Monitoring Distributed Component-Based Systems

Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2017-3

May 12, 2017

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF

Centre Équation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Monitoring Distributed Component-Based Systems

AUTEURS = Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga

Univ. Grenoble Alpes, CNRS, VERIMAG, Grenoble, France,
Univ. Grenoble Alpes, Inria, LIG, Grenoble, France
`Firstname.Lastname@imag.fr`

May 12, 2017

Abstract

This paper addresses the online monitoring of distributed component-based systems with multi-party interactions against user-provided properties expressed in linear-temporal logic and referring to global states. We consider intrinsically independent components whose interactions are partitioned on distributed controllers. In this context, the problem that arises is that a global state of the system is not available to the monitor. Instead, we attach local controllers to schedulers to retrieve the concurrent local traces. Local traces are sent to a global observer which reconstructs the set of global traces that are compatible with the local ones, in a concurrency-preserving fashion. The reconstruction of the global traces is done "on-the-fly" using a lattice of partial states encoding the global traces compatible with the locally-observed traces. We define an implementation of our framework in the BIP (Behavior, Interaction, Priority) framework, an expressive framework for the formal construction of heterogeneous distributed component-based systems. We define rigorous transformations of BIP components that preserve the semantics and the concurrency and, at the same time, allow to monitor global-state properties.

Keywords: component-based design, multiparty interaction, distributed systems, monitoring, runtime verification

Reviewers:

How to cite this report:

```
@techreport {TR-2017-3,
  title = {Monitoring Distributed Component-Based Systems},
  author = {Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2017-3},
  year = {2017}
}
```

1 Introduction

Distributed component-based systems with multi-party interactions. Component-based design consists in constructing complex systems using a set of predefined components. Each component is defined as an atomic entity with some actions and interfaces. Components communicate and interact with each other through their interfaces. The behavior of a component-based system (CBS) is defined according to the behavior of each component as well as the interactions between the components. Each interaction is a set of simultaneously-executed actions of the existing components [5]. In the distributed setting, for efficiency reasons, the execution of the interactions is distributed among several independent schedulers (also known as processors). Schedulers and components are interconnected (e.g., networked physical locations) and work together as a whole unit to meet some requirements. The execution of a multi-party interaction is then achieved by sending/receiving messages between the scheduler in charge of the execution of the interaction and the components involved in the interaction [1].

Problem statement. Verification techniques can be applied to ensure the correctness of a distributed CBS. Runtime Verification (RV) [19, 3, 9] consists in verifying the executions of the system against the desired specifications. In this work, our aim is to runtime verify a distributed CBS against properties referring to the global states of the system. This implies in particular that properties can not be "projected" and checked on individual components. In the following we point out the problems that one encounters when monitoring distributed CBSs at runtime. We use neither a global clock nor a shared memory. On the one hand, this makes the execution of the system more dynamic and parallel because we do not reconstruct global states at runtime. Thus, we avoid synchronization to take global snapshots, which would go against the distribution of the verified system. On the other hand, it complicates the monitoring problem because no component of the system can be aware of the global trace. Since the execution of interactions is based on sending/receiving messages, communications are asynchronous and delays in the reception of messages are inevitable. Moreover, the absence of ordering between the execution of the interactions in different schedulers causes the main problem in this case which is the actual global trace of the system is not observable. Our goal is to allow for the verification of distributed CBSs by formally instrumenting them to observe their global behavior while preserving their performance and initial behavior.

Approach overview. The main contribution of this paper is an approach for the runtime verification of distributed CBSs. Our concern is monitoring the behavior of a distributed system with respect to a given Linear Temporal Logic [18] (LTL) property φ which refers to the global states of the system. Intuitively, our method works as follows. First, we define a *monitoring hypothesis* based on the definition of an abstract model of distributed CBS with multi-party interactions. Our monitoring hypothesis is that the behavior of the monitored system comply to this model. This model is abstract enough to encompass a variety of distributed (component-based) systems. This model serves the purpose of describing the knowledge needed on the verified system and later guides their instrumentation.

In the distributed CBS, due to the parallel executions in different schedulers i) we have a set of events (i.e., actions which change the state of the system) which are not totally ordered, and ii) the actual global trace of a distributed system can not be obtained. Although each scheduler is only aware of its local trace (i.e., a set of ordered events), in order to evaluate the global behavior of the system, it is necessary to find a set of possible ordering among the events of all schedulers, that is, the set of compatible global traces. In our setting, schedulers do not communicate together and only communicate with their own associated components. Indeed, what makes the actions of different schedulers to be causally related is only the shared components which are involved in several multi-party interactions managed by different schedulers. In other words, the executions of two actions managed by two schedulers and involving a shared component are definitely causally related, because each execution requires the termination of the other execution in order to release the shared component. To take into account these existing causalities among the events, we i) employ vector clock to define the global order of events, ii) compose each scheduler with a controller to compute the correct vector clock of each generated event, iii) compose each shared component with a controller to resolve the causality, and iv) introduce a centralized entity as an observer module to accumulates events (i.e., local traces). At runtime the central observer collects the set of received local traces of schedulers and reconstructs a set of compatible global traces that could possibly happen in the system, that is a form of general notion of computation lattice. To evaluate the computation lattice with respect to a given LTL property, we define a novel on-the-fly progression method over the constructed computation lattice. To this end, we define a new structure of computation lattice in which each node n of the lattice is augmented by a set of formulas representing the evaluation of all the possible global traces from the initial node of the lattice (i.e., initial

state of the system) up to node n.

Outline. The remainder of this paper is organized as follows. Section 2 introduces some preliminary concepts. Section 3 defines an original abstract model of distributed CBSs, suitable for monitoring purposes, and allowing to define a monitoring hypothesis for the runtime verification of distributed CBSs. In Sec. 4, we present the instrumentation used to generate the events of each scheduler which are aimed to be used in the construction of the global trace of a distributed CBS. In Sec. 5, we construct the computation lattice by collecting the events from the different schedulers. Runtime verification of distributed CBSs is presented in Sec. 6. Section 7 describes RVDIST, a C++ implementation of the monitoring framework used to carry an evaluation of our approach described in Sec. 8. Section 9 presents related work. Section 10 concludes and presents future work. Proofs of the propositions are in Appendix **??**.

2 Preliminaries and Notations

Sequences. Considering a finite set of elements E, we define notations about sequences of elements of E. A sequence s containing elements of E is formally defined by a total function $s : I \rightarrow E$ where I is either the integer interval $[0, n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself (the set of natural numbers). Given a set of elements E, $e_1 \cdot e_2 \cdots e_n$ is a sequence or a list of length n over E, where $\forall i \in [1, n] : e_i \in E$. The empty sequence is noted ϵ or $[\,]$, depending on the context. The set of (finite) sequences over E is noted E^*. E^+ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence s is noted $\mathrm{length}(s)$. We define $s(i)$ as the ith element of s and $s(i \cdots j)$ as the factor of s from the ith to the jth element. $s(i \cdots j) = \epsilon$ if $i > j$. We also note $\mathrm{pref}(s)$, the set of non-empty *prefixes* of s, i.e., $\mathrm{pref}(s) = \{s(1 \cdots k) \mid 1 \leq k \leq \mathrm{length}(s)\}$. Operator pref is naturally extended to sets of sequences. We define function last : $E^+ \rightarrow E$ as $\mathrm{last}(e) = s(|e|)$. For an infinite sequence $s = e_1 \cdot e_2 \cdot e_3 \cdots$, we define $s(i \cdots) = e_i \cdot e_{i+1} \cdots$ as the suffix of sequence s from index i on.

Tuples. An n-tuple is an ordered list of n elements, where n is a strictly positive integer. By $t[i]$ we denote i^{th} element of tuple t.

Labeled transition systems. Labeled Transition Systems (LTSs) are used to define the semantics of CBSs. An LTS is defined over an alphabet Σ and is a 3-tuple $(\mathrm{State}, \mathrm{Lab}, \mathrm{Trans})$ where State is a non-empty set of states, Lab is a set of labels, and $\mathrm{Trans} \subseteq \mathrm{State} \times \mathrm{Lab} \times \mathrm{State}$ is the transition relation. A transition $(q, a, q') \in \mathrm{Trans}$ means that the LTS can move from state q to state q' by consuming label a. We abbreviate $(q, a, q') \in \mathrm{Trans}$ by $q \xrightarrow{a}_{\mathrm{Trans}} q'$ or by $q \xrightarrow{a} q'$ when clear from context. Moreover, relation Trans is extended to its reflexive and transitive closure in the usual way and we allow for regular expressions over Lab to label moves between states: if $expr$ is a regular expression over Lab (i.e., $expr$ denotes a subset of Lab^*), $q \xrightarrow{expr} q'$ means that there exists one sequence of labels in Lab matching $expr$ such that the system can move from q to q'.

Observational equivalence and bi-simulation. The *observational equivalence* of two transition systems is based on the usual definition of weak bisimilarity [15], where θ-transitions are considered to be unobservable. Given two transition systems $S_1 = (\mathrm{Sta}_1, \mathrm{Lab} \cup \{\theta\}, \rightarrow_2)$ and $S_2 = (\mathrm{Sta}_2, \mathrm{Lab} \cup \{\theta\}, \rightarrow_2)$, system S_1 *weakly simulates* system S_2, if there exists a relation $R \subseteq \mathrm{Sta}_1 \times \mathrm{Sta}_2$ that contains the 2-tuple made of the initial states of S_1 et S_2 and such that the two following conditions hold:

1. $\forall (q_1, q_2) \in R, \forall a \in \mathrm{Lab} : q_1 \xrightarrow{a}_1 q_1' \implies \exists q_2' \in \mathrm{Sta}_2 : \left((q_1', q_2') \in R \wedge q_2 \xrightarrow{\theta^* \cdot a \cdot \theta^*}_2 q_2'\right)$, and
2. $\forall (q_1, q_2) \in R : \left(\exists q_1' \in \mathrm{Sta}_1 : q_1 \xrightarrow{\theta}_1 q_1'\right) \implies \exists q_2' \in \mathrm{Sta}_2 : \left((q_1', q_2') \in R \wedge q_2 \xrightarrow{\theta^*}_2 q_2'\right)$.

