed systems of arbitrary sizes. For instance, there is no distributed algorithm over rings that can elect a leader under the assumption of anonymous processes [Chang and Roberts 1979; Dolev et al. 1982].
- Interactions between a bounded number of participants cannot describe broadcast between arbitrarily many components, as in most common consensus algorithms [Lamport 1998; Lamport et al. 1982].

We proceed in the rest of the paper under these simplifying assumptions (i.e., finite-state behavior and bounded-arity interactions), as our focus is modeling the reconfiguration aspect of a distributed system, and consider the following extensions for future work:

- *Identifiers in registers*: the behavior is described by a finite-state machine equipped with finitely many registers r holding component identifiers, that can be used to send ($p!r$) and receive ($p?r$) identifiers (p stands for a port name), perform equality ($r = r'$) and strict inequality ($r < r'$) checks, with no other relation or function on the domain of identifiers. For instance, identifier-aware behaviors are considered in [Aiswarya et al. 2015] in the context of bounded model checking i.e., verification of temporal properties (safety and liveness) under the assumption that the system has a ring topology and proceeds in a bounded number of rounds (in a round each component executes exactly one transition). Algorithms running on networks of arbitrary topologies (described by graphs, not necessarily linear) are modeled using *distributed register automata* [Bollig et al. 2019], that offer a promising lead for verifying properties of distributed systems with arbitrary/mutable networks.
- *Broadcast interactions*: interactions involving an unbounded number of component-port pairs e.g., the p_0 ports of all components except for a bounded set $c_1, \ldots, c_k$, that interact with ports $p_1, \ldots, p_k$, for a given integer constant $k \geq 0$. Broadcast interactions are described using universal quantifiers in [Bozga et al. 2020], where network topologies are specified using first-order logic. To accommodate broadcast interactions in our model, one has to redefine composition, by considering e.g., glueing of interactions, in addition to the disjoint union of configurations (Def. 2.3). Changing this definition would have a non-trivial impact on the configuration logic used to write assertions in reconfiguration proofs (§3).

Due to the separation of the behavior from the coordination aspect (i.e., network topology), adopting a richer model of behavior (e.g., register or timed automata) would impact mainly the part of the framework that deals with checking the properties (i.e., safety, liveness or havoc invariance) of a set of configurations described by a formula of the configuration logic (§3) but should not, in principle, impact the programming language nor the proof system for reconfiguration programs (§4).

3 A LOGIC OF CONFIGURATIONS

We define a *Configuration Logic* (CL) that is, an assertion language describing sets of configurations. Let $\mathbb{A}$ be a countably infinite set of predicate symbols, where $\#(\mathsf{A}) \geq 1$ denotes the arity of a

predicate symbol $\mathsf{A} \in \mathbb{A}$. The CL formulæ are inductively described by the following syntax:

$$\phi \quad ::= \quad \mathsf{true} \mid \mathsf{emp} \mid x = y \mid x@q \mid \langle x_1.p_1, x_2.p_2 \rangle \mid \mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \mid \phi * \phi \mid \phi \wedge \phi \mid \neg\phi \mid \exists x \,.\, \phi$$

where $q \in Q$, $\mathsf{A} \in \mathbb{A}$ are predicate symbols and $x, y, x_1, x_2, \ldots \in \mathbb{V}$ are variables. The atomic formulæ $x@q$, $\langle x_1.p_1, x_2.p_2 \rangle$, and $\mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})})$ are called *component*, *interaction* and *predicate atoms*, respectively. A formula is said to be *predicate-free* if it has no occurrences of predicate atoms. By $\mathrm{fv}(\phi)$ we denote the set of free variables in ϕ, that do not occur within the scope of an existential quantifier. A formula is *quantifier-free* if it has no occurrence of existential quantifiers. A *substitution* is a partial mapping $\sigma : \mathbb{V} \to \mathbb{V}$ and the formula $\phi\sigma$ is the result of replacing each free variable $x \in \mathrm{fv}(\phi) \cap \mathrm{dom}(\sigma)$ by $\sigma(x)$ in ϕ. We denote by $[x_1/y_1, \ldots, x_k/y_k]$ the substitution that replaces x_i with y_i, for all $i \in [1, k]$. We use the shorthands $\mathsf{false} \stackrel{\text{def}}{=} \neg\mathsf{true}$, $x \neq y \stackrel{\text{def}}{=} \neg x = y$, $\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\forall x \,.\, \phi_1 \stackrel{\text{def}}{=} \neg(\exists x \,.\, \neg\phi_1)$ and $x@\_ \stackrel{\text{def}}{=} \bigvee_{q \in Q} x@q$.

We distinguish the boolean ($\wedge$) from the separating ($*$) conjunction: $\phi_1 \wedge \phi_2$ means that ϕ_1 and ϕ_2 hold for the same configuration, whereas $\phi_1 * \phi_2$ means that ϕ_1 and ϕ_2 hold separately, on two disjoint parts of the same configuration. Intuitively, a formula emp describes empty configurations, with no components and interactions, $x@q$ describes a configuration with a single component, given by the store value of x, in state q, and $\langle x_1.p_1, x_2.p_2 \rangle$ describes a single interaction between ports p_1 and p_2 of the components given by the store values of x_1 and x_2, respectively. The formula $x_1@q_1 * \ldots * x_n@q_n * \langle x_1.p_1, x_2.p_2 \rangle * \ldots * \langle x_{n-1}.p_{n-1}, x_n.p_n \rangle$ describes a structure consisting of n *pairwise distinct* components, in states $q_1, \ldots, q_n$, respectively, joined by interactions between ports p_i and p_{i+1}, respectively, for all $i \in [1, n-1]$.

The CL logic is used to describe configurations of distributed systems of unbounded size, by means of predicate symbols, defined inductively by a given set of rules. As usual, the interpretation of a predicate is a set of configurations, that is a component of the least solution of the system of recursive equations defined by the set of rules. However, the existence and unicity of the least solution is subject to the monotonicity of the function induced by the set of definitions. For this reason, we avoid using negation and define the rules in a small fragment of the logic. The *symbolic configurations* are formulæ of the form $\zeta \wedge \pi$, where ζ and π are defined by the following syntax:

$$\zeta ::= \mathsf{emp} \mid x@q \mid \langle x_1.p_1, x_2.p_2 \rangle \mid \mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \mid \zeta * \zeta \qquad \pi ::= x = y \mid x \neq y \mid \pi \wedge \pi$$

The interpretation of CL formulæ is given by a semantic relation $\models_\Delta$, parameterized by a finite *set of inductive definitions* (SID) Δ, consisting of rules $\mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \leftarrow \exists y_1 \ldots y_k \,.\, \phi$, where ϕ is a symbolic configuration, such that $\mathrm{fv}(\phi) \subseteq \{x_1, \ldots, x_{\#(\mathsf{A})}\} \cup \{y_1, \ldots, y_k\}$. The relation $\models_\Delta$ is defined inductively on the structure of formulæ, as follows:

$$\begin{array}{llll}
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \mathsf{true} & \iff \mathsf{true} \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \mathsf{emp} & \iff C = \emptyset \text{ and } \mathcal{I} = \emptyset \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & x = y & \iff \nu(x) = \nu(y) \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & x@q & \iff C = \{\nu(x)\},\ \mathcal{I} = \emptyset \text{ and } \varrho(\nu(x)) = q \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \langle x_1.p_1, x_2.p_2 \rangle & \iff C = \emptyset,\ \mathcal{I} = \{(\nu(x_1), p_1, \nu(x_2), p_2)\} \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \mathsf{A}(y_1, \ldots, y_{\#(\mathsf{A})}) & \iff (C, \mathcal{I}, \varrho, \nu) \models_\Delta \xi[x_1/y_1, \ldots, x_{\#(\mathsf{A})}/y_{\#(\mathsf{A})}], \text{ for some} \\
 & & & \quad \text{rule } \mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \leftarrow \xi \text{ from } \Delta \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \phi_1 * \phi_2 & \iff \text{there exist configurations } \gamma_1 \text{ and } \gamma_2, \text{ such that} \\
 & & & \quad (C, \mathcal{I}, \varrho, \nu) = \gamma_1 \bullet \gamma_2 \text{ and } \gamma_i \models_\Delta \phi_i, \text{ for both } i = 1, 2 \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \phi_1 \wedge \phi_2 & \iff (C, \mathcal{I}, \varrho, \nu) \models_\Delta \phi_i, \text{ for both } i = 1, 2 \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \neg\phi_1 & \iff \text{not } (C, \mathcal{I}, \varrho, \nu) \models_\Delta \phi_1 \\
(C, \mathcal{I}, \varrho, \nu) & \models_\Delta & \exists x \,.\, \phi_1 & \iff (C, \mathcal{I}, \varrho, \nu[x \leftarrow c], \varrho) \models_\Delta \phi_1, \text{ for some } c \in \mathbb{C}
\end{array}$$

From now on, we consider the SID Δ to be clear from the context and write $\gamma \models \phi$ instead of $\gamma \models_\Delta \phi$. If $\gamma \models \phi$, we say that γ is a *model* of ϕ and define the set of models of ϕ as $[[\phi]] \stackrel{\text{def}}{=} \{\gamma \mid \gamma \models \phi\}$. A formula ϕ is *satisfiable* if and only if $[[\phi]] \neq \emptyset$. Given formulæ ϕ and ψ, we say that ϕ *entails* ψ if and only if $[[\phi]] \subseteq [[\psi]]$, written $\phi \models \psi$.

Example 3.1. The SID below defines chains of components and interactions, with at least $h, t \in \mathbb{N}$ components in state H and T, respectively:

$$\begin{array}{ll} \mathsf{chain}_{0,1}(x,x) \leftarrow x@\mathsf{T} & \mathsf{chain}_{h,t}(x,y) \leftarrow \exists z.\ x@\mathsf{T} * \langle x.out, z.in\rangle * \mathsf{chain}_{h,t\dot{-}1}(z,y) \\ \mathsf{chain}_{1,0}(x,x) \leftarrow x@\mathsf{H} & \mathsf{chain}_{h,t}(x,y) \leftarrow \exists z.\ x@\mathsf{H} * \langle x.out, z.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(z,y) \\ \mathsf{chain}_{0,0}(x,x) \leftarrow x@\_ & \end{array}$$

where $k\dot{-}1 \stackrel{\text{def}}{=} \max(k-1, 0)$, for all $k \in \mathbb{N}$. The configurations $(\{c_1, \ldots, c_n\}, \{(c_i, out, c_{(i \bmod n)+1}, in) \mid i \in [1, n]\}, \varrho, \nu)$ from Example 2.2 are models of the formula $\exists x \exists y\ .\ \mathsf{chain}_{0,0}(x,y) * \langle y.out, x.in\rangle$, for all $n \in \mathbb{N}$. This is because any such configuration can be decomposed into a model of $\langle y.out, x.in\rangle$ and a model of $\mathsf{chain}_{0,0}(x,y)$. The latter is either a model of $x@\_$, matching the body of the rule $\mathsf{chain}_{0,0}(x,x) \leftarrow x@\_$ if $x = y$, or a model of $\exists z.\ x@\_ * \langle x.out, z.in\rangle * \mathsf{chain}_{0,0}(z,y)$, matching the body of the rule $\mathsf{chain}_{0,0} \leftarrow \exists z.\ x@\_ * \langle x.out, z.in\rangle * \mathsf{chain}_{0,0}(z,y)$ etc. ■

3.1 The Expressiveness of CL

The CL logic is quite expressive, due to the interplay between first-order quantifiers and inductively defined predicates. For instance, the class of *cliques*, in which there is an interaction between the *out* and *in* ports of any two present components are defined by the formula:

$$\forall x \forall y\ .\ x@\_ * y@\_ * \mathsf{true} \rightarrow \langle x.out, y.in\rangle * \mathsf{true}$$

Describing cliques is an important step towards modeling the network topology and communication scheme in consensus protocols, such as Byzantine [Lamport et al. 1982] or Paxos [Lamport 1998] (though currently the model of behavior is too weak to describe such algorithms).

Connected networks, used in e.g., linearization algorithms [Gall et al. 2014], are such that there exists a path of interactions (involving e.g., the ports *out* and *in*) between each two components in the system: $connected \stackrel{\text{def}}{=} \forall x \forall y\ .\ x@\_ * y@\_ * \mathsf{true} \rightarrow reach(x,y)$, where the predicate $reach(x,y)$ is defined by the rules $reach(x,y) \leftarrow \langle x.out, y.in\rangle * \mathsf{true}$ and $reach(x,y) \leftarrow \exists z\ .\ \langle x.out, z.in\rangle * reach(z,y) * \mathsf{true}$. A system has a *cycle* if and only if it is a model of the formula $cyclic \stackrel{\text{def}}{=} \exists x\ .\ x@\_ * reach(x,x)$ and is *acyclic* if and only if it is a model of $\neg cyclic$. Acyclicity is an important property that allows to define dag and grid topologies, that are used in modeling distributed scientific computing [Foster 2002].

As suggested by work on Separation Logic (SL) [Demri et al. 2018], the price to pay for this expressivity is the inherent impossibility of having decision procedures for a fragment of CL, that combines first-order quantifiers with inductively defined predicates. A non-trivial fragment of CL that has decision procedures for satisfiability and entailment is the class of symbolic configurations [Bozga et al. 2022a]. However, modeling systems with clique or grid topologies lies beyond the expressive capability of the symbolic configurations fragment [Iosif and Zuleger 2022, §5].

Finally, CL is found to be strictly more expressive than SL. On one hand, heaps are directed graphs of fixed out-degree, that can be described by networks of components and (possibly loose) interactions. On the other hand, CL can describe sets of networks of unbounded degree, as e.g., the following definition of a star topology consisting of a controller interacting with an unbounded number of workers: $star(x) \leftarrow x@\_ * worker(x)$, $worker(x) \leftarrow \mathsf{emp}$ and $worker(x) \leftarrow \exists y\ .\ \langle x.out, y.in\rangle * y@\_ * worker(x)$. Such models are the source of a gap between SL and CL regarding the complexity of decision problems, such as entailment (see §7 for more details).