Equation 1. states that if a state q_1 simulates a state q_2 and if it is possible to perform a from q_1 to end in a state q_1', then there exists a state q_2' simulated by q_1' such that it is possible to go from q_2 to q_2' by performing some unobservable actions, the action a, and then some unobservable actions. Equation 2. states that if a state q_1 simulates a state q_2 and it is possible to perform an unobservable action from q_1 to reach a state q_1', then it is possible to reach a state q_2' by a sequence of unobservable actions such that q_1' simulates q_2'. In that case, we say that relation R is a weak simulation over S_1 and S_2 or equivalently that the states of S_1 are (weakly) similar to the states of S_2. Similarly, a weak bi-simulation over S_1 and S_2 is a relation R such that R and

$R^{-1} = \{(q_2, q_1) \in \mathrm{Sta}_2 \times \mathrm{Sta}_1 \mid (q_1, q_2) \in R\}$ are both weak simulations. In this latter case, we say that S_1 and S_2 are *observationally equivalent* and we write $S_1 \sim S_2$ to express this formally.

Vector Clock. Lamport introduced logical clocks as a device to substitute for a global real time clock [12]. Logical clocks are used to order events based on their relative logical dependencies rather than on a "time" in the common sense. Vector clocks are a more powerful extension (i.e., strongly consistent with the ordering of events) of Lamport's scalar logical clocks [6]. In a distributed system with a set of schedulers $\{S_1, \ldots, S_m\}$, $VC = \{(c_1, \ldots, c_m) \mid j \in [1, m] \wedge c_j \in \mathbb{N}\}$ is the set of vector clocks, such that vector clock $vc \in VC$ is a tuple of m scalar (initially zero) values $c_1, \ldots, c_m$ locally stored in each scheduler $S_j \in \{S_1, \ldots, S_m\}$ where $\forall k \in [1, m] : vc[k] = c_k$ holds the latest (scalar) clock value scheduler S_j knows about scheduler $S_k \in \{S_1, \ldots, S_m\}$. Each event in the system is associated to a unique vector clock. For two vector clocks vc_1 and vc_2, $\max(vc_1, vc_2)$ is a vector clock vc_3 such that $\forall k \in [1, m] : vc_3[k] = \max(vc_1[k], vc_2[k])$. $\min(vc_1, vc_2)$ is defined in similar way. Moreover two vector clocks can be compared together such that $vc_1 < vc_2 \iff \forall k \in [1, m] : vc_1[k] \leq vc_2[k] \wedge \exists z \in [1, m] : vc_1[z] < vc_2[z]$.

Happened-before relation [12]. The relation $\rightarrowtail$ on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same scheduler, and a comes before b, then $a \rightarrowtail b$. (2) If a is the sending of a message by one scheduler and b is the reception of the same message by another scheduler, then $a \rightarrowtail b$. (3) If $a \rightarrowtail b$ and $b \rightarrowtail c$ then $a \rightarrowtail c$. Two distinct events a and b are said to be concurrent if $a \not\rightarrowtail b$ and $b \not\rightarrowtail a$.

Vector clocks are strongly consistent with happened-before relation. That is, for two events a and b with associated vector clocks vc_a and vc_b respectively, $vc_a < vc_b \iff a \rightarrowtail b$.

Computation lattice [14]. The computation lattice of a distributed system is represented in the form of a directed graph with m (i.e., number of schedulers that are executed in distributed manner) orthogonal axes. Each axis is dedicated to the state evolution of a specific scheduler. A computation lattice expresses all the possible traces in a distributed system. Each path in the lattice represents a global trace of the system that could possibly have happened. A computation lattice $\mathcal{L}$ is a pair $(N, \rightarrowtail)$, where N is the set of nodes (i.e., global states) and $\rightarrowtail$ is the set of happened-before relations among the nodes.

Linear Temporal Logic (LTL) [18]. Linear temporal logic (LTL) is a formalism for specifying properties of systems. An LTL formula is built over a set of atomic propositions AP. LTL formulas are written with the following grammar:

$$\varphi ::= \quad p \quad | \quad \neg\varphi \quad | \quad \varphi_1 \vee \varphi_2 \quad | \quad \mathbf{X}\varphi \quad | \quad \varphi_1 \mathbf{U} \varphi_2$$

where $p \in AP$ is an atomic proposition. Note that we use only the **X** and **U** modalities for defining the valid formulas in LTL. The other modalities such as **F** (eventually), **G** (globally), **R** (release), *etc.* in LTL can be defined using the **X** and **U** modalities.

Let $\sigma = q_0 \cdot q_1 \cdot q_2 \cdots$ be an infinite sequence of states and $\models$ denotes the satisfaction relation. The semantics of LTL is defined inductively as follows:

- $\sigma \models p \iff q_0 \models p$ (i.e., $p \in q_0$), for any $p \in AP$
- $\sigma \models \neg\varphi \iff \sigma \not\models \varphi$
- $\sigma \models \varphi_1 \vee \varphi_2 \iff \sigma \models \varphi_1 \vee \sigma \models \varphi_2$
- $\sigma \models \mathbf{X}\varphi \iff \sigma(1 \cdots) \models \varphi$
- $\sigma \models \varphi_1 \mathbf{U} \varphi_2 \iff \exists j \geqslant 0 : \sigma(j \cdots) \models \varphi_2 \wedge \sigma(i \cdots) \models \varphi_1, 0 \leqslant i < j$

An atomic proposition p is satisfied by σ when it is member of the first state of σ. σ satisfies formula $\neg\varphi$ when it does not satisfy φ. Disjunction of φ_1 and φ_2 is satisfied when either φ_1 or φ_2 is satisfied by σ. σ satisfies formula $\mathbf{X}\varphi$ when the sequence of states starting from the next state of σ, that is, q_1 satisfies φ. $\varphi_1 \mathbf{U} \varphi_2$ is satisfied when φ_2 is satisfied at some point and φ_1 is satisfied until that point.

Pattern-matching. We shall use the mechanism of pattern-matching to concisely define some functions. We recall an intuitive definition for the sake of completeness. Evaluating the expression:

```
match expression with
    | pattern_1 → expression_1
    | pattern_2 → expression_2
    ...
    | pattern_n → expression_n
```

consists in comparing successively `expression` with the patterns `pattern_1`, ..., `pattern_n` in order. When a pattern `pattern_i` fits `expression`, then the associated `expression_i` is returned.

3 Distributed CBSs with Multi-Party Interactions

In the following, we describe our assumptions on the considered distributed component-based systems with multi-party interactions. To this end, we assume a general semantics to define the behavior of the distributed system under scrutiny in order to make our monitoring approach as general as possible. However, neither the exact model nor the behavior of the system are known. How the behaviors of the components and the schedulers are obtained is irrelevant. Inspiring from conformance-testing theory [26], we refer to this hypothesis as the **monitoring hypothesis**.

Consequently, our monitoring approach can be applied to (component-based) systems whose behavior can be modeled as described in the sequel. The semantics of the following model is similar to and compatible with other models for describing distributed computations (see Sec. 9 for a comparison with other models and possible translations between models). The remainder of this section is organized as follows. Subsection 3.1 defines an abstract distributed component-based model. Subsection 3.2 defines the execution traces of the abstract model, later used for runtime verification.

3.1 Semantics of a Distributed CBS with Multi-Party Interactions

Architecture of the system. The system under scrutiny is composed of *components* in a non-empty set $\mathbf{B} = \{B_1, \ldots, B_{|\mathbf{B}|}\}$ and *schedulers* in a non-empty set $\mathbf{S} = \{S_1, \ldots, S_{|\mathbf{S}|}\}$. Each component B_i is endowed with a set of actions Act_i. Joint actions of component, aka multi-party interactions, involve the execution of actions on several components. An interaction is a non-empty subset of $\cup_{i=1}^{|\mathbf{B}|} Act_i$ and we denote by *Int* the set of interactions in the system. At most one action of each component is involved in an interaction: $\forall a \in \mathit{Int} : |a \cap Act_i| \leq 1$. In addition, each component B_i has internal actions which we model as a unique action β_i. Schedulers coordinate the execution of interactions and ensure that each multi-party interaction is jointly executed (*cf.* Definition 2).

Let us assume some auxiliary functions obtained from the architecture of the system.

- Function $\mathrm{involved} : \mathit{Int} \to 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the components involved in an interaction. Moreover, we extend function involved to internal actions by setting $\mathrm{involved}(\beta_i) = i$, for any $\beta_i \in \{\beta_1, \ldots, \beta_{|\mathbf{B}|}\}$. Interaction $a \in \mathit{Int}$ is a joint action if and only if $|\mathrm{involved}(a)| \geq 2$.
- Function $\mathrm{managed} : \mathit{Int} \to \mathbf{S}$ indicates the scheduler managing an interaction: for an interaction $a \in \mathit{Int}$ $\mathrm{managed}(a) = S_j$ if a is managed by scheduler S_j.
- Function $\mathrm{scope} : \mathbf{S} \to 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the set of components in the scope of a scheduler such that $\mathrm{scope}(S_j) = \bigcup_{a' \in \{a \in \mathit{Int} \mid \mathrm{managed}(a) = S_j\}} \mathrm{involved}(a')$.

In the remainder, we describe the behavior of components, schedulers, and their composition.

Components. The behavior of an individual component is defined as follows.

Definition 1 (Behavior of a component) *The behavior of a component B is defined as an LTS $(Q_B, Act_B \cup \{\beta_B\}, \to_B)$ such that:*

- *$Q_B = Q_B^{\mathrm{r}} \cup Q_B^{\mathrm{b}}$ is the set of states, where Q_B^{r} (resp. Q_B^{b}) is the so-called set of* ready *(resp.* busy*) states,*

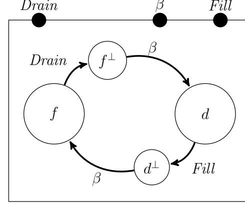


Figure 1: Component Tank

- Act_B *is the set of actions, and* β_B *is the internal action,*
- $\rightarrow_B \subseteq \left(Q^{\mathrm{r}}_B \times Act_B \times Q^{\mathrm{b}}_B\right) \cup \left(Q^{\mathrm{b}}_B \times \{\beta_B\} \times Q^{\mathrm{r}}_B\right)$ *is the set of transitions.*

Moreover, Q_B *has a partition* $\left\{Q^{\mathrm{r}}_B, Q^{\mathrm{b}}_B\right\}$.