4 A LANGUAGE FOR PROGRAMMING RECONFIGURATIONS

This section defines *reconfiguration* actions that change the structure of a configuration. We distinguish between reconfigurations and havoc actions (Def. 2.5), that change configurations in orthogonal ways (see Fig. 2 for an illustration of the interplay between the two types of actions). The reconfiguration actions are the result of executing a given reconfiguration program on the distributed system at hand. For simplicity, we consider a centralized reconfiguration model, in which the reconfiguration program is executed sequentially by a designated node, that has access to the entire network. This model is currently adopted by most architecture description languages that consider reconfiguration [Bradbury et al. 2004; Butting et al. 2017]. Reasoning about distributed reconfiguration models, e.g., [Peres et al. 2019], is considered for future work.

4.1 Syntax and Operational Semantics

Reconfiguration programs, ranged over by R, are inductively defined by the following syntax:

$$\begin{array}{rcl} \mathsf{R} & ::= & \mathsf{new}(q,x) \mid \mathsf{delete}(x) \mid \mathsf{connect}(x_1.p_1, x_2.p_2) \mid \mathsf{disconnect}(x_1.p_1, x_2.p_2) \\ & & \mid \mathsf{with}\ x_1,\ldots,x_k : \theta\ \mathsf{do}\ \mathsf{R}_1\ \mathsf{od} \mid \mathsf{R}_1;\mathsf{R}_2 \mid \mathsf{R}_1 + \mathsf{R}_2 \mid \mathsf{R}_1^* \end{array}$$

where $q \in Q$ is a state, $x, x_1, x_2, \ldots \in \mathbb{V}$ are program variables and θ is a *predicate-free quantifier-free* formula of the CL logic, called a *trigger*.

The *primitive commands* are $\mathsf{new}(q,x)$ and $\mathsf{delete}(x)$, that create and delete a component (the newly created component is set to execute from state q) given by the store value of x, $\mathsf{connect}(x_1.p_1, x_2.p_2)$ and $\mathsf{disconnect}(x_1.p_1, x_2.p_2)$, that create and delete an interaction, between the ports p_1 and p_2 of the components given by the store values of x_1 and x_2, respectively. We denote by $\mathfrak{P}$ the set of primitive commands.

A *conditional* is a program of the form $(\mathsf{with}\ x_1,\ldots,x_k : \theta\ \mathsf{do}\ \mathsf{R}\ \mathsf{od})$ that performs the following steps, *with no havoc action (Def. 2.5) in between the first and second steps* below:

(1) maps the variables $x_1,\ldots,x_k$ to some components $c_1,\ldots,c_k \in \mathbb{C}$ such that the configuration after the assignment contains a model of the trigger θ; the conditional is disabled if the current configuration is not a model of $\exists x_1 \ldots \exists x_k \,.\, \theta * \mathsf{true}$,
(2) launches the first command of the program R on this configuration, and
(3) continues with the remainder of R, in interleaving with havoc actions;
(4) upon completion of R, the values of $x_1,\ldots,x_k$ are forgotten.

To avoid technical complications, we assume that nested conditionals use pairwise disjoint tuples of variables; every program can be statically changed to meet this condition, by renaming variables. Note that the trigger θ of a conditional $(\mathsf{with}\ x_1,\ldots,x_k : \theta\ \mathsf{do}\ \mathsf{R}\ \mathsf{od})$ has no quantifiers nor predicate atoms, which means that the overall number of components and interactions in a model of θ is polynomially bounded by the size of (number of symbols needed to represent) θ. Intuitively, this means that the part of the system (matched by θ) to which the reconfiguration is applied is relatively small, thus the procedure that evaluates the trigger can be easily implemented in a distributed environment, as e.g., consensus between a small number of neighbouring components.

The sequential composition $\mathsf{R}_1;\mathsf{R}_2$ executes R_1 followed by R_2, with zero or more interactions firing in between. This is because, even though being sequential, a reconfiguration program runs in parallel with the state changes that occur as a result of firing the interactions. Last, $\mathsf{R}_1 + \mathsf{R}_2$ executes either R_1 or R_2, and R^* executes R zero or more times in sequence, nondeterministically.

It is worth pointing out that the reconfiguration language does not have explicit assignments between variables. As a matter of fact, the conditionals are the only constructs that nondeterministically bind variables to indices that satisfy a given logical constraint. This design choice sustains the view of a distributed system as a cloud of components and interactions in which reconfigurations can

$$\frac{c \in \mathbb{C} \setminus C}{\mathsf{new}(q,x) : (C, \mathcal{I}, \varrho, \nu) \rightsquigarrow (C \cup \{c\}, \mathcal{I}, \varrho[c \leftarrow q], \nu[x \leftarrow c])}$$

$$\frac{\nu(x) \in C}{\mathsf{delete}(x) : (C, \mathcal{I}, \varrho, \nu) \rightsquigarrow (C \setminus \{\nu(x)\}, \mathcal{I}, \varrho, \nu)} \qquad \frac{\nu(x) \notin C}{\mathsf{delete}(x) : (C, \mathcal{I}, \varrho, \nu) \lightning}$$

$$\frac{}{\mathsf{connect}(x_1.p_1, x_2.p_2) : (C, \mathcal{I}, \varrho, \nu) \rightsquigarrow (C, \mathcal{I} \cup \{(\nu(x_1), p_1, \nu(x_2), p_2)\}, \varrho, \nu)}$$

$$\frac{(\nu(x_1), p_1, \nu(x_2), p_2) \in \mathcal{I}}{\mathsf{disconnect}(x_1.p_1, x_2.p_2) : (C, \mathcal{I}, \varrho, \nu) \rightsquigarrow (C, \mathcal{I} \setminus \{(\nu(x_1), p_1, \nu(x_2), p_2)\}, \varrho, \nu)}$$

$$\frac{(\nu(x_1), p_1, \nu(x_2), p_2) \notin \mathcal{I}}{\mathsf{disconnect}(x_1.p_1, x_2.p_2) : (C, \mathcal{I}, \varrho, \nu) \lightning}$$

$$\frac{\begin{array}{l} c_1, c_1', \ldots, c_k, c_k' \in \mathbb{C} \quad (C, \mathcal{I}, \varrho, \nu[x_1 \leftarrow c_1, \ldots, x_k \leftarrow c_k]) \models \xi * \mathsf{true} \\ \mathsf{R} : (C, \mathcal{I}, \varrho, \nu[x_1 \leftarrow c_1, \ldots, x_k \leftarrow c_k]) \rightsquigarrow (C', \mathcal{I}', \varrho', \nu') \end{array}}{\mathsf{with}\ x_1, \ldots, x_k : \xi\ \mathsf{do}\ \mathsf{R}\ \mathsf{od} : (C, \mathcal{I}, \varrho, \nu) \rightsquigarrow (C', \mathcal{I}', \varrho', \nu'[x_1 \leftarrow c_1', \ldots, x_k \leftarrow c_k'])}$$

$$\frac{\mathsf{R}_1 : \gamma \rightsquigarrow \gamma_0 \quad \gamma_1 \in \mathfrak{h}(\gamma_0) \quad \mathsf{R}_2 : \gamma_1 \rightsquigarrow \gamma'}{\mathsf{R}_1; \mathsf{R}_2 : \gamma \rightsquigarrow \gamma'} \qquad \frac{\mathsf{R}_1 : \gamma \rightsquigarrow \gamma'}{\mathsf{R}_1 + \mathsf{R}_2 : \gamma \rightsquigarrow \gamma'}$$

$$\frac{\mathsf{R}^n : \gamma \rightsquigarrow \gamma'}{\mathsf{R}^* : \gamma \rightsquigarrow \gamma'}\ , \mathsf{R}^n = \begin{cases} \mathsf{R}^{n-1}; \mathsf{R} & \text{if } n \geq 1 \\ \mathsf{skip} & \text{if } n = 0 \end{cases}$$

Fig. 3. Operational Semantics of the Reconfiguration Language

occur anywhere a local condition is met. In other words, we do not need variable assignments to traverse the architecture — the program works rather by identifying a part of the system that matches a small pattern, and applying the reconfiguration locally to that subsystem. For instance, a typical pattern for writing reconfiguration programs is $(\mathsf{with}\ \mathbf{x}_1 : \theta_1\ \mathsf{do}\ \mathsf{R}_1\ \mathsf{od} + \ldots + \mathsf{with}\ \mathbf{x}_k : \theta_k\ \mathsf{do}\ \mathsf{R}_k\ \mathsf{od})^*$, where $\mathsf{R}_1, \ldots, \mathsf{R}_k$ are loop-free sequential compositions of primitive commands. This program continuously choses a reconfiguration sequence R_i nondeterministically and either applies it on a small part of the configuration that satisfies θ_i, or does nothing, if no such subconfiguration exists within the current configuration.

The operational semantics of reconfiguration programs is given by the structural rules in Fig. 3, that define the judgements $\mathsf{R} : \gamma \rightsquigarrow \gamma'$ and $\mathsf{R} : \gamma \lightning$, where γ and γ' are configurations and R is a program. Intuitively, $\mathsf{R} : \gamma \rightsquigarrow \gamma'$ means that γ' is a successor of γ following the execution of R and $\mathsf{R} : \gamma \lightning$ means that R faults in γ. The semantics of a program R is the action $\langle\!\langle \mathsf{R} \rangle\!\rangle : \Gamma \to \mathrm{pow}(\Gamma)^\top$, defined as $\langle\!\langle \mathsf{R} \rangle\!\rangle(\gamma) \stackrel{\text{def}}{=} \top$, if $\mathsf{R} : \gamma \lightning$ and $\langle\!\langle \mathsf{R} \rangle\!\rangle(\gamma) \stackrel{\text{def}}{=} \{\gamma' \mid \mathsf{R} : \gamma \rightsquigarrow \gamma'\}$, otherwise. The only primitive commands that may fault are $\mathsf{delete}(x)$ and $\mathsf{disconnect}(x_1.p_1, x_2.p_2)$; for both, the premisses of the faulty rules are disjoint from the ones for normal termination, thus the action $\langle\!\langle \mathsf{R} \rangle\!\rangle$ is properly defined for all programs R. Notice that the rule for sequential composition uses the havoc action $\mathfrak{h}$ in the premiss, thus capturing the interleaving of havoc state changes and reconfiguration actions.

4.2 Reconfiguration Proof System

To reason about the correctness properties of reconfiguration programs, we introduce a Hoare-style proof system consisting of a set of axioms that formalize the primitive commands (Fig. 4a), a set of inference rules for the composite programs (Fig. 4b) and a set of structural rules (Fig. 4c). The judgements are Hoare triples $\{\phi\}\ \mathsf{R}\ \{\psi\}$, where ϕ and ψ (called pre- and postcondition, respectively) are CL formulæ. The triple $\{\phi\}\ \mathsf{R}\ \{\psi\}$ is *valid*, written $\models \{\phi\}\ \mathsf{R}\ \{\psi\}$, if and only if $\langle\!\langle \mathsf{R} \rangle\!\rangle([\![\phi]\!]) \subseteq [\![\psi]\!]$. Note that this semantics of Hoare triples corresponds to partial correctness i.e., correctness under the assumption that R terminates, without enforcing the latter condition.

$$\frac{}{\{\mathsf{emp}\}\ \mathsf{new}(q,x)\ \{x@q\}} \qquad \frac{}{\{x@\_\}\ \mathsf{delete}(x)\ \{\mathsf{emp}\}}$$

$$\frac{}{\{\mathsf{emp}\}\ \mathsf{connect}(x_1.p_1, x_2.p_2)\ \{\langle x_1.p_1, x_2.p_2\rangle\}} \qquad \frac{}{\{\langle x_1.p_1, x_2.p_2\rangle\}\ \mathsf{disconnect}(x_1.p_1, x_2.p_2)\ \{\mathsf{emp}\}}$$

a. Axioms for Primitive Commands

$$\frac{\{\phi \wedge (\theta * \mathsf{true})\}\ \mathsf{R}\ \{\psi\}}{\{\phi\}\ \mathsf{with}\ x_1,\ldots,x_k : \theta\ \mathsf{do}\ \mathsf{R}\ \mathsf{od}\ \{\exists x_1 \ldots \exists x_k\ .\ \psi\}}\ \mathsf{fv}(\phi) \cap \{x_1,\ldots,x_k\} = \emptyset$$

$$\frac{\{\phi\}\ \mathsf{R}_1\ \{\xi\} \qquad \{\xi\}\ \mathsf{R}_2\ \{\psi\}}{\{\phi\}\ \mathsf{R}_1; \mathsf{R}_2\ \{\psi\}}\ \mathfrak{h}([\![\xi]\!]) \subseteq [\![\xi]\!]$$

$$\frac{\{\phi\}\ \mathsf{R}_1\ \{\psi\} \qquad \{\phi\}\ \mathsf{R}_2\ \{\psi\}}{\{\phi\}\ \mathsf{R}_1 + \mathsf{R}_2\ \{\psi\}} \qquad \frac{\{\phi\}\ \mathsf{R}\ \{\phi\}}{\{\phi\}\ \mathsf{R}^*\ \{\phi\}}\ \mathfrak{h}([\![\phi]\!]) \subseteq [\![\phi]\!]$$

b. Inference Rules for Programs

$$\frac{\{\phi_i\}\ \mathsf{R}\ \{\psi_i\} \mid i \in [1,k]}{\{\bigvee_{i=1}^{k} \phi_i\}\ \mathsf{R}\ \{\bigvee_{i=1}^{k} \psi_i\}} \qquad \frac{\{\phi_i\}\ \mathsf{R}\ \{\psi_i\} \mid i \in [1,k]}{\{\bigwedge_{i=1}^{k} \phi_i\}\ \mathsf{R}\ \{\bigwedge_{i=1}^{k} \psi_i\}}$$

$$\frac{\{\phi'\}\ \mathsf{R}\ \{\psi'\} \quad \phi \models \phi' \quad \psi' \models \psi}{\{\phi\}\ \mathsf{R}\ \{\psi\}} \qquad \frac{\{\phi\}\ \mathsf{R}\ \{\psi\}}{\{\phi * F\}\ \mathsf{R}\ \{\psi * F\}}\ \begin{array}{l}\mathsf{R} \in \mathfrak{L} \\ \mathsf{modif}(\mathsf{R}) \cap \mathsf{fv}(F) = \emptyset\end{array}$$

c. Structural Inference Rules

Fig. 4. Proof System for the Reconfiguration Language

Moreover, a triple is valid only if the program does not fault on any model of the precondition i.e., an invalid Hoare triple $\{\phi\}\ \mathsf{R}\ \{\psi\}$ cannot distinguish between $\langle\!\langle \mathsf{R} \rangle\!\rangle([\![\phi]\!]) \not\subseteq [\![\psi]\!]$ (non-faulting incorrectness) and $\langle\!\langle \mathsf{R} \rangle\!\rangle([\![\phi]\!]) = \top$ (faulting).