Intuitively, the set of ready (resp. busy) states Q^{r}_B (resp. Q^{b}_B) is the set of states such that the component is ready (resp. not ready) to perform an action. Component B (i) has actions in set Act_B which are possibly shared with some of the other components, (ii) has an internal action β_B such that $\beta_B \notin Act_B$ which models internal computations of component B, and (iii) alternates moving from a ready state to a busy state and from a busy state to a ready state, that is component B does not have busy to busy or ready to ready move (as defined in the transition relation above).

Example 1 (Component) *Figure 1 depicts a component* Tank *whose behavior is defined by the LTS* $(Q^{\mathrm{r}} \cup Q^{\mathrm{b}}, Act \cup \{\beta\}, \rightarrow)$ *such that:*

- $Q^{\mathrm{r}} = \{d, f\}$ *is the set of* ready *states and* $Q^{\mathrm{b}} = \left\{d^{\perp}, f^{\perp}\right\}$ *is the set of* busy *states,*
- $Act = \{\mathit{Drain}, \mathit{Fill}\}$ *is the set of actions and* β *is the internal action,*
- $\rightarrow = \left\{(d, \mathit{Fill}, d^{\perp}), (d^{\perp}, \beta, f), (f, \mathit{Drain}, f^{\perp}), (f^{\perp}, \beta, d)\right\}$ *is the set of transitions.*

On the border, each • represents an action and provides an interface for the component to synchronize with actions of other components in case of joint actions.

In the following, we assume that each component $B_i \in \mathbf{B}$ is defined by the LTS $(Q_{B_i}, Act_{B_i} \cup \{\beta_{B_i}\}, \rightarrow_{B_i})$ where Q_{B_i} has a partition $\left\{Q^{\mathrm{r}}_{B_i}, Q^{\mathrm{b}}_{B_i}\right\}$ of ready and busy states; as per Definition 1.

Schedulers. The behavior of a scheduler is defined as follows.

Definition 2 (Behavior of a scheduler) *The behavior of a scheduler* S *is defined as an LTS* $(Q_S, Act_S, \rightarrow_S)$ *such that:*

- Q_S *is the set of states,*
- $Act_S = Act^{\gamma}_S \cup Act^{\beta}_S$ *is the set of actions, where* $Act^{\gamma}_S = \{a \in \mathit{Int} \mid \mathrm{managed}(a) = S\}$ *and* $Act^{\beta}_S = \{\beta_i \mid B_i \in \mathrm{scope}(S)\}$,
- $\rightarrow_S \subseteq Q_S \times Act_S \times Q_S$ *is the set of transitions.*

$Act^{\gamma}_S \subseteq \mathit{Int}$ is the set of interactions managed by S, and Act^{β}_S is the set of internal actions of the components involved in an action managed by S.

In the following, we assume that each scheduler $S_j \in \mathbf{S}$ is defined by the LTS $(Q_{S_j}, Act_{S_j}, \rightarrow_{S_j})$ where $Act_{S_j} = Act^{\gamma}_S \cup Act^{\beta}_S$; as per Definition 2. The coordination of interactions of the system i.e., the interactions in Int, is distributed among schedulers. Actions of schedulers consist of interactions of the system. Nevertheless, each interaction of the system is associated to exactly one scheduler ($\forall a \in \mathit{Int}, \exists! S \in \mathbf{S} : a \in Act_S$). Consequently, schedulers manage disjoint sets of interactions (i.e., $\forall S_i, S_j \in \mathbf{S} : S_i \neq S_j \implies Act^{\gamma}_{S_i} \cap Act^{\gamma}_{S_j} = \emptyset$). Intuitively, when a scheduler executes an interaction, it triggers the execution of the associated actions on the involved components. Moreover, when a component executes an internal action, it triggers the execution of the corresponding action on the associated schedulers and also sends the updated state of the component to the associated schedulers, that is, the component sends a message including its current state to the schedulers. Note, we assume that, by construction, schedulers are always ready to receive such a state update.

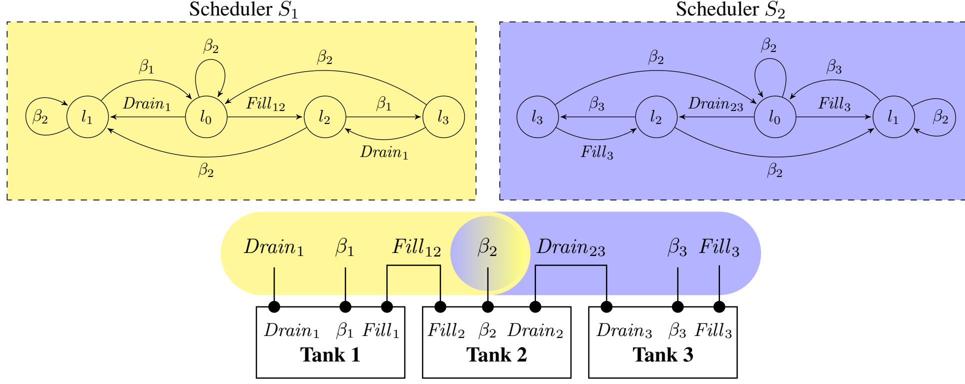


Figure 2: Abstract representation of a distributed CBS

Remark 1 *Since components send their update states to the associated schedulers, we assume that the current state of a scheduler contains the last state of each component in its scope.*

Example 2 (Scheduler of distributed CBS) *Figure 2 depicts a distributed component-based system consisting of three components each of which is an instance of the component in Figure 1. The set of interactions is* $Int = \{\{Drain_1\}, Fill_{12}, Drain_{23}, \{Fill_3\}\}$ *where* $Fill_{12} = \{Fill_1, Fill_2\}$ *and* $Drain_{23} = \{Drain_2, Drain_3\}$ *are joint actions. Two schedulers S_1 and S_2 coordinate the execution of interactions such that* $\text{managed}(\{Drain_1\}) = \text{managed}(Fill_{12}) = S_1$ *and* $\text{managed}(\{Fill_3\}) = \text{managed}(Drain_{23}) = S_2$*. For $j \in [1, 2]$, scheduler S_j is defined as* $(Q_{S_j}, Act_{S_j}, \rightarrow_{S_j})$ *with:*

- $Q_{S_j} = \{l_0, l_1, l_2, l_3\}$,
- $Act^{\gamma}_{S_1} = \{\{Drain_1\}, Fill_{12}\}$, $Act^{\beta}_{S_1} = \{\beta_1, \beta_2\}$,
- $Act^{\gamma}_{S_2} = \{Drain_{23}, \{Fill_3\}\}$, $Act^{\beta}_{S_2} = \{\beta_2, \beta_3\}$,
- $\rightarrow_{S_1} = \{(l_0, \beta_2, l_0), (l_1, \beta_2, l_1), (l_1, \beta_1, l_0), (l_2, \beta_2, l_1), (l_3, \beta_2, l_0), (l_2, \beta_1, l_3), (l_3, \{Drain_1\}, l_2), (l_0, \{Drain_1\}, l_1), (l_0, Fill_{12}, l_2)\}$,
- $\rightarrow_{S_2} = \{(l_0, \beta_2, l_0), (l_1, \beta_2, l_1), (l_1, \beta_3, l_0), (l_2, \beta_2, l_1), (l_3, \beta_2, l_0), (l_2, \beta_3, l_3), (l_3, \{Fill_3\}, l_2), (l_0, \{Fill_3\}, l_1), (l_0, Drain_{23}, l_2)\}$.

Definition 3 (Shared component) *The set of shared components is defined as*

$$\mathbf{B}_s = \{B \in \mathbf{B} \mid |\{S \in \mathbf{S} \mid B \in \text{scope}(S)\}| \geq 2\}.$$

A shared component $B \in \mathbf{B}_s$ is a component in the scope of more than one scheduler, and thus, the execution of the actions of B are managed by more than one scheduler.

Example 3 (Shared component) *In Figure 2, component $Tank_2$ is a shared component because interaction $Fill_{12}$, which is a joint action of $Fill_1$ and $Fill_2$, is coordinated by scheduler S_1 and interaction $Drain_{23}$, which is a joint action of $Drain_2$ and $Drain_3$, is coordinated by scheduler S_2.*

The global execution of the system can be described as the parallel execution of interactions managed by the schedulers.

Definition 4 (Global behavior) *The global behavior of the system is the LTS $(Q, GAct, \rightarrow)$ where:*

- $Q \subseteq \bigotimes_{i=1}^{|\mathbf{B}|} Q_i \times \bigotimes_{j=1}^{|\mathbf{S}|} Q_{S_j}$ *is the set of states consisting of the states of schedulers and components,*

- $GAct \subseteq 2^{Act \,\cup\, \bigcup_{i=1}^{|\mathbf{B}|}\{\beta_i\}} \setminus \{\emptyset\}$ *is the set of possible global actions of the system consisting of either several interactions and/or several internal actions (several interactions can be executed concurrently by the system),*

- $\rightarrow \subseteq Q \times GAct \times Q$ *is the transition relation defined as the smallest set abiding to the following rule. A transition is a move from state* $(q_1, \ldots, q_{|\mathbf{B}|}, q_{s_1}, \ldots, q_{s_{|\mathbf{S}|}})$ *to state* $(q'_1, \ldots, q'_{|\mathbf{B}|}, q'_{s_1}, \ldots, q'_{s_{|\mathbf{S}|}})$ *on global actions in set* $\alpha \cup \beta$*, where* $\alpha \subseteq Int$ *and* $\beta \subseteq \bigcup_{i=1}^{|\mathbf{B}|}\{\beta_i\}$*, noted* $(q_1, \ldots, q_{|\mathbf{B}|}, q_{s_1}, \ldots, q_{s_{|\mathbf{S}|}}) \xrightarrow{\alpha\cup\beta} (q'_1, \ldots, q'_{|\mathbf{B}|}, q'_{s_1}, \ldots, q'_{s_{|\mathbf{S}|}})$*, whenever the following conditions hold:*

  C_1: $\forall i \in [1, |\mathbf{B}|] : |(\alpha \cap Act_i) \cup (\{\beta_i\} \cap \beta)| \leq 1$,

  C_2: $\forall a \in \alpha : (\exists S_j \in \mathbf{S} : \mathrm{managed}(a) = S_j) \Rightarrow \left(q_{s_j} \xrightarrow{a}_{S_j} q'_{s_j} \wedge \forall B_i \in \mathrm{involved}(a) : q_i \xrightarrow{a\cap Act_i}_{B_i} q'_i\right)$,

  C_3: $\forall \beta_i \in \beta : q_i \xrightarrow{\beta_i}_{B_i} q'_i \wedge \forall S_j \in \mathbf{S} : B_i \in \mathrm{scope}(S_j) : q_{s_j} \xrightarrow{\beta_i}_{S_j} q'_{s_j}$,

  C_4: $\forall B_i \in \mathbf{B} \setminus \mathrm{involved}(\alpha \cup \beta) : q_i = q'_i$,

  C_5: $\forall S_j \in \mathbf{S} \setminus \mathrm{managed}(\alpha) : q_{s_j} = q'_{s_j}$.

  where functions involved *and* managed *are extended to sets of interactions and internal actions in the usual way.*

The above rule allows the components of the system to execute independently according to the decisions of the schedulers. It can intuitively be understood as follows:

- *Condition* C_1 states that a component can perform at most one execution step at a time. The executed global actions ($\alpha \cup \beta$) contains at most one interaction involving each component of the system.

- *Condition* C_2 states that whenever an interaction a managed by scheduler S_j is executed, S_j and all components involved in this multi-party interaction must be ready to execute it.

- *Condition* C_3 states that internal actions are executed whenever the corresponding components are ready to execute them. Moreover, schedulers are aware of internal actions of components in their scope. Note that, the awareness of internal actions of a component results in transferring the updated state of the component to the schedulers.

- *Conditions* C_4 *and* C_5 state that the components and the schedulers not involved in an interaction remain in the same state.

An example illustrating the global behavior of the system depicted in Figure 2 is provided later and described in terms of execution traces (*cf.* Example 4).

Remark 2 *The operational description of a distributed CBS has is usually more detailed. For instance, the execution of conflicting interactions in schedulers needs first to be authorized by a conflict-resolution module which guarantees that two conflicting interactions are not executed at the same time. Moreover, schedulers follow the (possible) priority rules among the interactions, that is, in the case of two or more enabled interactions (interactions which are ready to be executed by schedulers), those with higher priority are allowed to be executed. Since we only deal with the execution traces of a distributed system, we assume that the obtained traces are correct with respect to the conflicts and priorities. Therefore, defining the other modules is out of the scope of this work.*

Definition 5 (Monitoring hypothesis) *The behavior of the distributed CBS under scrutiny can be modeled as an LTS as per Definition 4.*

3.2 Traces of a Distributed CBS with Multi-Party Interactions

At runtime, the execution of a component-based system with multi-party interactions produces a global trace of events. Intuitively, a global trace is the sequence of traversed states of the system, from some initial state and following the transition relation of the LTS of the system. For the sake of simplicity and for our monitoring purposes, the states of schedulers are irrelevant in the trace and thus we restrict the global states to states of the components.

We consider a distributed component-based system consisting of a set $\mathbf{B}$ of components (as per Definition 1) and a set $\mathbf{S}$ of schedulers (as per Definition 2) with the global behavior as per Definition 4.

Definition 6 (Global trace of a distributed CBS) *A global trace of a distributed CBS is a continuously-growing sequence* $(q_1^0, \ldots, q_{|\mathbf{B}|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \ldots, q_{|\mathbf{B}|}^1) \cdots (q_1^k, \ldots, q_{|\mathbf{B}|}^k) \cdots$*, such that* $q_1^0, \ldots, q_{|\mathbf{B}|}^0$ *are the initial states of components* $B_1, \ldots, B_{|\mathbf{B}|}$ *and* $\forall i \in [0, k-1] : (q_1^i, \ldots, q_{|\mathbf{B}|}^i) \xrightarrow{\alpha^i \cup \beta^i} (q_1^{i+1}, \ldots, q_{|\mathbf{B}|}^{i+1})$*, where* $\rightarrow$ *is the transition relation of the global behavior of the system and the states of schedulers are discarded.*

Although the global trace of the system exists, it is not observable because it would require a perfect observer having simultaneous access to the states of the components. Introducing such an observer (able to observe global states) in the system would require all components to synchronize, and would defeat the purpose of building a distributed system. Instead of introducing such an observer, we shall instrument the system (see Sec. 4) to observe the sequence of states through schedulers.

In the following we consider a global trace $t = (q_1^0, \ldots, q_{|\mathbf{B}|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \ldots, q_{|\mathbf{B}|}^1) \cdots$, as per Definition 6. Each scheduler $S_j \in \mathbf{S}$, observes a local trace $s_j(t)$ which consists in the sequence of state-evolutions of the components it manages.

Definition 7 (The observable local trace of a scheduler obtained from a global trace) *The local trace* $s_j(t)$ *observed by scheduler* S_j *is inductively defined on the global trace* t *as follows:*

- $s_j\left(\left(q_1^0, \ldots, q_{|\mathbf{B}|}^0\right)\right) = \left(q_1^0, \ldots, q_{|\mathbf{B}|}^0\right)$*, and*
- $s_j\left(t \cdot (\alpha \cup \beta) \cdot (q_1, \ldots, q_{|\mathbf{B}|})\right) = \begin{cases} t & \text{if } S_j \notin \mathrm{managed}(\alpha) \wedge (\mathrm{involved}(\beta) \cap \mathrm{scope}(S_j) = \emptyset) \\ t \cdot \gamma \cdot q' & \text{otherwise} \end{cases}$

  where

  - $\gamma = (\alpha \cap \{a \in \mathit{Int} \mid \mathrm{managed}(a) = S_j\}) \cup (\beta \cap \{\beta_i \mid B_i \in \mathrm{scope}(S_j)\})$
  - $q' = (q'_1, \ldots, q'_{|\mathbf{B}|})$ *with*

    $$q'_i = \begin{cases} \mathrm{last}(s_j(t))[i] & \text{if } B_i \in \overline{\mathrm{involved}(\gamma)} \cap \mathrm{scope}(S_j), \\ q_i & \text{if } B_i \in \mathrm{involved}(\gamma) \cap \mathrm{scope}(S_j), \\ ? & \text{otherwise } (B_i \notin \mathrm{scope}(S_j)). \end{cases}$$

We assume that the initial state of the system is observable by all schedulers. An interaction $a \in \mathit{Int}$ is observable by scheduler S_j if S_j manages the interaction (i.e., $S_j \in \mathrm{managed}(a)$). Moreover, an internal action β_i, $i \in [1, |\mathbf{B}|]$, is observable by scheduler S_j if B_i is in the scope of S_j (i.e., $B_i \in \mathrm{scope}(S_j)$). The state observed after an observable interaction or internal action consists of the states of components in the scope of S_j, that is a state $(q_1, \ldots, q_{|\mathbf{B}|})$ where q_i is the new state of component B_i if $B_i \in \mathrm{scope}(S_j)$ and ? otherwise.

Example 4 (Global trace and local trace) *Two possible global traces of the system in Example 2 (depicted in Figure 2) are:*[1]

- $t_1 = (d_1, d_2, d_3) \cdot \{\mathit{Fill}_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{\mathit{Drain}_1\}, \{\mathit{Fill}_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$,
- $t_2 = (d_1, d_2, d_3) \cdot \{\mathit{Fill}_{12}, \{\mathit{Fill}_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_3\} \cdot (\bot, \bot, f_3) \cdot \{\beta_2\} \cdot (\bot, f_2, f_3) \cdot \{\{\mathit{Drain}_{23}\}, \beta_1\} \cdot (f_1, \bot, \bot)$.

Traces t_1 *and* t_2 *are obtained following the global behavior of the system (Definition 4).*

- *In trace* t_1*, the execution of interaction* Fill_{12} *represents the simultaneous execution of (i) action* Fill_{12} *in scheduler* S_1*, (ii) action* Fill_1 *in component* Tank_1*, and (iii) action* Fill_2 *in component* Tank_2*. After interaction* Fill_{12}*, component* Tank_1 *and* Tank_2 *move to their busy state whereas the state of component* Tank_3 *remains unchanged. Moreover, the execution of internal action* β_2 *in trace* t_1 *represents the simultaneous execution of (i) internal action* β_2 *in component* Tank_2*, (ii) action* β_2 *in scheduler* S_1 *and (iii) action* β_2 *in scheduler* S_2*. After the internal action* β_2*, component* Tank_2 *goes to ready state* f_2*.*
- *In trace* t_2*, the execution of global action* $\{\mathit{Fill}_{12}, \{\mathit{Fill}_3\}\}$ *represents the simultaneous execution two interactions* Fill_{12} *and* $\{\mathit{Fill}_3\}$ *that is the simultaneous executions of (i) action* Fill_{12} *in scheduler* S_1*, (ii) action* Fill_3 *in scheduler* S_2*, (iii) action* Fill_1 *in component* Tank_1*, (iv) action* Fill_2 *in component* Tank_2*, and (v) action* Fill_3 *in component* Tank_3*. Trace* t_2 *ends up with the simultaneous execution of interaction* Drain_{23} *and the internal action of component* Tank_1*.*

[1] To facilitate the description of the trace, we represent each busy state as $\bot$.

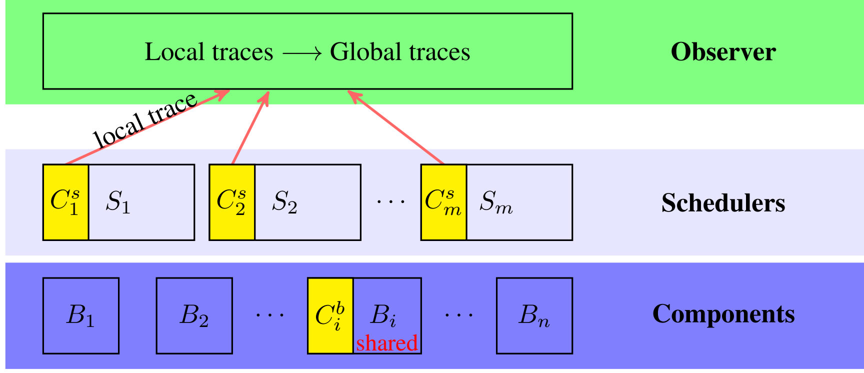


Figure 3: Passive observer

The associated local traces are:

- $s_1(t_1) = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, ?) \cdot \{\beta_1\} \cdot (f_1, \bot, ?) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, ?) \cdot \{\beta_2\} \cdot (\bot, f_2, ?)$,
- $s_2(t_1) = (d_1, d_2, d_3) \cdot \{\{Fill_3\}\} \cdot (?, d_2, \bot) \cdot \{\beta_2\} \cdot (?, f_2, \bot)$,
- $s_1(t_2) = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, ?) \cdot \{\beta_2\} \cdot (\bot, f_2, ?) \cdot \{\beta_1\} \cdot (f_1, f_2, ?)$,
- $s_2(t_2) = (d_1, d_2, d_3) \cdot \{\{Fill_3\}\} \cdot (?, d_2, \bot) \cdot \{\beta_3\} \cdot (?, d_2, f_3) \cdot \{\beta_2\} \cdot (?, f_2, f_3) \cdot \{Drain_{23}\} \cdot (?, \bot, \bot)$.