The axioms (Fig. 4a) give the *local specifications* of the primitive commands in the language by Hoare triples whose preconditions describe only those resources (components and interactions) necessary to avoid faulting. In particular, $\mathsf{delete}(x)$ and $\mathsf{disconnect}(x_1.p_1, x_2.p_2)$ require a single component $x@\_$ and an interaction $\langle x_1.p_1, x_2.p_2\rangle$ to avoid faulting, respectively. The rules for sequential composition and iteration (Fig 4b) use the following semantic side condition, based on the havoc action (Def. 2.5):

Definition 4.1. A formula ϕ is *havoc invariant* if and only if $\mathfrak{h}([\![\phi]\!]) \subseteq [\![\phi]\!]$.

Note that the dual inclusion $[\![\phi]\!] \subseteq \mathfrak{h}([\![\phi]\!])$ always holds, because $\mathfrak{h}$ is the reflexive and transitive closure of the $\Rightarrow$ relation (Def. 2.5). Since havoc invariance is required to prove the validity of Hoare triples involving sequential composition, it is important to have a way of checking havoc invariance. We describe a proof system for such havoc queries in §5. Moreover, the side condition of the *consequence rule* (Fig. 4c left) consists of two entailments, that are discharged by an external decision procedure (discussed in §7).

The *frame rule* (Fig. 4c bottom-right) allows to apply the specification of a *local program*, defined below, to a set of configurations that may contain more resources (components and interactions) than the ones asserted by the precondition. Intuitively, a local program requires a bounded amount of components and interactions to avoid faulting and, moreover, it only changes the configuration of the local subsystem, not affecting the entire system's configuration. Formally, the set $\mathfrak{L}$ of local programs is the least set that contains the primitive commands $\mathfrak{P}$ and is closed under the application of the following rules:

$\mathsf{R} \in \mathfrak{L} \Rightarrow \mathsf{with}\ \mathbf{x} : \pi\ \mathsf{do}\ \mathsf{R}\ \mathsf{od} \in \mathfrak{L}$, if π a conjunction of (dis-)equalities $\quad \mathsf{R}_1, \mathsf{R}_2 \in \mathfrak{L} \Rightarrow \mathsf{R}_1 + \mathsf{R}_2 \in \mathfrak{L}$

The extra resources, not required to execute a local program, are specified by a frame F, whose free variables are not modified by the local program R. Formally, the set of variables modified by a local program $\mathsf{R} \in \mathfrak{L}$ is defined inductively on its structure:

$$\mathsf{modif}(\mathsf{new}(q,x)) \stackrel{\text{def}}{=} \{x\} \qquad \mathsf{modif}(\mathsf{R}) \stackrel{\text{def}}{=} \emptyset, \text{ for all } \mathsf{R} \in \mathfrak{P} \setminus \{\mathsf{new}(q,x) \mid q \in Q, x \in \mathbb{V}\}$$
$$\mathsf{modif}(\mathsf{with}\ \mathbf{x} : \theta\ \mathsf{do}\ \mathsf{R}\ \mathsf{od}) \stackrel{\text{def}}{=} \mathbf{x} \cup \mathsf{modif}(\mathsf{R}) \qquad \mathsf{modif}(\mathsf{R}_1 + \mathsf{R}_2) \stackrel{\text{def}}{=} \mathsf{modif}(\mathsf{R}_1) \cup \mathsf{modif}(\mathsf{R}_2)$$

The frame rule is sound only for programs whose semantics are *local actions*, defined below:

Definition 4.2 (Locality). Given a set of variables $X \subseteq \mathbb{V}$, an action $f : \Gamma \to \mathsf{pow}(\Gamma)^\top$ is *local for* X if and only if $f(\gamma_1 \bullet \gamma_2) \subseteq f(\gamma_1) \bullet \{\gamma_2\}\!\uparrow^X$ for all $\gamma_1, \gamma_2 \in \Gamma$, where, for any set S of configurations:

$$S\!\uparrow^X \stackrel{\text{def}}{=} \{(C, \mathcal{I}, \varrho', \nu') \mid (C, \mathcal{I}, \varrho, \nu) \in S, \forall x \in \mathbb{V} \setminus X \,.\, \nu'(x) = \nu(x),\ \forall c \in \mathbb{C} \setminus \nu(X) \,.\, \varrho'(c) = \varrho(c)\}\,.$$

An action f is *local* if and only if it is local for the empty set of variables.

An action that is local for a set of variables X allows for the change of the store values of the variables in X and the states of the components indexed by those values, only. Essentially, $\mathsf{new}(q,x)$ is local for $\{x\}$, because the fresh index associated to x is nondeterministically chosen and the state is q, whereas the other primitive commands are local, in general. The semantics of every local program $\mathsf{R} \in \mathfrak{L}$ is a local action, as shown below:

Lemma 4.3. *For every program* $\mathsf{R} \in \mathfrak{L}$*, the action* $\langle\!\langle \mathsf{R} \rangle\!\rangle$ *is local for* $\mathsf{modif}(\mathsf{R})$.

Moreover, $\mathfrak{L}$ is precisely the set of programs with local semantics, as conditionals and sequential compositions (hence also iterations) are not local, in general:

Example 4.4. To understand why $\mathfrak{L}$ is precisely the set of local commands, consider the programs:

- $(\mathsf{with}\ x : x@q\ \mathsf{do}\ \mathsf{delete}(x)\ \mathsf{od})$ is not local because, letting γ_1 be a configuration with zero components and γ_2 be a configuration with one component in state q, we have:

  $\langle\!\langle \mathsf{with}\ x : x@q\ \mathsf{do}\ \mathsf{delete}(x)\ \mathsf{od} \rangle\!\rangle(\gamma_1 \bullet \gamma_2) = \langle\!\langle \mathsf{with}\ x : x@q\ \mathsf{do}\ \mathsf{delete}(x)\ \mathsf{od} \rangle\!\rangle(\gamma_2) = \{\gamma_1\}$
  whereas $\langle\!\langle \mathsf{with}\ x : x@q\ \mathsf{do}\ \mathsf{delete}(x)\ \mathsf{od} \rangle\!\rangle(\gamma_1) \bullet \{\gamma_2\} = \emptyset \bullet \{\gamma_2\} = \emptyset$.

- $(\mathsf{skip}; \mathsf{skip})$ is not local because, considering the system from Fig. 1, if we take γ_1 and γ_2, such that $\gamma_1 \models x@\mathsf{T}$ and $\gamma_2 \models \langle x.out, y.in \rangle * y@\mathsf{H}$, we have:

  $\langle\!\langle \mathsf{skip}; \mathsf{skip} \rangle\!\rangle(\gamma_1 \bullet \gamma_2) = [\![x@\mathsf{T} * \langle x.out, y.in \rangle * y@\mathsf{H}]\!] \ \cup\ [\![x@\mathsf{H} * \langle x.out, y.in \rangle * y@\mathsf{T}]\!]$
  whereas $\langle\!\langle \mathsf{skip}; \mathsf{skip} \rangle\!\rangle(\gamma_1) \bullet \{\gamma_2\} = [\![x@\mathsf{T} * \langle x.out, y.in \rangle * y@\mathsf{H}]\!]$ ■

We write $\vdash \{\phi\}\ \mathsf{R}\ \{\psi\}$ if and only if $\{\phi\}\ \mathsf{R}\ \{\psi\}$ can be derived from the axioms using the inference rules from Fig. 4 and show the soundness of the proof system in the following. The next lemma gives sufficient conditions for the soundness of the axioms (Fig. 4a):

Lemma 4.5. *For each axiom* $\{\phi\}\ \mathsf{R}\ \{\psi\}$*, where* $\mathsf{R} \in \mathfrak{P}$ *is primitive, we have* $\langle\!\langle \mathsf{R} \rangle\!\rangle([\![\phi]\!]) = [\![\psi]\!]$.

The soundness of the proof system in Fig. 4 follows from the soundness of each inference rule:

Theorem 4.6. *For any Hoare triple* $\{\phi\}\ \mathsf{R}\ \{\psi\}$*, if* $\vdash \{\phi\}\ \mathsf{R}\ \{\psi\}$ *then* $\models \{\phi\}\ \mathsf{R}\ \{\psi\}$.

As an optimization, reconfiguration proofs can often be simplified, by safely skipping the check of one or more havoc invariance side conditions of sequential compositions, as explained below.

Definition 4.7. A program of the form $\mathsf{disconnect}(x_1.p_1, x_1'.p_1')$; ... $\mathsf{disconnect}(x_k.p_k, x_k'.p_k')$; $\mathsf{connect}(x_{k+1}.p_{k+1}, x_{k+1}'.p_{k+1}')$; ... $\mathsf{connect}(x_\ell.p_\ell, x_\ell'.p_\ell')$ is said to be a *single reversal program.*

Single reversal programs first disconnect components and then reconnect them in a different way. For such programs, only the first and last application of the sequential composition rule require checking havoc invariance:

Proposition 4.8. *Let* $\mathsf{R} = \mathsf{disconnect}(x_1.p_1, x_1'.p_1'); \ldots \mathsf{disconnect}(x_k.p_k, x_k'.p_k'); \mathsf{connect}(x_{k+1}.p_{k+1}, x_{k+1}'.p_{k+1}'); \ldots \mathsf{connect}(x_\ell.p_\ell, x_\ell'.p_\ell')$ *be a single reversal program. If* $\phi_0, \ldots, \phi_\ell$ *are CL formulæ, such that:*

- $\models \{\phi_{i-1}\}\ \mathsf{disconnect}(x_i.p_i, x_i'.p_i')\ \{\phi_i\}$, *for all* $i \in [1, k]$,
- $\models \{\phi_{j-1}\}\ \mathsf{connect}(x_j.p_j, x_j'.p_j')\ \{\phi_j\}$, *for all* $j \in [k+1, \ell]$, *and*
- ϕ_1 *and* $\phi_{\ell-1}$ *are havoc invariant,*

then we have $\models \{\phi_0\}\ \mathsf{R}\ \{\phi_\ell\}$.

4.3 Examples of Reconfiguration Proofs

We prove that the outcome of the reconfiguration program from Fig. 1 (Listing 2), started in a token ring configuration with at least two components in state H and at least one in state T, is a token ring with at least one component in each state. The pre- and postcondition are $\exists x \exists y\ .\ \mathsf{chain}_{2,1}(x, y) * \langle y.out, x.in\rangle$ and $\exists x \exists y\ .\ \mathsf{chain}_{1,1}(x, y) * \langle y.out, x.in\rangle$, respectively, with the definitions of $\mathsf{chain}_{h,t}(x, y)$ given in Example 3.1, for all constants $h, t \in \mathbb{N}$.

$\{\exists x \exists y\ .\ \mathsf{chain}_{2,1}(x, y) * \langle y.out, x.in\rangle\}$
with $x, y, z : \langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle$ **do**
$\{(\exists x \exists y\ .\ \mathsf{chain}_{2,1}(x, y) * \langle y.out, x.in\rangle) \wedge (\langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{true})\}$ $(\star)$
$\{\langle x.out, y.in\rangle * \boxed{y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{chain}_{1,1}(z, x)}\}$
`disconnect(x.out, y.in);`
$\{\boxed{y@\mathsf{H}} * \langle y.out, z.in\rangle * \boxed{\mathsf{chain}_{1,1}(z, x)}\}$ $(\sharp)$
`disconnect(y.out, z.in);`
$\{y@\mathsf{H} * \boxed{\mathsf{chain}_{1,1}(z, x)}\}$ $(\sharp)$
`delete(y);`
$\{\boxed{\mathsf{chain}_{1,1}(z, x)}\}$ $(\sharp)$
`connect(x.out, z.in)`
$\{\mathsf{chain}_{1,1}(z, x) * \langle x.out, z.in\rangle\}$
od
$\{\exists x \exists y\ .\ \mathsf{chain}_{1,1}(x, y) * \langle y.out, x.in\rangle\}$

The inference rule for conditional programs sets up the precondition $(\star)$ for the body of the conditional. This formula is equivalent to $\langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{chain}_{1,1}(z, x)$. To understand this point, we derive from $(\star)$ the equivalences:

$$\begin{aligned}&(\exists x \exists y\,.\,\mathsf{chain}_{2,1}(x, y) * \langle y.out, x.in\rangle) \wedge (\langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{true})\\ &\equiv \exists \overline{x} \exists \overline{y} \exists \overline{z}\,.\,\langle \overline{x}.out, \overline{y}.in\rangle * \overline{y}@\mathsf{H} * \langle \overline{y}.out, \overline{z}.in\rangle * \mathsf{chain}_{1,1}(\overline{z}, \overline{x}) \wedge\\ &\quad (\langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{true}) \equiv \langle x.out, y.in\rangle * y@\mathsf{H} * \langle y.out, z.in\rangle * \mathsf{chain}_{1,1}(z, x)\end{aligned}$$

where the first step can be proven by entailment checking (discussed in §7). The following four annotations above are obtained by applications of the axioms and the frame rule (the frame formulæ are displayed within boxes). The sequential composition rule is applied by proving first that the annotations marked as $(\sharp)$ are havoc invariant (the proof is given later in §5.4).

We have considered the reconfiguration program from Fig. 1 (Listing 2) which deletes a component from a token ring. The dual operation is the addition of a new component. Here the precondition states that the system is a valid token ring, with at least one component in state H and at least another one in state T. We prove that the execution of the dual program yields a token ring with at least two components in state H, as the new component is added without a token.

$\{\exists x \exists y \,.\, \mathsf{chain}_{1,1}(x,y) * \langle y.out, x.in\rangle\}$
with $x, z : \langle x.out, z.in\rangle$ **do**
$\{(\exists x \exists y \,.\, \mathsf{chain}_{1,1}(x,y) * \langle y.out, x.in\rangle) \wedge (\langle x.out, z.in\rangle * \mathsf{true})\}$ $(\star)$
$\{\langle x.out, z.in\rangle * \boxed{\mathsf{chain}_{1,1}(z,x)}\}$
`disconnect(x.out, z.in);`
$\{\mathsf{chain}_{1,1}(z,x)\}$ $(\sharp)$
`new(H,y);`
$\{y@\mathsf{H} * \boxed{\mathsf{chain}_{1,1}(z,x)}\}$ $(\sharp)$
`connect(y.out,z.in);`
$\{\boxed{y@\mathsf{H}} * \langle y.out, z.in\rangle * \boxed{\mathsf{chain}_{1,1}(z,x)}\}$
$\{\mathsf{chain}_{2,1}(y,x)\}$ $(\sharp)$
`connect(x.out,y.in)`
$\{\mathsf{chain}_{2,1}(y,x) * \langle x.out, y.in\rangle\}$
od
$\{\exists x \exists y \,.\, \mathsf{chain}_{2,1}(x,y) * \langle y.out, x.in\rangle\}$

The annotation $(\star)$ is given by the inference rule for conditional programs. Then we can derive the equivalence $(\exists x \exists y \,.\, \mathsf{chain}_{1,1}(x,y) * \langle y.out, x.in\rangle) \wedge (\langle x.out, z.in\rangle * \mathsf{true}) \equiv \langle x.out, z.in\rangle * \mathsf{chain}_{1,1}(z,x)$. In the subsequent lines, some axioms and the frame rule are applied (the frame is displayed in the postconditions of the commands within the boxes). The annotations marked as $(\sharp)$ must be shown to be havoc invariant and then the sequential composition rule is applied to complete the proof.

5 THE HAVOC PROOF SYSTEM

This section describes a set of axioms and inference rules for proving the validity of havoc invariance queries of the form $\mathfrak{h}([\![\phi]\!]) \subseteq [\![\phi]\!]$, where ϕ is a CL formula interpreted over a given SID and $\mathfrak{h}$ is the havoc action (Def. 2.5). Such a query is valid if and only if the result of applying any sequence of interactions on a model of ϕ is again a model of ϕ (Def. 4.1). Havoc invariance queries occur as side conditions in the rules for sequential composition and iteration (Fig. 4b) of reconfiguration programs. Thus, having a proof system for havoc invariance is crucial for the applicability of the rules in Fig. 4 to obtain proofs of reconfiguration programs.

The havoc proof system uses a compositional rule, able to split a query of the form $\mathfrak{h}([\![\phi_1 * \phi_2]\!]) \subseteq [\![\psi_1 * \psi_2]\!]$ into two queries $\mathfrak{h}([\![\phi_i * \mathcal{F}_i]\!]) \subseteq [\![\psi_i * \mathcal{F}_i]\!]$, where each *frontier formula* $\mathcal{F}_i$ defines a set of interactions that over-approximate the effect of executing the system described by ϕ_{3-i} (resp. ψ_{3-i}) over the one described by ϕ_i (resp. ψ_i), for $i = 1, 2$. In principle, the frontier formulæ ($\mathcal{F}_1$ and $\mathcal{F}_2$) can be understood as describing the interference between parallel actions in an assume/rely guarantee-style parallel composition rule [Jones 1981; Owicki and Gries 1978]. In particular, since the frontier formulæ only describe interactions and carry no state information whatsoever, such assumptions about events triggered by the environment are reminiscent of compositional reasoning about input/output automata [Chilton et al. 2014].

Compositional reasoning about havoc actions requires the following relaxation of the definition of havoc state changes (Def. 2.5), by allowing the firing of loose, in addition to tight interactions. The relaxation from tight to loose interaction semantics is needed in order to reason about partial systems, in which some end of an interaction is controlled by an external environment. This loose semantics considers thus that these loose ends might get active at any time.

Definition 5.1. The following rules define a relation $\overset{(c_1,p_1,c_2,p_2)}{\dashrightarrow} \subseteq \Gamma \times \Gamma$, parameterized by a given interaction (c_1, p_1, c_2, p_2):

$$\text{(Loose)}\ \frac{(c_1,p_1,c_2,p_2) \in \mathcal{I} \quad c_i \in C, c_{3-i} \notin C \quad \varrho(c_i) = q_i \quad q_i \xrightarrow{p_i} q_i'}{(C,\mathcal{I},\varrho,\nu) \overset{(c_1,p_1,c_2,p_2)}{\dashrightarrow} (C,\mathcal{I},\varrho[c_i \leftarrow q_i'],\nu)}\ i = 1,2$$

$$\text{(Tight)}\ \frac{(c_1,p_1,c_2,p_2) \in \mathcal{I} \quad c_1 \neq c_2 \in C \quad \varrho(c_i) = q_i \quad q_i \xrightarrow{p_i} q_i',\ i = 1,2}{(C,\mathcal{I},\varrho,\nu) \overset{(c_1,p_1,c_2,p_2)}{\dashrightarrow} (C,\mathcal{I},\varrho[c_1 \leftarrow q_1'][c_2 \leftarrow q_2'],\nu)}$$

For a sequence $w = i_1 \dots i_n$ of interactions, we define $\overset{w}{\dashrightarrow}$ to be the composition of $\overset{i_1}{\dashrightarrow}, \dots, \overset{i_n}{\dashrightarrow}$, assumed to be the identity relation, if w is empty.

The difference with Def. 2.5 is that only the states of the components from the configuration are changed according to the transitions in the behavior. This more relaxed definition matches the intuition of partial systems in which certain interactions may be controlled by an external environment; those interactions are conservatively assumed to fire anytime they are enabled by the components of the current structure, independently of the environment.

Example 5.2. (contd. from Example 2.6) Let $\widetilde{\gamma_i} = (\{c_2, c_3\}, \{(c_i, out, c_{i \bmod 3+1}, in) \mid i \in [1,3]\}, \varrho_i, \nu)$, for $i \in [1,3]$ be the top-most configurations from Fig. 2 without the c_1 component, where $\varrho_1(c_2) = \mathsf{H}$, $\varrho_1(c_3) = \mathsf{T}$, $\varrho_2(c_2) = \varrho_2(c_3) = \mathsf{H}$, $\varrho_3(c_2) = \mathsf{T}$, $\varrho_3(c_3) = \mathsf{H}$. Then, by executing the loose interactions (c_3, out, c_1, in) and (c_1, out, c_2, in) from γ_1, we obtain $\widetilde{\gamma_1} \overset{(c_3,out,c_1,in)}{\dashrightarrow} \widetilde{\gamma_2} \overset{(c_1,out,c_2,in)}{\dashrightarrow} \widetilde{\gamma_3}$. Executing the tight interaction (c_2, out, c_3, in) from $\widetilde{\gamma_3}$ leads back to $\widetilde{\gamma_1}$ i.e., $\widetilde{\gamma_3} \overset{(c_2,out,c_3,in)}{\dashrightarrow} \widetilde{\gamma_1}$. ■

5.1 Regular Expressions

Proving the validity of a havoc query $\mathfrak{h}([[\phi]]) \subseteq [[\psi]]$ involves reasoning about the sequences of interactions that define the outcome of the havoc action. We specify languages of such sequences using extended regular expressions, defined inductively by the following syntax:

$$\mathsf{L} ::= \epsilon \mid \Sigma[\alpha] \mid \mathsf{L} \cdot \mathsf{L} \mid \mathsf{L} \cup \mathsf{L} \mid \mathsf{L}^* \mid \mathsf{L} \bowtie_{\eta,\eta} \mathsf{L}$$

where ϵ denotes the empty string, $\Sigma[\alpha]$ is an *alphabet symbol* associated with either an interaction atom or a predicate atom α and $\cdot$, $\cup$ and * are the usual concatenation, union and Kleene star. By $\mathsf{L}_1 \bowtie_{\eta_1,\eta_2} \mathsf{L}_2$ we denote the interleaving (zip) product of the languages described by L_1 and L_2 with respect to the sets η_1 and η_2 of alphabet symbols of the form $\Sigma[\alpha]$, respectively.

The *language* of a regular expression L in a configuration $\gamma = (C, \mathcal{I}, \varrho, \nu)$ is defined below:

$$\begin{array}{ll}
\langle\!\langle \epsilon \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \{\epsilon\} & \langle\!\langle \Sigma[\alpha] \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \bigcup\{\mathcal{I} \mid (C, \mathcal{I}, \varrho, \nu) \sqsubseteq \gamma,\ (C, \mathcal{I}, \varrho, \nu) \models \alpha\} \\
\langle\!\langle \mathsf{L}_1 \cup \mathsf{L}_2 \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \langle\!\langle \mathsf{L}_1 \rangle\!\rangle(\gamma) \cup \langle\!\langle \mathsf{L}_2 \rangle\!\rangle(\gamma) & \langle\!\langle \mathsf{L}_1 \cdot \mathsf{L}_2 \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \{w_1 w_2 \mid w_i \in \langle\!\langle \mathsf{L}_i \rangle\!\rangle(\gamma),\ i = 1,2\} \\
\langle\!\langle \mathsf{L}^* \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \bigcup_{i \geq 0} \langle\!\langle \mathsf{L}^i \rangle\!\rangle(\gamma) & \langle\!\langle \mathsf{L}_1 \bowtie_{\eta_1,\eta_2} \mathsf{L}_2 \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \{w \mid w \downarrow_{\langle \eta_i \rangle(\gamma)} \in \langle\!\langle \mathsf{L}_i \rangle\!\rangle(\gamma),\ i = 1,2\}
\end{array}$$

where $\langle\!\langle \eta \rangle\!\rangle(\gamma) \overset{\text{def}}{=} \bigcup_{\Sigma[\alpha] \in \eta} \langle\!\langle \Sigma[\alpha] \rangle\!\rangle(\gamma)$ and $w \downarrow_{\langle \eta \rangle(\gamma)}$ is the word obtained from w by deleting each symbol not in $\langle\!\langle \eta \rangle\!\rangle(\gamma)$ from it. The i-th composition of L with itself is defined, as usual, by $\mathsf{L}^0 \overset{\text{def}}{=} \epsilon$ and $\mathsf{L}^{i+1} = \mathsf{L}^i \cdot \mathsf{L}$, for $i \geq 0$. We denote by $\mathsf{supp}(\mathsf{L})$ the *support* of L i.e., set of alphabet symbols $\Sigma[\alpha]$ from the regular expression L.

Example 5.3. Let $\gamma = (\{c_1, c_2, c_3, c_4\}, \{(c_1, out, c_2, in), (c_2, out, c_3, in), (c_3, out, c_4, in)\}, \varrho, \nu)$ be a configuration, such that $\nu(x) = c_1$, $\nu(y) = c_2$ and $\nu(z) = c_3$. Then, we have $\langle\!\langle \Sigma[\langle x.out, y.in \rangle] \rangle\!\rangle(\gamma) = \{(c_1, out, c_2, in)\}$, $\langle\!\langle \Sigma[\langle y.out, z.in \rangle] \rangle\!\rangle(\gamma) = \{(c_2, out, c_3, in)\}$ and $\langle\!\langle \Sigma[\mathsf{chain}_{0,0}(x, z)] \rangle\!\rangle(\gamma) = \{(c_1, out, c_2, in), (c_2, out, c_3, in)\}$. ■

Given a configuration γ and a predicate atom α, there can be, in principle, more than one subconfiguration $\gamma' \sqsubseteq \gamma$, such that $\gamma' \models \alpha$. This is problematic, because then $\langle\!\langle \Sigma[\alpha] \rangle\!\rangle(\gamma)$ may contain interactions from different subconfigurations of γ, that are models of α, thus cluttering the definition of the language $\langle\!\langle \Sigma[\alpha] \rangle\!\rangle(\gamma)$. We fix this issue by adapting the notion of *precision*, originally introduced for SL [Calcagno et al. 2007; O'Hearn et al. 2009], to our configuration logic:

Definition 5.4 (Precision). A formula ϕ is *precise on a set S of configurations* if and only if, for every configuration $\gamma \in S$, there exists at most one configuration γ', such that $\gamma' \sqsubseteq \gamma$ and $\gamma' \models \phi$. A set of formulæ Φ is *precisely closed* if ψ is precise on $[[\phi]]$, for any two formulæ $\phi, \psi \in \Phi$.

Symbolic configurations using predicate atoms are not precise for Γ, in general[5]. To understand this point, consider a configuration consisting of two overlapping models of $\mathsf{chain}_{h,t}(x, y)$, starting and ending in x and y, respectively, with a component that branches on two interactions after x and another component that joins the two branches before y. Then $\mathsf{chain}_{h,t}(x, y)$ is not precise on such configurations (that are not models of $\mathsf{chain}_{h,t}(x, y)$ whatsoever). On the positive side, we can state the following:

Proposition 5.5. *The set of symbolic configurations built using predicate atoms* $\mathsf{chain}_{h,t}(x, y)$*, for* $h, t \geq 0$ *(Example 3.1) is precisely closed.*

Two regular expressions are *congruent* if they denote the same language, whenever interpreted in the same configuration. Lifted to models of a symbolic configuration, we define:

Definition 5.6. Given a symbolic configuration ϕ, the regular expressions L_1 and L_2 are *congruent for* ϕ, denoted $\mathsf{L}_1 \cong_\phi \mathsf{L}_2$, if and only if $\langle\!\langle \mathsf{L}_1 \rangle\!\rangle(\gamma) = \langle\!\langle \mathsf{L}_2 \rangle\!\rangle(\gamma)$, for all configurations $\gamma \in [\![\phi]\!]$.