For instance, the local trace $s_1(t_2)$ shows that scheduler S_1 is aware of execution of interaction $Fill_{12}$ but it is not aware of the occurrence of internal action β_3 because component $Tank_3$ is not in the scope of scheduler S_1 and consequently the state of component $Tank_3$ in the local trace of scheduler S_1 is denoted ? (except for the initial state). Moreover, scheduler S_1 is aware of the occurrences of internal actions β_2 and β_1 but it is not aware of action $Drain_{23}$ because scheduler S_1 does not manage action $Drain_{23}$.

4 From Local Traces to Global Traces

We define a new component as a *passive* observer which runs in parallel with the system and collects local traces of schedulers and reconstruct the set of possible global traces compatible with the local traces (Figure 3). The observer is always ready to receive information from schedulers. We use the term *passive* for the observer since it does not force schedulers to send data and thus does not modify the execution of the monitored system. We shall prove that such observer does not violate the semantics nor the behavior of the distributed system, that is, the observed system is observationally equivalent (see Sec. 2) to the initial system (*cf.* Property 1).

For monitoring purposes, the observer should be able to order the execution of interactions from the received local traces appropriately. In our abstract model, since schedulers do not interact directly together by sending/receiving messages, the execution of an interaction by one scheduler seems to be concurrent with the execution of all interactions by other schedulers. Nevertheless, if scheduler S_j manages interaction a and scheduler S_k manages interaction b such that a shared component $B_i \in \mathbf{B}_s$ is involved in a and b, i.e., $B_i \in \text{involved}(a) \cap \text{involved}(b)$, as a matter of fact, the execution of interactions a and b are causally related. In other words, there exists only one possible ordering of a and b and they could not have been executed concurrently. Ignoring the actual ordering of a and b would result in retrieving inconsistent global states (i.e., states that does not belong to the original system).

We instrument the system by adding controllers to the schedulers and to the shared components. The controllers of schedulers and the controllers of shared components interact whenever the scheduler and the shared components interact to transmit vector clocks and state update. Each time a scheduler executes an interaction, the associated controller attaches a vector clock to this execution and notifies the observer. Hence, the local trace of each scheduler is augmented by vector clocks and is then sent to the observer.

In the following, we define an instrumentation of abstract distributed systems to let schedulers send their local traces to an observer.

4.1 Composing Schedulers and Shared Components with Controllers

We consider a distributed system consisting of a set of components $\mathbf{B} = \{B_1, \ldots, B_{|\mathbf{B}|}\}$ (as per Definition 1) and a set of schedulers $\mathbf{S} = \{S_1, \ldots, S_{|\mathbf{S}|}\}$ where scheduler $S_j = (Q_{S_j}, Act_{S_j}, \rightarrow_{s_j})$ manages the interactions in $Act^{\gamma}_{S_j}$ and is notified by internal actions in $Act^{\beta}_{S_j}$, for S_j (as per Definition 2). We attach to S_j a local controller $\mathcal{C}^{\mathrm{s}}_j$ in charge of computing the vector clock and sending the local trace of S_j to the observer. Moreover, for each shared component $B_i \in \mathbf{S}$, we attach a local controller $\mathcal{C}^{\mathrm{b}}_i$ to communicate with the controllers of the schedulers that have B_i in their scope.

In the following, we define the controllers (instrumentation code) and the composition $\otimes$ as instrumentation process.

4.1.1 Controllers of Schedulers

Controller $\mathcal{C}^{\mathrm{s}}_j$ is in charge of computing the correct vector clock of scheduler S_j (Definition 8). It does so through the data exchange with the controllers of shared components, i.e., the controllers in the set

$$\left\{\mathcal{C}^{\mathrm{b}}_i \mid S_j \in \mathbf{S} \wedge B_i \in \mathrm{scope}(S_j)\right\},$$

which are later defined in Definition 10.

Definition 8 (Controller of scheduler) *Controller $\mathcal{C}^{\mathrm{s}}_j$ is an LTS $(Q_{\mathcal{C}^{\mathrm{s}}_j}, \mathcal{R}_{\mathcal{C}^{\mathrm{s}}_j}, \rightarrow_{\mathcal{C}^{\mathrm{s}}_j})$ such that:*

- $Q_{\mathcal{C}^{\mathrm{s}}_j} = 2^{[1,|\mathbf{B}|]} \times VC$ *is the set of states where* $2^{[1,n]}$ *is the set of subsets of component indexes and* VC *is the set of vector clocks;*
- $\mathcal{R}_{\mathcal{C}^{\mathrm{s}}_j} = \left\{\left(\widehat{\beta_i}, \emptyset\right) \,\middle|\, B_i \in \mathrm{scope}(S_j)\right\} \bigcup \{(-, \{rcv_i\lfloor vc\rfloor\}) \mid B_i \in \mathrm{scope}(S_j) \cap \mathbf{B}_{\mathrm{s}} \wedge vc \in VC\}$
$\bigcup \left\{\left(\widehat{a\lfloor vc\rfloor}, snd\right) \,\middle|\, a \in Act^{\gamma}_{S_j} \wedge vc \in VC \wedge snd \subseteq \{snd_i\lfloor vc\rfloor \mid B_i \in \mathrm{scope}(S_j) \cap \mathbf{B}_{\mathrm{s}} \wedge vc \in VC\}\right\}$
is the set of actions;
- $\rightarrow_{\mathcal{C}^{\mathrm{s}}_j} \subseteq Q_{\mathcal{C}^{\mathrm{s}}_j} \times \mathcal{R}_{\mathcal{C}^{\mathrm{s}}_j} \times Q_{\mathcal{C}^{\mathrm{s}}_j}$ *is the transition relation defined as:*

$$\left\{(\mathcal{I}, vc) \xrightarrow{\left(\widehat{a\lfloor vc'\rfloor}, \left\{snd_i\lfloor vc'\rfloor \,\mid\, i \in \mathrm{involved}(a) \wedge B_i \in \mathbf{B}_{\mathrm{s}}\right\}\right)}_{\mathcal{C}^{\mathrm{s}}_j} (\mathcal{I} \cup \mathrm{involved}(a), vc') \,\middle|\, a \in Act^{\gamma}_{S_j} \wedge vc' = \mathrm{inc}(vc, j)\right\}$$
$$\bigcup \left\{(\mathcal{I}, vc) \xrightarrow{(\widehat{\beta_i}, \emptyset)}_{\mathcal{C}^{\mathrm{s}}_j} (\mathcal{I} \setminus \{i\}, vc) \,\middle|\, \beta_i \in Act^{\beta}_{S_j}\right\}$$
$$\bigcup \left\{(\mathcal{I}, vc) \xrightarrow{(-, \{rcv_i\lfloor vc'\rfloor\})}_{\mathcal{C}^{\mathrm{s}}_j} (\mathcal{I}, \max(vc, vc')) \,\middle|\, \beta_i \in Act^{\beta}_{S_j} \wedge B_i \in \mathbf{B}_{\mathrm{s}}\right\}$$

where $\mathrm{inc}(vc, j)$ *increments the* j^{th} *element of vector clock* vc.

When the controller $\mathcal{C}^{\mathrm{s}}_j$ is in state $(\mathcal{I}, vc)$, it means that (i) $\mathcal{I}$ is the set of busy components in the scope of scheduler S_j, (ii) the execution of their latest action has been managed by scheduler S_j, and (iii) vc is the current value of the vector clock of scheduler S_j.

An action in $\mathcal{R}_{\mathcal{C}^{\mathrm{s}}_j}$ is a pair (x, y) where x is associated to the actions which send information from the controller to the observer and y is associated to the actions in which the controller sends/receives information to/from the controllers of shared components, such that
$x \in \left\{\widehat{a\lfloor vc\rfloor} \,\middle|\, a \in Act^{\gamma}_{S_j} \wedge vc \in VC\right\} \cup \left\{\widehat{\beta_i} \,\middle|\, i \in \mathrm{scope}(j)\right\} \cup \{-\}$, and
$y \subseteq \{snd_i\lfloor vc\rfloor, rcv_i\lfloor vc\rfloor \mid B_i \in \mathrm{scope}(S_j) \cap \mathbf{B}_{\mathrm{s}} \wedge vc \in VC\}$ can be intuitively understood as follows,

- action $\widehat{a\lfloor vc\rfloor}$ consists in notifying the observer about the execution of interaction a with vector clock vc attached.
- action $\widehat{\beta_i}$ consists in notifying the observer about the internal action of component B_i. The last state of component B_i is also transmitted to the observer.
- action $-$ is used in the case when the controller does not interact with the observer,

$$\text{CONT-SCH1}\ \frac{a \in \mathit{Int} \qquad q_s \xrightarrow{a}_{S_j} q_s' \qquad q_c \xrightarrow{\left(\widehat{a\lfloor vc'\rfloor},\{snd_i\lfloor vc'\rfloor \mid i \in \text{involved}(a) \wedge B_i \in \mathbf{B}_\text{s}\}\right)}_{C_j^\text{s}} q_c'}{(q_s,q_c) \xrightarrow{\left(a,\left(\widehat{a\lfloor vc'\rfloor},\{snd_i\lfloor vc'\rfloor \mid i \in \text{involved}(a) \wedge B_i \in \mathbf{B}_\text{s}\}\right)\right)}_{sc_j} (q_s',q_c')}$$

$$\text{CONT-SCH2}\ \frac{i \in \mathcal{I} \qquad q_s \xrightarrow{\beta_i}_{S_j} q_s' \qquad q_c \xrightarrow{(\widehat{\beta_i},\emptyset)}_{C_j^\text{s}} q_c'}{(q_s,q_c) \xrightarrow{(\beta_i,(\widehat{\beta_i},\emptyset))}_{sc_j} (q_s',q_c')}$$

$$\text{CONT-SCH3}\ \frac{i \in \mathcal{I} \qquad q_s \xrightarrow{\beta_i}_{S_j} q_s' \qquad q_c \xrightarrow{(\widehat{\beta_i},\emptyset)}_{C_j^\text{s}} q_c'}{(q_s,q_c) \xrightarrow{(\beta_i,(\widehat{\beta_i},\emptyset))}_{sc_j} (q_s',q_c')}$$

Figure 4: Semantics rules defining the composition controller / scheduler

- action $snd_i\lfloor vc\rfloor$ consists in sending the value of the vector clock vc of the scheduler to the shared component B_i,
- action $rcv_i\lfloor vc\rfloor$ consists in receiving the value of the vector clock vc stored in the shared component B_i.

The set of transitions is obtained as the union of three sets which can be intuitively understood as follows:

- For each interaction $a \in Act_{S_j}^{\gamma}$ managed by scheduler S_j, we include a transition with action

$$\left(\widehat{a\lfloor vc'\rfloor}, \{snd_i\lfloor vc'\rfloor \mid B_i \in \text{involved}(a) \cap \mathbf{B}_\text{s}\}\right),$$

  where $\widehat{a\lfloor vc'\rfloor}$ is a notification to the observer about the execution of interaction a along with the value of vector clock vc', and actions in set $\{snd_i\lfloor vc'\rfloor \mid B_i \in \text{involved}(a) \cap \mathbf{B}_\text{s}\}$ send the value of the vector clock vc' to the shared components involved in interaction a. Moreover, the set of indexes of the components involved in interaction a (i.e., in $\text{involved}(a)$) is added to the set of busy components; and the current value of the vector clock is incremented.

- For each action associated to the notification of the internal action of component B_i (that is, β_i), we include a transition labeled with action $(\widehat{\beta_i}, \emptyset)$ in the controller to send the updated state to the observer. Moreover, this transition removes index i from the set of busy components.

- For each action associated to the notification of internal action of a shared components $B_i \in \mathbf{B}_\text{s}$, we include a transition labeled with action $(-, \{rcv_i\lfloor vc'\rfloor\})$ in the controller to receive the value of the vector clock vc' stored in the shared component to update the vector clock of the scheduler by comparing the vector clock stored in the scheduler and the received vector clock from the shared component.

Note that, to each shared component $B_i \in \mathbf{B}_\text{s}$, we also attach a local controller in order to exchange the vector clock among schedulers in the set $\{S_j \in \mathbf{S} \mid B_i \in \text{scope}(S_j)\}$; see Definition 10.

Below, we define how a scheduler is composed with its controller. Intuitively, the controller of a scheduler ensures sending/receiving information among the scheduler, associated shared components and the observer.

Definition 9 (Semantics of $S_j \otimes_\text{s} \mathcal{C}_j^\text{s}$) *The composition of scheduler S_j and controller $\mathcal{C}_j^\text{s}$, denoted by $S_j \otimes_\text{s} \mathcal{C}_j^\text{s}$, is the LTS $(Q_{S_j} \times Q_{\mathcal{C}_j^\text{s}}, Act_{S_j} \times \mathcal{R}_{\mathcal{C}_j^\text{s}}, \rightarrow_{sc_j})$ where the transition relation $\rightarrow_{sc_j} \subseteq (Q_{S_j} \times Q_{\mathcal{C}_j^\text{s}}) \times (Act_{S_j} \times \mathcal{R}_{\mathcal{C}_j^\text{s}}) \times (Q_{S_j} \times Q_{\mathcal{C}_j^\text{s}})$ is defined by the semantics rules in Figure 4.*

The semantics rules in Figure 4 can be intuitively be understood as follows:

- *Rule* CONT-SCH1. When the scheduler executes an interaction $a \in \mathit{Int}$, the controller (i) updates the vector clock by increasing its local clock, (ii) updates the set of busy components, (iii) notifies the observer of the execution of a along with the associated vector clock vc', and (iv) sends vector clock vc' to the shared components involved in a.

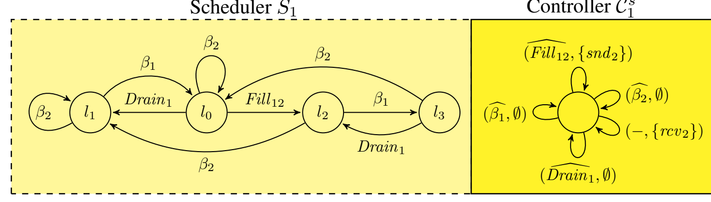


Figure 5: Controller attached to the scheduler

- *Rule* CONT-SCH2. When the scheduler is notified of an internal action of component B_i where $i \in \mathcal{I}$ (that is, the scheduler has managed the latest action of component B_i) through action β_i, the controller transfers the updated state of component B_i to the observer through action $\widehat{\beta_i}$.
- *Rule* CONT-SCH3. When the scheduler is notified of an internal action of the shared component B_i where $i \notin \mathcal{I}$ (that is, the scheduler has not managed the latest action of component B_i), the controller receives the vector clock stored in component B_i and updates the vector clock.

Example 5 (Controller of scheduler) *Figure 5 depicts the controller of scheduler S_1 (depicted in Figure 2). Actions $(\widehat{\beta_1}, \emptyset)$ and $(\widehat{\beta_2}, \emptyset)$ consist in sending the updated state to the observer. Actions $(\widehat{Drain_1}, \emptyset)$ and $(\widehat{Fill_{12}}, \{snd_2\})$ consist in notifying the observer about the occurrence of interactions managed by the scheduler. Moreover, snd_2 sends the vector clock to the shared component $Tank_2$. The controller receives the vector clock stored in the shared component $Tank_2$ through action $(-, \{rcv_2\})$ and updates its vector clock. For the sake of simplicity, variables attached to the transition labels are not shown.*

4.1.2 Controllers of Shared Components

Below, we define the controllers attached to shared components. Intuitively, the controller of a shared component ensures data exchange among the shared component and the corresponding schedulers. A scheduler sets it's current clock in a shared component's controller which can be used later by another scheduler.

Definition 10 (Controller of shared component) *Local controller $\mathcal{C}_i^{\mathrm{b}}$ for a shared component $B_i \in \mathbf{B}_{\mathrm{s}}$ with the behavior $(Q_i, Act_i \cup \{\beta_i\}, \rightarrow_i)$ is the LTS $(Q_{\mathcal{C}_i^{\mathrm{b}}}, \mathcal{R}_{\mathcal{C}_i^{\mathrm{b}}}, \rightarrow_{\mathcal{C}_i^{\mathrm{b}}})$, where*

- *$Q_{\mathcal{C}_i^{\mathrm{b}}} = VC$ is the set of states,*
- *$\mathcal{R}_{\mathcal{C}_i^{\mathrm{b}}} \subseteq \{snd_j\lfloor vc \rfloor, rcv_j\lfloor vc \rfloor \mid S_j \in \mathbf{S} \wedge B_i \in \mathrm{scope}(S_j) \wedge vc \in VC\}$ is the set of actions,*
- *$\rightarrow_{\mathcal{C}_i^{\mathrm{b}}} \subseteq Q_{\mathcal{C}_i^{\mathrm{b}}} \times \mathcal{R}_{\mathcal{C}_i^{\mathrm{b}}} \times Q_{\mathcal{C}_i^{\mathrm{b}}}$ is the transition relation defined as*

$$\left\{ vc \xrightarrow{\{rcv_j\lfloor vc' \rfloor\}}_{\mathcal{C}_i^{\mathrm{b}}} \max(vc, vc') \mid a \in Int \wedge a \cap Act_i \neq \emptyset \wedge \mathrm{managed}(a) = S_j \right\}$$
$$\bigcup \left\{ vc \xrightarrow{\{snd_j\lfloor vc \rfloor\}}_{\mathcal{C}_i^{\mathrm{b}}} vc \;\middle|\; S_j \in \mathbf{S} \wedge B_i \in \mathrm{scope}(S_j) \right\}.$$

The state of the controller $\mathcal{C}_i^{\mathrm{b}}$ is represented by its vector clock. Controller $\mathcal{C}_i^{\mathrm{b}}$ has two types of actions:

- action $rcv_j\lfloor vc' \rfloor$ consists in receiving the vector clock vc' of scheduler S_j,
- action $snd_j\lfloor vc \rfloor$ consists in sending the vector clock vc stored in the controller $\mathcal{C}_i^{\mathrm{b}}$ to scheduler S_j.

The two types of transitions can be understood as follow:

- For each action of component B_i, which is managed by scheduler S_j, we include a transition executing action $rcv_j\lfloor vc' \rfloor$ to receive the vector clock vc' of scheduler S_j and to update the vector clock stored in controller $\mathcal{C}_i^{\mathrm{b}}$.

$$\textsc{Cont-Sha1}\ \frac{a \in Int \qquad a \cap Act_i = \{a'\} \qquad \text{managed}(a) = S_j \qquad q_b \xrightarrow{a'}_i q_b' \qquad q_c \xrightarrow{\{rcv_j \lfloor vc' \rfloor\}}_{\mathcal{C}_i^{\mathrm{b}}} q_c'}{(q_b, q_c) \xrightarrow{(a', \{rcv_j \lfloor vc' \rfloor\})}_{bc_i} (q_b', q_c')}$$

$$\textsc{Cont-Sha2}\ \frac{q_b \xrightarrow{\beta_i}_i q_b' \qquad J = \{j \in [1, m] \mid B_i \in \text{scope}(S_j)\} \qquad q_c \xrightarrow{\{snd_j \lfloor vc \rfloor \mid j \in J\}}_{\mathcal{C}_i^{\mathrm{b}}} q_c'}{(q_b, q_c) \xrightarrow{(\beta_i, \{snd_j \lfloor vc \rfloor \mid j \in J\})}_{bc_i} (q_b', q_c')}$$

Figure 6: Semantics rules defining the composition controller / shared component

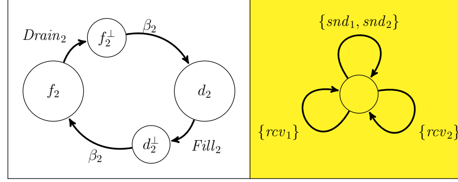


Figure 7: Controller of shared component $Tank_2$

- We include a transition with a set of actions for all the schedulers that have component B_i in their scope, that is $\{S_j \in \mathbf{S} \mid B_i \in \text{scope}(S_j)\}$, to send the stored vector clock of controller $\mathcal{C}_i^{\mathrm{b}}$ to the controllers of the corresponding schedulers, that is $\left\{\mathcal{C}_j^{\mathrm{s}} \mid S_j \in \mathbf{S} \wedge B_i \in \text{scope}(S_j)\right\}$.