Despite the universal condition that ranges over a possibly infinite set of configurations, congruence of regular expressions with alphabet symbols of the form $\Sigma[\alpha]$, where α is an interaction or a predicate atom, is effectively decidable by an argument similar to the one used to prove equivalence of symbolic automata [D'Antoni and Veanes 2021].

5.2 Inference Rules for Havoc Triples

We use judgements of the form $\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}$, called *havoc triples*, where ϕ and ψ are CL formulæ, L is a regular expression, and η is an *environment* (a set of alphabet symbols), whose role will be made clear below (Def. 5.13 and Lemma 5.14). A havoc triple states that each finite sequence of (possibly loose) interactions described by a word in L, when executed in a model of the precondition ϕ, yields a model of the postcondition ψ.

Definition 5.7. A havoc triple $\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}$ is *valid*, written $\models \eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}$ if and only if, for each configuration $\gamma \in [\![\phi]\!]$, each sequence of interactions $w \in \langle\!\langle \mathsf{L} \rangle\!\rangle(\gamma)$ and each configuration γ', such that $\gamma \overset{w}{\dashrightarrow} \gamma'$, we have $\gamma' \in [\![\psi]\!]$.

For a symbolic configuration ϕ, we denote by $\mathsf{inter}(\phi)$ and $\mathsf{preds}(\phi)$ the sets of interaction and predicate atoms from ϕ, respectively and define the set of atoms $\mathsf{atoms}(\phi) \overset{\text{def}}{=} \mathsf{inter}(\phi) \cup \mathsf{preds}(\phi)$ and the regular expression $\Sigma[\phi] \overset{\text{def}}{=} \bigcup_{\alpha \in \mathsf{atoms}(\phi)} \Sigma[\alpha]$. We show that the validity of a havoc triple is a sufficient argument for the validity of a havoc query; because havoc triples are evaluated via open state changes (Def. 5.7), the dual implication is not true, in general.

Proposition 5.8. *If* $\models \eta \triangleright \{\!\{\phi\}\!\}\ \Sigma[\phi]^*\ \{\!\{\psi\}\!\}$ *then* $\mathfrak{h}([\![\phi]\!]) \subseteq [\![\psi]\!]$.

We describe next a set of axioms and inference rules used to prove the validity of havoc triples. For a symbolic configuration ϕ, we write $x \simeq_\phi y$ ($x \not\simeq_\phi y$) if and only if the equality (disequality) between x and y is asserted by the symbolic configuration ϕ, e.g. $x \simeq_{\mathsf{emp} * x=z * z=y} y$ and $x \not\simeq_{x@\_ * y@\_} y$; note that $x \not\simeq_\phi y$ is not necessarily the negation of $x \simeq_\phi y$.

Definition 5.9. For a symbolic configuration ϕ and an interaction atom $\langle x_1.p_1, x_2.p_2 \rangle$, we write:

- $\phi \dagger \langle x_1.p_1, x_2.p_2 \rangle$ if and only if ϕ contains a subformula $y@q$, such that $y \simeq_\phi x_i$ and q is not the pre-state of some behavior transition with label p_i, for some $i = 1, 2$; intuitively, any interaction defined by the formula $\langle x_1.p_1, x_2.p_2 \rangle$ is disabled in any model of ϕ,

[5]Unlike the predicates that define acyclic data structures (lists, trees) in SL, which are typically precise.

- $\phi \ddagger \langle x_1.p_1, x_2.p_2 \rangle$ if and only if, for each interaction atom $\langle y_1.p_1', y_2.p_2' \rangle \in \mathrm{inter}(\phi)$, there exists $i \in [1,2]$, such that $x_i \not\approx_\phi y_i$; intuitively, the interaction defined by the formula $\langle x_1.p_1, x_2.p_2 \rangle$ is not already present in a model of ϕ i.e., $\langle x_1.p_1, x_2.p_2 \rangle * \phi$ is satisfiable.

The axioms (Fig. 5a) discharge valid havoc triples for the empty sequence (ϵ), that changes nothing and the sequence consisting of a single interaction atom, that can be either disabled in every model (†), or enabled in some model (Σ) of the precondition, respectively; in particular, the (Σ) axiom describes the open state change produced by an interaction (Def. 5.1), firing on a (possibly empty) set of components, whose states match the pre-states of transitions for the associated behaviors. The ($\bot$) axiom discharges trivially valid triples with unsatisfiable (false) preconditions.

The redundancy rule (I−) in Fig. 5b removes an interaction atom from the precondition of a havoc triple, provided that the atom is never interpreted as an interaction from the language denoted by the regular expression from the triple. Conversely, the rule (I+) adds an interaction to the precondition, provided that the precondition (with that interaction atom) is consistent. Note that, without the $\phi \ddagger \alpha$ side condition, we would obtain a trivial proof for any triple, by adding an interaction atom twice to the precondition, i.e. using the rule (I+), followed by ($\bot$).

The composition rule ($\bowtie$) splits a proof obligation into two simpler havoc triples (Fig. 5c). The pre- and postconditions of the premisses are subformulæ of the pre- and postcondition of the conclusion, joined by separating conjunction and extended by so-called *frontier* formulæ, describing those sets of interaction atoms that may cross the boundary between the two separated conjuncts. The frontier formulæ play the role of environment assumptions in a rely/assume-guarantee style of reasoning [Jones 1981; Owicki and Gries 1978]. They are required for soundness, under the semantics of open state changes (Def. 5.1), which considers that the interactions can fire anytime, unless they are explicitly disabled by some component from ϕ_i, for $i = 1, 2$.

Nevertheless, defining the frontier syntactically faces the following problem: interactions introduced by a predicate atom in ϕ_i, can impact the state of a component defined by ϕ_{3-i}. We tackle this problem by forbidding predicate atoms that describe configurations with loose ports, that belong to components lying outside of the current configuration. We recall that a configuration $(C, \mathcal{I}, \varrho, \nu)$ is tight if and only if, for each interaction $(c_1, p_1, c_2, p_2) \in \mathcal{I}$, we have $c_1, c_2 \in C$. Moreover, we say that a formula ξ is *tight* if and only if every model of ξ is tight. For instance, a predicate atom $\mathsf{chain}_{h,t}(x, y)$, for given $h, t \geq 0$ (Example 3.1) is tight, because, in each model, the interactions involve only the *out* and *in* ports of adjacent components from the configuration.

Definition 5.10 (Frontier). Given symbolic configurations ϕ_1 and ϕ_2, the *frontier of* ϕ_i *and* ϕ_{3-i} is the formula $\mathcal{F}(\phi_i, \phi_{3-i}) \stackrel{\text{def}}{=} \circledast_{\alpha \in \mathrm{inter}(\phi_{3-i}) \setminus (\mathrm{inter}(\overline{\phi}_{3-i}) \cup \mathrm{inter}(\phi_i))} \alpha$, where $\overline{\phi}_i$ is the largest tight subformula of ϕ_i, for $i = 1, 2$.

Example 5.11. Let $\phi_1 = \mathsf{chain}_{h,t}(x, y) * \langle y.out, z.in \rangle$ and $\phi_2 = \mathsf{chain}_{h,t}(y, z) * \langle x.out, y.in \rangle$. We have $\mathcal{F}(\phi_1, \phi_2) = \langle x.out, y.in \rangle$ and $\mathcal{F}(\phi_2, \phi_1) = \langle y.out, z.in \rangle$, because the tightness of $\mathsf{chain}_{h,t}(x, y)$ and $\mathsf{chain}_{h,t}(y, z)$ means that the only interactions crossing the boundary of ϕ_1 and ϕ_2 are the ones described by $\langle y.out, z.in \rangle$ and $\langle x.out, y.in \rangle$. ■

Finally, the regular expression of the conclusion of the ($\bowtie$) rule is the interleaving of the regular expressions from the premisses, taken with respect to the sets of alphabet symbols $\eta_i = \Sigma[\phi_i * \mathcal{F}(\phi_i, \phi_{3-i})]$, for $i = 1, 2$.

The rules in Fig. 5d introduce regular expressions built using concatenation, Kleene star and union. In particular, for reasons related to the soundness of the proof system, the concatenation rule ($\cdot$) applies to havoc triples whose preconditions are finite disjunctions of symbolic configurations, sharing the same structure of component, interaction and predicate atoms, whereas the cut formulæ

$$(\epsilon)\ \frac{}{\eta \triangleright \{\!\{\phi\}\!\}\ \epsilon\ \{\!\{\phi\}\!\}} \qquad (\dagger)\ \frac{}{\eta \triangleright \{\!\{\phi\}\!\}\ \Sigma[\alpha]\ \{\!\{\mathsf{false}\}\!\}}\ \begin{array}{l}\alpha = \langle x_1.p_1, x_2.p_2\rangle \\ \phi\ \dagger\ \alpha\end{array}$$

$$(\bot)\ \frac{}{\eta \triangleright \{\!\{\mathsf{false}\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}}$$

$$(\Sigma)\ \frac{}{\eta \triangleright \{\!\{\alpha * \circledast_{j\in J}\ x_j@q_j\}\!\}\ \Sigma[\alpha]\ \{\!\{\alpha * \circledast_{j\in J} \bigvee_{q_j \xrightarrow{p_j} q'_j} x_j@q'_j\}\!\}}\ \begin{array}{l}\alpha = \langle x_1.p_1, x_2.p_2\rangle \\ J \subseteq [1,2]\end{array}$$

a. Axioms

$$(\mathsf{I}-)\ \frac{\eta \setminus \{\Sigma[\alpha]\} \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}}{\eta \triangleright \{\!\{\phi * \alpha\}\!\}\ \mathsf{L}\ \{\!\{\psi * \alpha\}\!\}}\ \begin{array}{l}\alpha = \langle x_1.p_1, x_2.p_2\rangle \\ \Sigma[\alpha] \in \eta \setminus \mathsf{supp}(\mathsf{L})\end{array}$$

$$(\mathsf{I}+)\ \frac{\eta \cup \{\Sigma[\alpha]\} \triangleright \{\!\{\phi * \alpha\}\!\}\ \mathsf{L}\ \{\!\{\psi * \alpha\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}}\ \begin{array}{l}\alpha = \langle x_1.p_1, x_2.p_2\rangle \\ \phi\ \ddagger\ \alpha\end{array}$$

b. Redundancy Rules

$$(\bowtie)\ \frac{\eta_i \triangleright \{\!\{\phi_i * \mathcal{F}(\phi_i, \phi_{3-i})\}\!\}\ \mathsf{L}_i\ \{\!\{\psi_i * \mathcal{F}(\phi_i, \phi_{3-i})\}\!\} \mid i = 1,2}{\eta_1 \cup \eta_2 \triangleright \{\!\{\phi_1 * \phi_2\}\!\}\ \mathsf{L}_1 \bowtie_{\eta_1,\eta_2} \mathsf{L}_2\ \{\!\{\psi_1 * \psi_2\}\!\}}\ \begin{array}{l}\eta_i = \Sigma[\phi_i * \mathcal{F}(\phi_i, \phi_{3-i})] \\ i = 1,2\end{array}$$

c. Composition Rule

$$(\cdot)\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1\ \{\!\{\xi\}\!\} \qquad \eta \triangleright \{\!\{\xi\}\!\}\ \mathsf{L}_2\ \{\!\{\psi\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1 \cdot \mathsf{L}_2\ \{\!\{\psi\}\!\}}\ \phi \succeq \xi \qquad (*)\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\phi\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}^*\ \{\!\{\phi\}\!\}}$$

$$(\cup)\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1\ \{\!\{\psi\}\!\} \qquad \eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_2\ \{\!\{\psi\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1 \cup \mathsf{L}_2\ \{\!\{\psi\}\!\}} \qquad (\subset)\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1 \cup \mathsf{L}_2\ \{\!\{\psi\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1\ \{\!\{\psi\}\!\}}$$

$$(\cong)\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_1\ \{\!\{\psi\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}_2\ \{\!\{\psi\}\!\}}\ \mathsf{L}_1 \cong_\phi \mathsf{L}_2$$

d. Regular Expression Rules

$$(\mathsf{C})\ \frac{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi'\}\!\}}{\eta \triangleright \{\!\{\phi\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}}\ \psi' \models \psi$$

$$(\mathsf{LU})\ \frac{\eta' \triangleright \{\!\{\phi * \xi'\}\!\}\ \mathsf{L}'\ \{\!\{\psi\}\!\} \quad \left|\ \begin{array}{l}\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})}) \leftarrow \exists \mathbf{z}\,.\,\xi\ \in\ \Delta \\ (\exists \mathbf{z}\,.\,\xi)[x_1/y_1,\ldots,x_{\#(\mathsf{A})}/y_{\#(\mathsf{A})}] = \exists \mathbf{z}\,.\,\xi' \\ \eta' = \big(\eta \setminus \{\Sigma[\mathsf{A}(y_1,\ldots,y_{\#(\mathsf{A})})]\}\big) \cup \Sigma[\xi'] \\ \mathsf{L}' = \mathsf{L}\big[\Sigma[\mathsf{A}(x_1,\ldots,x_{\#(\mathsf{A})})]\ /\ \Sigma[\xi']\big]\end{array}\right.}{\eta \triangleright \{\!\{\phi * \mathsf{A}(y_1,\ldots,y_{\#(\mathsf{A})})\}\!\}\ \mathsf{L}\ \{\!\{\psi\}\!\}}$$

$$(\vee)\ \frac{\eta \triangleright \{\!\{\phi_i\}\!\}\ \mathsf{L}\ \{\!\{\psi_i\}\!\} \mid i \in [1,k]}{\eta \triangleright \{\!\{\bigvee_{i=1}^k \phi\}\!\}\ \mathsf{L}\ \{\!\{\bigvee_{i=1}^k \psi_i\}\!\}}\ \begin{array}{l}\phi_i \simeq \phi_j \\ i \neq j \in [1,k]\end{array}$$

$$(\wedge)\ \frac{\eta \triangleright \{\!\{\phi_i\}\!\}\ \mathsf{L}\ \{\!\{\psi_i\}\!\} \mid i \in [1,k]}{\eta \triangleright \{\!\{\bigwedge_{i=1}^k \phi_i\}\!\}\ \mathsf{L}\ \{\!\{\bigwedge_{i=1}^k \psi_i\}\!\}}\ \begin{array}{l}\phi_i \simeq \phi_j \\ i \neq j \in [1,k]\end{array}$$

e. Structural Rules

Fig. 5. Proof System for Havoc Triples

(postcondition of the left and precondition of the right premisse) share the same structure as the precondition. We formalize below the fact that two formulæ share the same structure:

Definition 5.12. Two formulæ ϕ and ψ *share the same structure*, denoted $\phi \simeq \psi$ if and only if they become equivalent when every component atom $x@q$ is replaced by the formula $x@\_$, in both ϕ and ψ. We write $\phi \succeq \psi$ if and only if ϕ is satisfiable and ψ is not, or else $\phi \simeq \psi$.