Definition 11 (Semantics of $B_i \otimes_{\mathrm{b}} \mathcal{C}_i^{\mathrm{b}}$) *The composition of shared component B_i and controller $\mathcal{C}_i^{\mathrm{b}}$, denoted by $B_i \otimes_{\mathrm{b}} \mathcal{C}_i^{\mathrm{b}}$, is the LTS $(Q_i \times Q_{c_i^{\mathrm{b}}}, (Act_i \cup \{\beta_i\}) \times \mathcal{R}_{c_i^{\mathrm{b}}}, \rightarrow_{bc_i})$ where the transition relation $\rightarrow_{bc_i} \subseteq (Q_i \times Q_{c_i^{\mathrm{b}}}) \times \Big((Act_i \cup \{\beta_i\}) \times \mathcal{R}_{c_i^{\mathrm{b}}}\Big) \times (Q_i \times Q_{c_i^{\mathrm{b}}})$ is defined by the semantics rules in Figure 6.*

The semantics rules in Figure 6 can be intuitively understood as follows:

- *Rule* $\textsc{Cont-Sha1}$. applies when the scheduler notifies the shared component to execute an action part of an interaction. Controller $\mathcal{C}_i^{\mathrm{b}}$ receives the value of the vector clock of scheduler S_j from the associated controller $\mathcal{C}_j^{\mathrm{s}}$ in order to update the value of the vector clock stored in controller $\mathcal{C}_i^{\mathrm{b}}$.

- *Rule* $\textsc{Cont-Sha2}$. applies when the shared component B_i finishes its computation by executing β_i, and controller $\mathcal{C}_i^{\mathrm{b}}$ notifies the controllers of the schedulers that have component B_i in their scope, through actions snd_j, for $j \in J$, which sends the vector clock stored in controller $\mathcal{C}_i^{\mathrm{b}}$ to controllers $\mathcal{C}_j^{\mathrm{s}}$ with $j \in J$, where J is the set of indexes of schedulers which have the shared component B_i in their scope.

Example 6 (Controller of shared component) *Figure 7 depicts the controller of the shared component $Tank_2$ (depicted in Figure 2). Action rcv_1 (resp. rcv_2) consists in the reception and storage of the vector clock from scheduler S_1 (resp. S_2) upon the execution of interaction $Fill_{12}$ (resp. $Drain_{23}$). Action $\{snd_1, snd_2\}$ sends the stored vector clock to the schedulers S_1 and S_2 when the component $Tank_2$ performs its internal action β_2.*

Definition 12 (Instrumented system) *Given a distributed system with global behavior $(Q, GAct, \rightarrow)$ as per Definition 4, the global behavior of the augmented distributed system with a set of controllers (as per Definitions 8 and 10) consisting $(S_1 \otimes_{\mathrm{s}} \mathcal{C}_1^{\mathrm{s}}, \ldots, S_{|\mathbf{S}|} \otimes_{\mathrm{s}} \mathcal{C}_{|\mathbf{S}|}^{\mathrm{s}}, B_1', \ldots, B_{|\mathbf{B}|}')$ where $B_i' = B_i \otimes_{\mathrm{b}} \mathcal{C}_i^{\mathrm{b}}$ if $B_i \in \mathbf{B}_{\mathrm{s}}$ and $B_i' = B_i$ otherwise, is the LTS $(Q_c, GAct_c, \rightarrow_c)$ where:*

- *$Q_c \subseteq \bigotimes_{i=1}^{|\mathbf{B}|} Q_i' \times \bigotimes_{j=1}^{|\mathbf{S}|} (Q_{S_j} \times Q_{\mathcal{C}_j^{\mathrm{s}}})$ where $Q_i' = Q_i \times Q_{c_i^{\mathrm{b}}}$ if $B_i \in \mathbf{B}_{\mathrm{s}}$ and $Q_i' = Q_i$ otherwise, is the set of states consisting of the states of schedulers and components with their controllers,*

- *$GAct_c = GAct \times \{\mathcal{R}_{\mathcal{C}_j^{\mathrm{s}}}, \mathcal{R}_{\mathcal{C}_i^{\mathrm{b}}} \mid S_j \in \mathbf{S} \wedge B_i \in \mathbf{B}_{\mathrm{s}}\}$ is the set of actions,*

- *$\rightarrow_c \subseteq Q_c \times GAct_c \times Q_c$ is the transition relation.*

4.2 Correctness of Instrumentation

The next proposition states that the LTS of the instrumented distributed system (see Definition 12) is weakly bisimilar with the LTS of initial distributed system, thus the composition of a set of controllers with schedulers and shared components defined in Sec. 4.1 does not affect the semantics of the initial distributed system.

***Proposition* 1** $(Q, GAct, \rightarrow) \sim (Q_c, GAct_c, \rightarrow_c)$.

Proof: The proof of this proposition is in Appendix **??** (p. **??**).

4.3 Event Extraction from the Local Traces of the Instrumented System

According to Definitions 8 and 10, the first action in the semantics rules of a controlled scheduler or shared component corresponds to an interaction of the initial system. Thus, the notion of trace is extended in the natural way by considering the additional semantics rules. Elements of a trace are updated by including the new configurations and actions of controlled schedulers and shared components.

Example 7 (Local traces of instrumented system) *Consider Example 4, the local traces of the instrumented system for two global traces t_1 and t_2 are:*

- $s_1(t_1) = (d_1, d_2, d_3) \cdot \Big(Fill_{12}, \Big(\widehat{Fill_{12}\lfloor(1,0)\rfloor}, snd_2\lfloor(1,0)\rfloor\Big)\Big) \cdot (\bot, \bot, ?) \cdot \Big(\{\beta_1\}, \Big(\widehat{\{\beta_1\}}, \emptyset\Big)\Big) \cdot (f_1, \bot, ?) \cdot \Big(\{Drain_1\}, \widehat{\{Drain_1\}\lfloor(2,0)\rfloor}\Big) \cdot (\bot, \bot, ?) \cdot \Big(\{\beta_2\}, \Big(\widehat{\{\beta_2\}}, \emptyset\Big)\Big) \cdot (\bot, f_2, ?)$,
- $s_2(t_1) = (d_1, d_2, d_3) \cdot \Big(\{Fill_3\}, \widehat{\{Fill_3\}\lfloor(0,1)\rfloor}\Big) \cdot (?, d_2, \bot) \cdot (\{\beta_2\}, (-, rcv_1\lfloor(1,0)\rfloor)) \cdot (?, f_2, \bot)$,
- $s_1(t_2) = (d_1, d_2, d_3) \cdot \Big(Fill_{12}, \Big(\widehat{Fill_{12}\lfloor(1,0)\rfloor}, snd_2\lfloor(1,0)\rfloor\Big)\Big) \cdot (\bot, \bot, ?) \cdot \Big(\{\beta_2\}, \Big(\widehat{\{\beta_2\}}, \emptyset\Big)\Big) \cdot (\bot, f_2, ?) \cdot \Big(\{\beta_1\}, \Big(\widehat{\{\beta_1\}}, \emptyset\Big)\Big) \cdot (f_1, f_2, ?)$,
- $s_2(t_2) = (d_1, d_2, d_3) \cdot \Big(\{Fill_3\}, \widehat{\{Fill_3\}\lfloor(0,1)\rfloor}\Big) \cdot (?, d_2, \bot) \cdot \Big(\{\beta_3\}, \Big(\widehat{\{\beta_3\}}, \emptyset\Big)\Big) \cdot (?, d_2, f_3) \cdot (\{\beta_2\}, (-, rcv_1\lfloor(1,0)\rfloor)) \cdot (?, f_2, f_3) \cdot \Big(Drain_{23}, \Big(\widehat{Drain_{23}\lfloor(1,0)\rfloor}, snd_2\lfloor(1,2)\rfloor\Big)\Big) \cdot (?, \bot, \bot)$.

In both traces, scheduler S_2 is notified of the state update of component $Tank_2$ (that is β_2), but scheduler S_2 does not sent it to the observer. Indeed, following the semantics rules of composition of a scheduler and its controller (Definition 9), a scheduler only sends the received state from a component only if the execution of the latest action on this component has been managed by this scheduler.

Definition 13 (Sequence of events) *Let t be the global trace of the distributed system and $s_j(t) = q_0 \cdot \gamma_1 \cdot q_1 \cdots \gamma_{k-1} \cdot q_{k-1} \cdot \gamma_k \cdot q_k$, for $j \in [1, m]$, be the local trace of scheduler S_j (as per Definition 7). The sequence of events of $s_j(t)$ is inductively defined as follows:*

- $\text{event}(q_0) = \epsilon$,
- $$\text{event}\,(s_j(t) \cdot \gamma \cdot q) = \begin{cases} \text{event}(s_j(t)) \cdot (a, vc) & \textit{if } \gamma \textit{ is of the form } (*, (\widehat{a\lfloor vc\rfloor}, *))\,, \\ \text{event}(s_j(t)) \cdot \beta_i & \textit{if } \gamma \textit{ is of the form } (*, (\widehat{\beta_i}, *))\,, \\ \text{event}(s_j(t)) & \textit{otherwise}. \end{cases}$$

Intuitively, any communication between the controller of scheduler S_j and the observer is defined as an event of scheduler S_j. According to the semantic rules of composition $S_j \otimes \mathcal{C}_j^{\mathrm{s}}$ (see Definition 9), controller $\mathcal{C}_j^{\mathrm{s}}$ sends information to the observer (actions denoted by ^ over them) when scheduler S_j (i) executes an interaction $a \in Act$, or (ii) is notified by the internal action of a component which the execution of its latest action has been managed by scheduler S_j.