The ($\subset$) rule is the dual of ($\cup$), that restricts the language from the conclusion to a subset of the one from the premisse. As a remark, by applying the (I+) and ($\subset$) rules in any order, one can derive the havoc invariance of the intermediate assertions in a single-reversal reconfiguration sequence (see Def. 4.7 and Prop. 4.8). The rule ($\cong$) substitutes a regular expression with a congruent one, with respect to the precondition.

Last, the rules in Fig. 5e modify the structure of the pre- and postconditions. In particular, the left unfolding rule (LU) has a premisse for each step of unfolding of a predicate atom from the conclusion's precondition, with respect to a rule from the SID. The environment and the regular expression in each premisse are obtained by replacing the alphabet symbol of the unfolded predicate symbol by the set of alphabet symbols from the unfolding step, where $\mathsf{L}[\Sigma[\alpha]/\mathsf{L}']$ denotes the regular expression obtained by replacing each occurrence of the alphabet symbol $\Sigma[\alpha]$ in L with the regular expression L'.

5.3 Havoc Proofs

A *proof tree* is a finite tree T whose nodes are labeled by havoc triples and, for each node n not on the frontier of T, the children of n are the premisses of the application of a rule from Fig. 5, whose conclusion is the label of n. For the purposes of this paper, we consider only proof trees that meet the following condition:

Assumption 1. *The root of the proof tree is labeled by a havoc triple* $\eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$, *such that* ϕ *is a symbolic configuration and* $\eta = \{\Sigma[\alpha] \mid \alpha \in \mathrm{atoms}(\phi)\}$.

It is easy to check that the above condition on the shape of the precondition and the relation between the precondition and the environment holds recursively, for the labels of all nodes in a proof tree that meets assumption 1. Before tackling the soundness of the havoc proof system (Fig. 5), we state an invariance property of the environments of havoc triples that occur in a proof tree:

Definition 5.13. A havoc triple $\eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$ is *distinctive* if and only if $\langle\!\langle \Sigma[\alpha_1] \rangle\!\rangle(\gamma) \cap \langle\!\langle \Sigma[\alpha_2] \rangle\!\rangle(\gamma) = \emptyset$, for all $\Sigma[\alpha_1], \Sigma[\alpha_2] \in \eta$ and all $\gamma \in [\![\phi]\!]$.

The next lemma is proved inductively on the structure of the proof tree, using Assumption 1.

Lemma 5.14. *Given a proof tree* T, *each node in* T *is labeled with a distinctive havoc triple.*

In order to deal with inductively defined predicates that occur within the pre- and postconditions of the havoc triples, we use cyclic proofs [Brotherston and Simpson 2011]. A *cyclic proof tree* T is a proof tree such that every node on the frontier is either the conclusion of an axiom in Fig. 5a, or there is another node m whose label matches the label of n via a substitution of variables; we say that n is a *bud* and m is its *companion*. A cyclic proof tree is a *cyclic proof* if and only if every infinite path through the proof tree extended with bud-companion edges, goes through the conclusion of a (LU) rule infinitely often[6]. We denote by $\Vdash \eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$ the fact that $\eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$ labels the root of a cyclic proof and state the following soundness theorem:

Theorem 5.15. *If* $\Vdash \eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$ *then* $\models \eta \triangleright \{\!\{\phi\}\!\} \mathsf{L} \{\!\{\psi\}\!\}$.

The proof is by induction on the structure of the proof tree, using Lemma 5.14.

[6]This condition can be effectively decided by checking the emptiness of a Büchi automaton [Brotherston and Simpson 2011].

5.4 A Havoc Proof Example

We illustrate the use of the proof system in Fig. 5 on the havoc invariance side conditions required by the reconfiguration proofs from §4.3. In fact, we prove a more general statement, namely that $\mathsf{chain}_{h,t}(x,y)$ is havoc invariant, for all $h, t \geq 0$ (see Example 3.1 for the definition of the $\mathsf{chain}_{h,t}(x,y)$ predicates).

The idea of the proof is to unfold the precondition of the havoc triple $\{\Sigma[\mathsf{chain}_{h,t}(z,x)]\} \triangleright \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}\ \Sigma[\mathsf{chain}_{h,t}(z,x)]^*\ \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}$ by an application of (LU), discharge the base cases $x@\_$, $x@\mathsf{H}$ and $x@\mathsf{T}$ by applications of (ϵ) and prove the non-trivial subgoals (**A**) and (**B**), corresponding to the unfoldings $z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)$ and $z@\mathsf{T} * \langle z.out, y.in\rangle * \mathsf{chain}_{h,t\dot{-}1}(y,x)$ of $\mathsf{chain}_{h,t}(z,x)$, respectively.

$$\text{(LU)}\ \frac{\text{(}\epsilon\text{)}\ \dfrac{}{\emptyset \triangleright \{\!\{x@\_\}\!\} \in \{\!\{x@\_\}\!\}} \quad \text{(}\epsilon\text{)}\ \dfrac{}{\emptyset \triangleright \{\!\{x@\mathsf{H}\}\!\} \in \{\!\{x@\mathsf{H}\}\!\}} \quad \text{(}\epsilon\text{)}\ \dfrac{}{\emptyset \triangleright \{\!\{x@\mathsf{T}\}\!\} \in \{\!\{x@\mathsf{T}\}\!\}} \quad \textbf{(A) (B)}}{\{\Sigma[\mathsf{chain}_{h,t}(z,x)]\} \triangleright \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}\ \Sigma[\mathsf{chain}_{h,t}(z,x)]^*\ \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}\ \textbf{(1)}}$$

The regular expression describing the language of interactions is split (using congruence) into:

- sequences in which the interaction $\langle z.out, y.in\rangle$ is never executed, described by the regular expression ${\Sigma^1_{y,x}}^*$, where $\Sigma^1_{y,x} \stackrel{\text{def}}{=} \Sigma[\mathsf{chain}_{h\dot{-}1,t}(y,x)]$, and
- sequences in which $\langle z.out, y.in\rangle$ is executed at least once, described by the regular expression ${\Sigma^1_{y,x}}^* \cdot \Sigma_{z,y} \cdot (\Sigma_{z,y} \cup \Sigma^1_{y,x})^*$, where $\Sigma_{z,y} \stackrel{\text{def}}{=} \Sigma[\langle z.out, y.in\rangle]$.

The proof for the first case leads to the subgoal $\{\Sigma^1_{y,x}\} \triangleright \{\!\{\mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}\ {\Sigma^1_{y,x}}^*\ \{\!\{\mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}$, with a similar structure as the goal of the proof, hence the backlink to (**1**)[7]. The rule (C) strenghtens the postcondition $\mathsf{chain}_{h,t}(z,x)$ to an unfolding $\mathsf{chain}_{h,t}(z,x) \Leftarrow \exists y\ .\ z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)$, whose existentially quantified variable is, moreover, bound to the free variable y from the precondition. The frontier formulæ for the rule $(\bowtie)$ are $\mathcal{F}(z@\mathsf{H}, \mathsf{chain}_{h\dot{-}1,t}(y,x)) = \mathcal{F}(\mathsf{chain}_{h\dot{-}1,t}(y,x), z@\mathsf{H}) = \mathsf{emp}$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\text{(}\epsilon\text{)}\ \dfrac{}{\emptyset \triangleright \{\!\{z@\mathsf{H}\}\!\} \in \{\!\{z@\mathsf{H}\}\!\}} \quad \text{(}\bowtie\text{)}\ \dfrac{\text{backlink to } \textbf{(1)}}{\{\Sigma^1_{y,x}\} \triangleright \{\!\{\mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}\ {\Sigma^1_{y,x}}^*\ \{\!\{\mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}{\text{(I}-\text{)}\ \dfrac{\{\Sigma^1_{y,x}\} \triangleright \{\!\{z@\mathsf{H} * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}\ {\Sigma^1_{y,x}}^*\ \{\!\{z@\mathsf{H} * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}{\textbf{(A1)}\ \{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}}{\text{(C)}\ \dfrac{{\Sigma^1_{y,x}}^*\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}{\{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}}{\text{(}\cup\text{)}\ \dfrac{{\Sigma^1_{y,x}}^*\ \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}}{\{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}\ \textbf{(A2)}}{\text{(}\cong\text{)}\ \dfrac{{\Sigma^1_{y,x}}^* \cup [{\Sigma^1_{y,x}}^* \cdot \Sigma_{z,y} \cdot (\Sigma_{z,y} \cup \Sigma^1_{y,x})^*]\ \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}}{\textbf{(A)}\ \{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}\ (\Sigma_{z,y} \cup \Sigma^1_{y,x})^*\ \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}}}$$

The proof for the language ${\Sigma^1_{y,x}}^* \cdot \Sigma_{z,y} \cdot (\Sigma_{z,y} \cup \Sigma^1_{y,x})^*$ (**A2**) is given below:

[7] To save space, we draw backlinks between nodes whose labels differ by a renaming of predicates $\mathsf{chain}_{h,t}$ into $\mathsf{chain}_{h',t'}$, such that (h',t') is lexicographically smaller than (h,t).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{backlink to } \textbf{(A1)}}{\{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}} \quad \textbf{(A3)} \quad \textbf{(A4)}}{(\cdot)\ \{\Sigma^1_{y,x}{}^{*}\ \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}{\{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}{(\mathrm{C})\ \dfrac{\Sigma^1_{y,x}{}^{*} \cdot \Sigma_{z,y} \cdot (\Sigma_{z,y} \cup \Sigma^1_{y,x})^* \{\!\{\mathsf{false}\}\!\}}{\textbf{(A2)}\ \{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}}}{(\dagger)\ \dfrac{\Sigma^1_{y,x}{}^{*} \cdot \Sigma_{z,y} \cdot (\Sigma_{z,y} \cup \Sigma^1_{y,x})^* \{\!\{\mathsf{chain}_{h,t}(z,x)\}\!\}}{\textbf{(A3)}\ \{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright \{\!\{z@\mathsf{H} * \langle z.out, y.in\rangle * \mathsf{chain}_{h\dot{-}1,t}(y,x)\}\!\}\ \Sigma_{z,y}\ \{\!\{\mathsf{false}\}\!\}}}{(\bot)\ \dfrac{}{\textbf{(A4)}\ \{\Sigma_{z,y}, \Sigma^1_{y,x}\} \triangleright \{\!\{\mathsf{false}\}\!\}\ (\Sigma_{z,y} \cup \Sigma^1_{y,x})^*\ \{\!\{\mathsf{false}\}\!\}}}$$

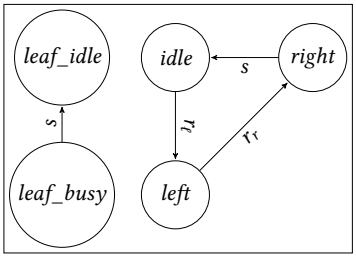
6 A WORKED-OUT EXAMPLE: RECONFIGURABLE TREE ARCHITECTURES

In addition to token rings (Fig. 1), we apply our method to reconfiguration scenarios of distributed systems with tree-shaped architectures. Such (virtual) architectures are e.g. used in flooding and leader election algorithms. They are applicable, for instance, when every component in the system must notify a designated controller, placed in the root of the tree, about an event that involves each component from the frontier of the tree. Conversely, the root component may need to notify the rest of the components. The tree architecture guarantees that the notification phase takes time $O(\log n)$ in the number n of components in the tree, when the tree is balanced, i.e. the lengths of the longest and shortest paths between the root and the frontier differ by at most a constant factor. A reconfiguration of a tree places a designated component (whose priority has increased dynamically) closer to the frontier (dually, closer to the root) in order to receive the notification faster. In balanced trees, reconfigurations involve structure-preserving rotations. For instance, *self-adjustable splay-tree networks* [Schmid et al. 2016] use the *zig* (left rotation), *zig-zig* (left-left rotation) and *zig-zag* (left-right rotation) operations [Sleator and Tarjan 1985] to move nodes in the tree, while keeping the balance between the shortest and longest paths.

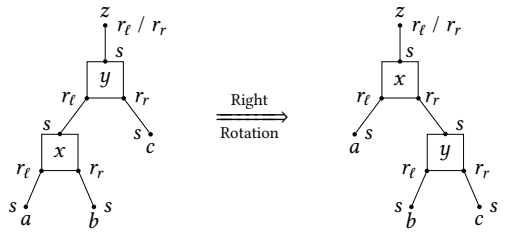
Fig. 6 shows a model of reconfigurable tree architectures, in which each leaf component starts in state *leaf_busy* and sends a notification to its parent before entering the *leaf_idle* state. An inner component starts in state *idle* and waits for notifications from both its left (r_ℓ) and right (r_r) children before sending a notification to its parent (s), unless this component is the root (Fig. 6a). We model notifications by interactions of the form $\langle \_.s, \_.r_\ell\rangle$ and $\langle \_.s, \_.r_r\rangle$. The notification phase is completed when the root is in state *right*, every inner component is in the *idle* state and every leaf is in the *leaf_idle* state.

Fig. 6b shows a *right rotation* that reverses the positions of components with identifiers x and y, implemented by the reconfiguration program from Fig. 7. The rotation applies only to configurations in which both x and y are in state *idle*, by distinguishing the case when y is a left or a right child of z. For simplicity, Fig. 7 shows the program in case y is a left child, the other case being symmetric. Note that, applying the rotation in a configuration where the component indexed by x is in state *right* (both a and b have sent their notifications to x) and the one indexed by y is in state *idle* (c has not yet sent its notification to y) yields a configuration from which c cannot send its notification further, because x has now become the root of the subtree changed by the rotation.

We prove that, *whenever a right rotation is applied to a tree, such that the subtrees rooted at a, b and c have not sent their notifications yet, the result is another tree in which the subtrees rooted at a, b and c are still waiting to submit their notifications.* This guarantees that the notification phase



(a)



(b)

$$\mathsf{tree}_{idle}(x) \leftarrow x@leaf\_idle$$
$$\mathsf{tree}_{idle}(x) \leftarrow \exists y \exists z \,.\, x@idle * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}_{idle}(y) * \mathsf{tree}_{idle}(z)$$

$$\mathsf{tree}_{\neg idle}(x) \leftarrow x@leaf\_busy$$
$$\mathsf{tree}_{\neg idle}(x) \leftarrow \exists y \exists z \,.\, x@left * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}_{idle}(y) * \mathsf{tree}_{\neg idle}(z)$$
$$\mathsf{tree}_{\neg idle}(x) \leftarrow \exists y \exists z \,.\, x@right * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}_{idle}(y) * \mathsf{tree}_{idle}(z)$$
$$\mathsf{tree}_{\neg idle}(x) \leftarrow \exists y \exists z \,.\, x@idle * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}_{\neg idle}(y) * \mathsf{tree}_{\neg idle}(z)$$

(c)

$$\mathsf{tree}(x) \leftarrow x@leaf\_idle$$
$$\mathsf{tree}(x) \leftarrow x@leaf\_busy$$
$$\mathsf{tree}(x) \leftarrow \exists y \exists z \,.\, x@\_ * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}(y) * \mathsf{tree}(z)$$

$$\mathsf{tseg}(x, x) \leftarrow x@\_$$
$$\mathsf{tseg}(x, u) \leftarrow \exists y \exists z \,.\, x@\_ * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tseg}(y, u) * \mathsf{tree}(z)$$
$$\mathsf{tseg}(x, u) \leftarrow \exists y \exists z \,.\, x@\_ * \langle y.s, x.r_\ell \rangle * \langle z.s, x.r_r \rangle * \mathsf{tree}(y) * \mathsf{tseg}(z, u)$$

Fig. 6. Reconfiguration of a Tree Architecture

will terminate properly with every inner component (except for the root) in state *idle* and every leaf component in state *leaf_idle*, even if one or more reconfigurations take place in between. In particular, this proves the correctness of more complex reconfigurations of splay tree architectures, using e.g. the zig-zig and zig-zag operations [Schmid et al. 2016].

The proof in Fig. 7 uses the inductive definitions from Fig. 6c. The predicates $\mathsf{tree}_{idle}(x)$ and $\mathsf{tree}_{\neg idle}(x)$ define trees where all components are idle, and where some notifications are still being propagated, respectively. The predicate $\mathsf{tree}(x)$ conveys no information about the states of the components and the predicate $\mathsf{tseg}(x, u)$ defines a tree segment, from component x to component u. To use the havoc proof system from Fig. 5, we need the following statement[8]:

Proposition 6.1. *The set of symbolic configurations using predicate atoms* $\mathsf{tree}_{idle}(x)$, $\mathsf{tree}_{\neg idle}(x)$, $\mathsf{tree}(x)$ *and* $\mathsf{tseg}(x, y)$ *is precisely closed.*

Moreover, each predicate atom $\mathsf{tree}_{idle}(x)$, $\mathsf{tree}_{\neg idle}(x)$, $\mathsf{tree}(x)$ and $\mathsf{tseg}(x, y)$ is tight, because, in each model of these atoms, the interactions $\langle u.s, v.r_\ell \rangle$ and $\langle u.s, v.r_r \rangle$ are between the ports $\langle s, r_\ell \rangle$ and $\langle s, r_r \rangle$ of the components u and v, respectively.

The precondition of the reconfiguration program in Fig. 7 states that x and y are idle components, and the a, b and c subtrees are not idle, whereas the postcondition states that the x subtree is not idle. As mentioned, *this is sufficient to guarantee the correct termination of the notification phase after the right rotation.* As in the proofs from §4.3, proving the correctness of the sequential composition of primitive commands requires proving the havoc invariance of the annotations. However, since in this case, the reconfiguration sequence is single-reversal (Def. 4.7), we are left with proving havoc invariance only for the annotations marked with (♯) in Fig. 7 (Prop. 4.8).

[8]This is similar to Prop. 5.5.

$$\left\{\begin{array}{l}\exists r\exists x\exists y\exists z\exists a\exists b\exists c\ .\ \mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle y.s, z.r_\ell\rangle * \langle x.s, y.r_\ell\rangle * \langle b.s, x.r_r\rangle * \\ \qquad\qquad x@idle * y@idle * \mathsf{tree}_{\neg idle}(a) * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c)\end{array}\right\}$$

with $x, y, z, a, b, c : \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle y.s, z.r_\ell\rangle * \langle x.s, y.r_\ell\rangle * \langle b.s, x.r_r\rangle * x@idle * y@idle$ **do**

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle y.s, z.r_\ell\rangle * \langle x.s, y.r_\ell\rangle * \langle b.s, x.r_r\rangle\ * \\ x@idle * y@idle * \mathsf{tree}_{\neg idle}(a) * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c)\end{array}\right\}$$

`disconnect(`$b.s, x.r_r$`);`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle y.s, z.r_\ell\rangle * \langle x.s, y.r_\ell\rangle\ * \\ (x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c)\end{array}\right\}\ (\sharp)$$

`disconnect(`$x.s, y.r_\ell$`);`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle y.s, z.r_\ell\rangle * \\ (x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c)\end{array}\right\}$$

`disconnect(`$y.s, z.r_\ell$`);`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \\ (x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c)\end{array}\right\}$$

`connect(`$b.s, y.r_\ell$`);`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle b.s, y.r_\ell\rangle * \\ (x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * \\ (y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c) \vee y@left * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{\neg idle}(c) \\ \vee y@right * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c))\end{array}\right\}$$

`connect(`$y.s, x.r_r$`);`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle b.s, y.r_\ell\rangle * \langle y.s, x.r_r\rangle * \\ ((x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * \\ (y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c) \vee y@left * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{\neg idle}(c) \\ \vee y@right * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)))\vee \\ x@right * y@idle * \mathsf{tree}_{idle}(a) * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)\end{array}\right\}\ (\sharp)$$

`connect(`$x.s, z.r_\ell$`)`

$$\left\{\begin{array}{l}\mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle b.s, y.r_\ell\rangle * \langle y.s, x.r_r\rangle * \langle x.s, z.r_\ell\rangle * \\ ((x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * \\ (y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c) \vee y@left * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{\neg idle}(c) \\ \vee y@right * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)))\vee \\ x@right * y@idle * \mathsf{tree}_{idle}(a) * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)\end{array}\right\}$$

od

$$\left\{\begin{array}{l}\exists r, x, y, z, a, b, c\ .\ \mathsf{tseg}(r,z) * \langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle b.s, y.r_\ell\rangle * \langle y.s, x.r_r\rangle * \langle x.s, z.r_\ell\rangle * \\ ((x@idle * \mathsf{tree}_{\neg idle}(a) \vee x@left * \mathsf{tree}_{idle}(a)) * \\ (y@idle * \mathsf{tree}_{\neg idle}(b) * \mathsf{tree}_{\neg idle}(c) \vee y@left * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{\neg idle}(c) \\ \vee y@right * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)))\vee \\ x@right * y@idle * \mathsf{tree}_{idle}(a) * \mathsf{tree}_{idle}(b) * \mathsf{tree}_{idle}(c)\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists r, x, y, z, a, b, c\ .\ \mathsf{tseg}(r,z) * \langle x.s, z.r_\ell\rangle * \mathsf{tree}_{\neg idle}(x)\ \wedge \\ (\langle a.s, x.r_\ell\rangle * \langle c.s, y.r_r\rangle * \langle b.s, y.r_\ell\rangle * \langle y.s, x.r_r\rangle * \mathsf{true})\end{array}\right\}$$

Fig. 7. Proof of a Tree Rotation

7 TOWARDS AUTOMATED PROOF GENERATION

Proof generation can be automated, by tackling the following technical problems.

Entailment. Given a SID Δ and two CL formulæ ϕ and ψ, interpreted over Δ, is every model of ϕ also a model of ψ? This problem arises e.g., when applying the rule of consequence (Fig. 4c bottom-left) in a Hoare-style proof of a reconfiguration program. Unsurprisingly, the CL entailment inherits the positive and negative aspects of the SL entailment [Reynolds 2002]. For instance, one

can reduce the undecidable problem of universality of context-free languages [Bar-Hillel et al. 1961] to CL entailment, with ϕ and ψ restricted to predicate atoms. Decidability can be recovered via two restrictions on the syntax of the rules in the SID and a semantic restriction on the configurations that occur as models of the predicate atoms defined by the SID. The syntactic restrictions are that, each rule is of the form $\mathsf{A}(x_1, \ldots, x_{\#(\mathsf{A})}) \leftarrow \exists y_1 \ldots \exists y_m \,.\, x@q * \phi * \circledast_{\ell=1}^{h} \mathsf{B}^{\ell}(z_1^{\ell}, \ldots, z_{\#(\mathsf{B}^{\ell})}^{\ell})$, where ϕ consists of interaction atoms, such that: (1) x_1 occurs in each interaction atom from ϕ, (2) $\bigcup_{ell=1}^{h} \{z_1^{\ell}, \ldots, z_{\#(\mathsf{B}^{\ell})}^{\ell}\} = \{x_2, \ldots, x_{\#(\mathsf{A})}\} \cup \{y_1, \ldots, y_m\}$, and (3) for each $\ell \in [1, h]$, z_1^{ℓ} occurs in ϕ. Furthermore, the semantic restriction is that, in each model of a predicate atom, a component must occur in a *bounded number of interactions*, i.e., the structure is a graph of bounded degree. For instance, star topologies with a central controller and an unbounded number of workers can be defined in CL, but do not satisfy this constraint. With these restrictions, it can be shown that the CL entailment problem is 2EXP-complete, thus matching the complexity of the similar problem for SL [Echenim et al. 2020; Katelaan and Zuleger 2020]. Details are given in [Bozga et al. 2022a].

Frame inference. Given two CL formulæ ϕ and ψ find a formula ξ, such that $\phi \models \psi * \xi$. This problem occurs e.g., when applying the frame rule (Fig. 4c bottom-right) with a premisse $\{\phi\}\ \mathsf{R}\ \{\psi\}$ to an arbitrary precondition π i.e., one must infer a frame ξ such that $\pi \models \phi * \xi$. This problem has been studied for SL [Calcagno et al. 2011; Gorogiannis et al. 2011], in cases where the SID defines only data structures of a restricted form (typically nested lists). Reconsidering the frame inference problem for CL is of paramount importance for automating the generation of Hoare-style correctness proofs and is an open problem.

Automated havoc invariance. Given a precondition ϕ and a regular expression L, the parallel composition rule ($\bowtie$) requires the inference of regular expressions L_1 and L_2, such that $\mathsf{L}_1 \bowtie_{\eta_1,\eta_2} \mathsf{L}_2 \cong_{\phi} \mathsf{L}$. We conjecture that, under the bounded degree restriction above, the languages of the frontier (cross-boundary) interactions (Def. 5.10) are regular and can be automatically inferred by classical automata construction techniques. Another promising direction is using regular model checking techniques based on tree transducers, to reduce the havoc invariance to the entailment problem [Bozga et al. 2022b], for which a general decidable fragment was found [Bozga et al. 2022a].

8 RELATED WORK

The ability of reconfiguring coordinating architectures of software systems has received much interest in the Software Engineering community, see the surveys [Bradbury et al. 2004; Butting et al. 2017]. We consider *programmed reconfiguration*, in which the architecture changes occur according to a sequential program, executed in parallel with the system to which reconfiguration applies. The languages used to write such programs are classified according to the underlying formalism used to define their operational semantics: *process algebras*, e.g. π-ADL [Cavalcante et al. 2015], DARWIN [Magee and Kramer 1996], *hyper-graphs* and *graph rewriting* [Arad 2013; Cao et al. 2005; Le Metayer 1998; Taentzer et al. 1998; Wermelinger and Fiadeiro 2002], *chemical reactions* [Wermelinger 1998], etc. We separate architectures (structures) from behaviors, thus relating to the BIP framework [Basu et al. 2006] and its extensions for dynamic reconfigurable systems DR-BIP [El-Ballouli et al. 2021]. In a similar vein, the REO language [Arbab 2004] supports reconfiguration by changing the structure of connectors [Clarke 2008].

Checking the correctness of a dynamically reconfigurable system considers mainly *runtime verification* methods, i.e. checking a given finite trace of observed configurations against a logical specification. For instance, in [Bucchiarone and Galeotti 2008], configurations are described by annotated hyper-graphs and configuration invariants of finite traces, given first-order logic, are checked using ALLOY [Jackson 2002]. More recently, [Dormoy et al. 2010; El-Hokayem et al. 2021;

Lanoix et al. 2011] apply temporal logic to runtime verification of reconfigurable systems. Model checking of temporal specifications is also applied to REO programs, under simplifying assumption that render the system finite-state [Clarke 2008]. In contrast, we use induction to deal with parameterized systems of unbounded sizes.

To the best of our knowledge, our work is the first to tackle the *verification* of reconfiguration programs, by formally proving the absence of bugs, using a Hoare-style annotation of a reconfiguration program with assertions that describe infinite sets of configurations, with unboundedly many components. Traditionally, reasoning about the correctness of unbounded networks of parallel processes uses mostly hard-coded architectures (see [Bloem et al. 2015] for a survey), whereas the more recently developed architecture description logics [Konnov et al. 2016; Mavridou et al. 2017] do not consider the reconfigurability aspect of distributed systems.

Specifying parameterized component-based systems by inductive definitions is not new. *Network grammars* [Hirsch et al. 1998; Le Metayer 1998; Shtadler and Grumberg 1989] use context-free grammar rules to describe systems with linear (pipeline, token-ring) architectures obtained by composition of an unbounded number of processes. In contrast, we use predicates of unrestricted arities to describe architectural styles that are, in general, more complex than trees. Moreover, we write inductive definitions using a resource logic, suitable also for writing Hoare logic proofs of reconfiguration programs, based on local reasoning [Calcagno et al. 2007].

The assertion language introduced in this paper is a resource logic that supports local reasoning [O'Hearn et al. 2001]. Local reasoning about parallel programs has been traditionally within the scope of Concurrent Separation Logic (CSL), that introduced a parallel composition rule [O'Hearn 2007], with a non-interfering (race-free) semantics of shared-memory parallelism [Brookes and O'Hearn 2016]. Considering interference in CSL requires more general proof rules, combining ideas of assume- and rely-guarantee [Jones 1981; Owicki and Gries 1978] with local reasoning [Feng et al. 2007; Vafeiadis and Parkinson 2007] and abstract notions of framing [Dinsdale-Young et al. 2013, 2010; Farka et al. 2021]. These rules generalize from both standard CSL parallel composition and rely-guarantee rules, allowing even to reason about properties of concurrent objects, such as (non-)linearizability [Sergey et al. 2016]. However, the body of work on CSL deals almost entirely with shared-memory multithreading programs, instead of distributed systems, which is the aim of our work. In contrast, we develop a resource logic in which the processes do not just share and own resources, but become mutable resources themselves.

9 CONCLUSIONS

We present a framework for deductive verification of reconfiguration programs, based on a configuration logic that supports local reasoning. We prove the absence of design bugs in ideal networks, without packet loss and communication delays, using a discrete event-based model of behavior, the usual level of abstraction in formal verification of parameterized distributed systems. Our configuration logic relies on inductive predicates to describe systems with unbounded number of components. It is used to annotate reconfiguration programs with Hoare triples, whose validity relies on havoc invariants about the ongoing interactions in the system. These invariants are tackled with a specific proof system, that uses a parallel composition rule in the style of assume/rely-guarantee reasoning.

REFERENCES

Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. 2007. Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007 (LNCS, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 721–736.

C. Aiswarya, Benedikt Bollig, and Paul Gastin. 2015. An Automata-Theoretic Approach to the Verification of Distributed Algorithms. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*

(LIPIcs, Vol. 42), Luca Aceto and David de Frutos-Escrig (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 340–353. https://doi.org/10.4230/LIPIcs.CONCUR.2015.340

C. I. Arad. 2013. *Programming Model and Protocols for Reconfigurable Distributed Systems.* Ph.D. Dissertation. KTH Royal Institute of Technology.

Farhad Arbab. 2004. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical. Structures in Comp. Sci.* 14, 3 (June 2004), 329–366. https://doi.org/10.1017/S0960129504004153

Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On Formal Properties of Simple Phrase Structure Grammars. *Sprachtypologie und Universalienforschung* 14 (1961), 143–172.

Ananda Basu, Marius Bozga, and Joseph Sifakis. 2006. Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*. IEEE Computer Society, 3–12.

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification.* Morgan & Claypool Publishers.

Benedikt Bollig, Patricia Bouyer, and Fabian Reiter. 2019. Identifiers in Registers. In *Foundations of Software Science and Computation Structures*, Mikołaj Bojańczyk and Alex Simpson (Eds.), Vol. 11425. Springer International Publishing, Cham, 115–132.

Marius Bozga, Lucas Bueri, and Radu Iosif. 2022a. Decision Problems in a Logic for Reasoning about Reconfigurable Distributed Systems. In *International Joint Conference on Automated Reasoning (IJCAR 2022), to appear.*

Marius Bozga, Lucas Bueri, and Radu Iosif. 2022b. On an Invariance Problem for Parameterized Concurrent Systems. In *33rd International Conference on Concurrency Theory (CONCUR 2022), to appear.*

Marius Bozga, Javier Esparza, Radu Iosif, Joseph Sifakis, and Christoph Welzel. 2020. Structural Invariants for the Verification of Systems with Parameterized Architectures. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020 (LNCS, Vol. 12078)*. Springer, 228–246.

Marius Bozga and Radu Iosif. 2021. Specification and Safety Verification of Parametric Hierarchical Distributed Systems. In *Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28-29, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13077)*. Springer, 95–114.

Marius Bozga, Radu Iosif, and Joseph Sifakis. 2021. Verification of Component-based Systems with Recursive Architectures. *CoRR* abs/2112.08292 (2021). arXiv:2112.08292 https://arxiv.org/abs/2112.08292

Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. 2004. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems.* ACM, 28–33.

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent Separation Logic. *ACM SIGLOG News* 3, 3 (Aug. 2016), 47–65.

James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *J. Log. Comput.* 21, 6 (2011), 1177–1216.

Antonio Bucchiarone and Juan P. Galeotti. 2008. Dynamic Software Architectures Verification using DynAlloy. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 10 (2008).

Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2017. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description. In *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp) (CEUR Workshop Proceedings, Vol. 2019)*. CEUR-WS.org, 10–16.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

Jiannong Cao, Alvin T. S. Chan, and Yudong Sun. 2005. *GOP: A Graph-Oriented Programming Model for Parallel and Distributed Systems.* Springer US, Boston, MA, 21–36. https://doi.org/10.1007/0-387-28967-4_2

Everton Cavalcante, Thaís Vasconcelos Batista, and Flávio Oquendo. 2015. Supporting Dynamic Software Architectures: From Architectural Description to Implementation. In *12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*, Len Bass, Patricia Lago, and Philippe Kruchten (Eds.). IEEE Computer Society, 31–40.

Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (may 1979), 281–283. https://doi.org/10.1145/359104.359108

Yu-Fang Chen, Chih-Duo Hong, Anthony W. Lin, and Philipp Rümmer. 2017. Learning to prove safety over parameterised concurrent systems. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 76–83.

Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. 2014. Compositional assume-guarantee reasoning for input/output component theories. *Sci. Comput. Program.* 91 (2014), 115–137. https://doi.org/10.1016/j.scico.2013.12.010

Dave Clarke. 2008. A Basic Logic for Reasoning about Connector Reconfiguration. *Fundam. Inf.* 82, 4 (Feb. 2008), 361–390.

Loris D'Antoni and Margus Veanes. 2021. Automata modulo theories. *Commun. ACM* 64, 5 (2021), 86–95.

Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Vienna, Austria) *(ESEC/FSE-9)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/503209.503226

Stéphane Demri, Étienne Lozes, and Alessio Mansutti. 2018. The Effects of Adding Reachability Predicates in Propositional Separation Logic. In *FOSSACS 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*. Springer, 476–493. https://doi.org/10.1007/978-3-319-89366-2_26

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. *SIGPLAN Not.* 48, 1 (Jan. 2013), 287–300.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Vol. 6183. Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528.

Danny Dolev, Maria Klawe, and Michael Rodeh. 1982. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* 3, 3 (1982), 245–260. https://doi.org/10.1016/0196-6774(82)90023-2

Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. 2010. Using Temporal Logic for Dynamic Reconfigurations of Components. In *Formal Aspects of Component Software - 7th International Workshop, FACS 2010 (Lecture Notes in Computer Science, Vol. 6921)*, Luís Soares Barbosa and Markus Lumpe (Eds.). Springer, 200–217.

Mnacho Echenim, Radu Iosif, and Nicolas Peltier. 2020. Entailment Checking in Separation Logic with Inductive Definitions is 2-EXPTIME hard. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing, Vol. 73)*. EasyChair, 191–211. https://easychair.org/publications/paper/DdNg

Rim El-Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. 2021. Programming Dynamic Reconfigurable Systems. *International Journal on Software Tools for Technology Transfer* (January 2021).

Antoine El-Hokayem, Marius Bozga, and Joseph Sifakis. 2021. A temporal configuration logic for dynamic reconfigurable systems. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song (Eds.). ACM, 1419–1428. https://doi.org/10.1145/3412841.3442017

František Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2021. On Algebraic Abstractions for Concurrent Separation Logics. *Proc. ACM Program. Lang.* 5, POPL, Article 5 (Jan. 2021).

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Programming Languages and Systems*, Vol. 4421. Springer Berlin Heidelberg, 173–188.

Klaus-Tycho Foerster and Stefan Schmid. 2019. Survey of Reconfigurable Data Center Networks: Enablers, Algorithms, Complexity. *SIGACT News* 50, 2 (2019), 62–79. https://doi.org/10.1145/3351452.3351464

Ian Foster. 2002. What is the Grid? A Three Point Checklist. *GRID today* 1 (01 2002), 32–36.

Dominik Gall, Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Taeubig. 2014. A Note on the Parallel Runtime of Self-Stabilizing Graph Linearization. *Theory of Computing Systems (TOCS)* (2014).

Nikos Gorogiannis, Max I. Kanovich, and Peter W. O'Hearn. 2011. The Complexity of Abduction for Separated Heap Abstractions. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 25–42. https://doi.org/10.1007/978-3-642-23702-7_7

Dan Hirsch, Paolo Inverardi, and Ugo Montanari. 1998. Graph Grammars and Constraint Solving for Software Architecture Styles. In *Proceedings of the Third International Workshop on Software Architecture* (Orlando, Florida, USA) *(ISAW '98)*. Association for Computing Machinery, New York, NY, USA, 69–72. https://doi.org/10.1145/288408.288426

Radu Iosif and Florian Zuleger. 2022. On the Expressiveness of a Logic of Separated Relations. https://doi.org/10.48550/ARXIV.2208.01520

Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

Cliff B. Jones. 1981. *Developing methods for computer programs including a notion of interference.* Ph.D. Dissertation. University of Oxford, UK.

Jens Katelaan and Florian Zuleger. 2020. Beyond Symbolic Heaps: Deciding Separation Logic With Inductive Definitions. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing, Vol. 73)*. EasyChair, 390–408. https://easychair.org/publications/paper/VTGk

Igor V. Konnov, Tomer Kotek, Qiang Wang, Helmut Veith, Simon Bliudze, and Joseph Sifakis. 2016. Parameterized Systems in BIP: Design and Model Checking. In *27th International Conference on Concurrency Theory, CONCUR 2016 (LIPIcs, Vol. 59)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16.

Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. 2011. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Sci. Comput. Program.* 76, 1 (2011), 23–36. https://doi.org/10.1016/j.scico.2009.10.006

Joseph B. Kruskal. 1956. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. Amer. Math. Soc.* 7, 1 (1956), 48–50. http://www.jstor.org/stable/2033241

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 382–401. https://doi.org/10.1145/357172.357176

Arnaud Lanoix, Julien Dormoy, and Olga Kouchnarenko. 2011. Combining Proof and Model-checking to Validate Reconfigurable Architectures. *Electron. Notes Theor. Comput. Sci.* 279, 2 (2011), 43–57.

D. Le Metayer. 1998. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering* 24, 7 (1998), 521–533. https://doi.org/10.1109/32.708567

Nancy Lynch and M.R. Tuttle. 1989. An introduction to input/output automata. *CWI Quarterly* 2, 3 (Sept. 1989), 219–246.

Jeff Magee and Jeff Kramer. 1996. Dynamic structure in software architectures. In *ACM SIGSOFT Software Engineering Notes*, Vol. 21(6). ACM, 3–14.

Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. 2017. Configuration logics: Modeling architecture styles. *J. Log. Algebr. Meth. Program.* 86, 1 (2017), 2–29.

Othon Michail, George Skretas, and Paul G. Spirakis. 2020. Distributed Computation and Reconfiguration in Actively Dynamic Networks. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, Yuval Emek and Christian Cachin (Eds.). ACM, 448–457. https://doi.org/10.1145/3382734.3405744

Mohammad Noormohammadpour and Cauligi S. Raghavendra. 2018. Datacenter Traffic Control: Understanding Techniques and Tradeoffs. *IEEE Commun. Surv. Tutorials* 20, 2 (2018), 1492–1525.

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. 1–19.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2009. Separation and Information Hiding. *ACM Trans. Program. Lang. Syst.* 31, 3, Article 11 (April 2009), 50 pages. https://doi.org/10.1145/1498926.1498929

Susan Owicki and David Gries. 1978. *An Axiomatic Proof Technique for Parallel Programs.* Springer New York, New York, NY, 130–152.

Bruna Soares Peres, Otavio Augusto de Oliveira Souza, Olga Goussevskaia, Chen Avin, and Stefan Schmid. 2019. Distributed Self-Adjusting Tree Networks. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 145–153. https://doi.org/10.1109/INFOCOM.2019.8737417

R. C. Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. 2016. SplayNet: Towards Locally Self-Adjusting Networks. *IEEE/ACM Trans. Netw.* 24, 3 (June 2016), 1421–1433. https://doi.org/10.1109/TNET.2015.2410313

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-Style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects. *SIGPLAN Not.* 51, 10 (Oct. 2016), 92–110.

Ze'ev Shtadler and Orna Grumberg. 1989. Network Grammars, Communication Behaviors and Automatic Verification. In *Automatic Verification Methods for Finite State Systems, International Workshop (LNCS, Vol. 407)*, Joseph Sifakis (Ed.). Springer, 151–165.

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (July 1985), 652–686. https://doi.org/10.1145/3828.3835

Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. 1998. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *International Workshop on Theory and Application of Graph Transformations*. Springer, 179–193.

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*. Springer Berlin Heidelberg, 256–271.

Michel Wermelinger. 1998. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings-Software* 145, 5 (1998), 130–136.

Michel Wermelinger and José Luiz Fiadeiro. 2002. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.* 44, 2 (2002), 133–155.

Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. 2001. A Graph Based Architectural (Re)Configuration Language. *SIGSOFT Softw. Eng. Notes* 26, 5 (Sept. 2001), 21–32. https://doi.org/10.1145/503271.503213