Example 8 (Sequence of events) *The sequences of events of local traces in Example 7 are:*

- $\text{event}(s_1(t_1)) : (Fill_{12}, (1,0)) \cdot \beta_1 \cdot (Drain_1, (2,0)) \cdot \beta_2$,
- $\text{event}(s_2(t_1)) : (Fill_3, (0,1))$,

- $\mathrm{event}(s_1(t_2)) : (\mathit{Fill}_{12}, (1,0)) \cdot \beta_2 \cdot \beta_1$,
- $\mathrm{event}(s_2(t_2)) : (\mathit{Fill}_3, (0,1)) \cdot \beta_3 \cdot (\mathit{Drain}_{23}, (1,2))$.

To interpret the state updates received by the observer, we use the notion of computation lattice, adapted to distributed CBSs with multi-party interactions in the next section.

5 Computation Lattice of a Distributed CBS with Multi-party Interactions

In the previous section, we define how to instrument the system to have controllers generating events (i.e., local traces) sent to a central observer. In this section, we define how the central observer constructs on-the-fly a computation lattice representing the possible global traces compatible with the local traces received from the controllers of schedulers.

5.1 Extended Computation Lattice

The constructed lattice is represented implicitly using vector clocks. The construction of the lattice mainly performs the two following operations: (i) *creations of new nodes* and (ii) *updates* of existing nodes in the lattice. The observer receives two sorts of events: events related to the execution of an interaction in Int, referred to as *action events*, and events related to internal actions in $\cup_{i\in[1,|\mathbf{B}|]}(\{\beta_i\} \times Q_i)$, referred to as *update events*. (Recall that internal actions carry the state of the component which has performed the action – the state is transmitted to the observer by the controller that is notified of this action. See Sec. 3). Hence, the set of action events is defined as $E_{\mathrm{a}} = \mathit{Int} \times \mathit{VC}$ and the set of update events is defined as $E_{\beta} = \cup_{i\in[1,|\mathbf{B}|]}(\{\beta_i\} \times Q_i)$. Action events lead to the creation of new nodes in the direction of the scheduler emitting the event while update events complete the information in the nodes of the lattice related to the state of the component related to the event. The set of all events is denoted by $E = E_{\beta} \cup E_{\mathrm{a}}$. Since the received events are not totally ordered (because of potential communication delay), we construct the computation lattice based on the vector clocks attached to the received events. Note, we assume that the events received from a scheduler are totally ordered.

We first extend the notion of computation lattice.

Definition 14 (Extended Computation lattice) *A computation lattice $\mathcal{L}$ is a tuple $(N, \mathit{Int}, \rightarrowtail\!\!\!\!\twoheadrightarrow)$, where*

- *$N \subseteq Q^l \times \mathit{VC}$ is the set of nodes, with $Q^l = \bigotimes_{i=1}^{|\mathbf{B}|}\left(Q_i^{\mathrm{r}} \bigcup \left\{\perp_i^j \mid S_j \in \mathbf{S} \wedge B_i \in \mathrm{scope}(S_j)\right\}\right)$ and VC is the set of vector clocks,*
- *Int is the set of multi-party interactions as defined in Sec. 3.1,*
- $\rightarrowtail\!\!\!\!\twoheadrightarrow = \left\{(\eta, a, \eta') \in N \times \mathit{Int} \times N \mid a \in \mathit{Int} \wedge \eta \rightarrowtail \eta' \wedge \eta.\mathit{state} \xrightarrow{a} \eta'.\mathit{state}\right\}$,

where $\rightarrowtail\!\!\!\!\twoheadrightarrow$ is the extended presentation of happened-before relation which is labeled by the set of multi-party interactions and $\eta.\mathit{state}$ referring to the state of node η.

Intuitively, a computation lattice consists of a set of partially connected nodes, where each node is a pair, consisting of a state of the system and a vector clock. A system state consists in the states of all components. The state of a component is either a ready state or a busy state (as per Definition 1). In this context we represent a busy state of component $B_i \in \mathbf{B}$, by $\perp_i^j$ which shows that component B_i is busy to finish its latest action which has been managed by scheduler $S_j \in \mathbf{S}$. A computation lattice $\mathcal{L}$ initially consists of an initial node $\mathit{init}_{\mathcal{L}} = (\mathit{init}, (0, \ldots, 0))$, where init is the initial state of the system and $(0, \ldots, 0)$ is a vector clock where all the clocks associated to the schedulers are zero. The set of nodes of computation lattice $\mathcal{L}$ is denoted by $\mathcal{L}.\mathit{nodes}$, and for a node $\eta = (q, vc) \in \mathcal{L}.\mathit{nodes}$, $\eta.\mathit{state}$ denotes q and $\eta.\mathit{clock}$ denotes vc. If (i) the event of node η happened before the events of node η', that is $\eta'.\mathit{clock} > \eta.\mathit{clock}$ and $\eta \rightarrowtail \eta'$, and (ii) the states of η and η' follow the global behavior of the system (as per Definition 4) in the sense that the execution of an interaction $a \in \mathit{Int}$ from the state of η brings the system to the state of η', that is $\eta.\mathit{state} \xrightarrow{a} \eta'.\mathit{state}$, then in the computation lattice it is denoted by $\eta \overset{a}{\rightarrowtail\!\!\!\!\twoheadrightarrow} \eta'$ or by $\eta \rightarrowtail\!\!\!\!\twoheadrightarrow \eta'$ when clear from context.

Two nodes η and η' of the computation lattice $\mathcal{L}$ are said to be concurrent if neither $\eta.\mathit{clock} > \eta'.\mathit{clock}$ nor $\eta'.\mathit{clock} > \eta.\mathit{clock}$. For two concurrent nodes η and η' if there exists a node η'' such that $\eta'' \rightarrowtail\!\!\!\!\twoheadrightarrow \eta$ and $\eta'' \rightarrowtail\!\!\!\!\twoheadrightarrow \eta'$, then node η'' is said to be the *meet* of η and η' denoted by $\mathrm{meet}(\eta, \eta', \mathcal{L}) = \eta''$.

The rest of this section is structured as follows. In Sec. 5.2 some intermediate notions are defined in order to introduce our algorithm to construct the computation lattice in Sec. 5.3. In Sec. 5.4 we discuss the correctness of the algorithm.

5.2 Intermediate Operations for the Construction of the Computation Lattice

In the reminder, we consider a computation lattice $\mathcal{L}$ as per Definition 14. The reception of a new event either modifies $\mathcal{L}$ or is kept in a queue to be used later. Action events extend $\mathcal{L}$ using operator extend (Definition 15), and update events update the existing nodes of $\mathcal{L}$ by adding the missing state information into them using operator update (Definition 18). By extending the lattice with new nodes, one needs to further complete the lattice by computing joints of created nodes (Definition 17) with existing ones so as to complete the set of possible global states and global traces.

Extension of the lattice. We define a function to extend a node of the lattice with an action event which takes as input a node of the lattice and an action event and outputs a new node.

Definition 15 (Node extension) *Function* $\mathrm{extend} : (Q^l \times VC) \times E_{\mathrm{a}} \rightarrow Q^l \times VC$ *is defined as follows. For a node* $\eta = ((q_1, \ldots, q_{|\mathbf{B}|}), vc) \in Q^l \times VC$ *and an action event* $e = (a, vc') \in E_{\mathrm{a}}$,

$$\mathrm{extend}(\eta, e) = \begin{cases} ((q'_1, \ldots, q'_{|\mathbf{B}|}), vc') & \text{if } \exists j \in [1, |\mathbf{S}|] : \\ & \quad (vc'[j] = vc[j] + 1 \wedge \forall j' \in [1, |\mathbf{S}|] \setminus \{j\} : vc'[j'] = vc[j']) \\ \textit{undefined} & \textit{otherwise} \; ; \end{cases}$$

with $\forall i \in [1, |\mathbf{B}|] : q'_i = \begin{cases} q_i & \textit{if } B_i \in \mathrm{involved}(a), \\ \bot_i^k & \textit{otherwise.} \end{cases}$

where $k = \mathrm{managed}(a).index$.

Node η said to be extendable by event e if $\mathrm{extend}(\eta, e)$ is defined. Intuitively, node $\eta = (q, vc)$ represents a global state of the system and extensibility of η by action event $e = (a, vc')$ means that from the global state q, scheduler $S_j = \mathrm{managed}(a)$, could execute interaction a. State $\bot_i^k$ indicates that component B_i is busy and being involved in a global action which has been executed (managed) by scheduler S_k for $k \in [1, |\mathbf{S}|]$.

We say that computation lattice $\mathcal{L}$ is extendable by action event e if there exists a node $\eta \in \mathcal{L}.nodes$ such that $\mathrm{extend}(\eta, e)$ is defined.

***Property* 1** $\forall e \in E_{\mathrm{a}} : |\{\eta \in \mathcal{L}.nodes \mid \exists \eta' \in Q^l \times VC : \eta' = \mathrm{extend}(\eta, e)\}| \leq 1$.

Property 1 states that for any update event e, there exists at most one node in the lattice for which function extend is defined (meaning that $\mathcal{L}$ can be extended by event e from that node).

Example 9 (Node extension) *Considering the local traces described in Example 8, initially, the computation lattice consists of the initial node which has the initial state init, with an associated vector clock* $(0, 0)$*, i.e.,* $init_{\mathcal{L}} = ((d_1, d_2, d_3), (0, 0))$. *Consider the sequence of events in trace* t_1 *from Example 8, node* $((d_1, d_2, d_3), (0, 0))$ *is extendable by event* $(Fill_{12}, (1, 0))$ *because, according to Definition 15, we have:*

$$\mathrm{extend}(((d_1, d_2, d_3), (0, 0)), (Fill_{12}, (1, 0))) = ((\bot_1^1, \bot_2^1, d_3), (1, 0)).$$

Furthermore, to illustrate Property 1, let us consider the extended lattice after event $(Fill_{12}, (1, 0))$ *which consists of two nodes,* $init_{\mathcal{L}}$ *and* $((\bot_1^1, \bot_2^1, d_3), (1, 0))$. *When action event* $(Fill_3, (0, 1))$ *is received,* $\mathrm{extend}(init_{\mathcal{L}}, (Fill_3, (0, 1)) = ((d_1, d_2, \bot_3^2), (0, 1))$ *whereas* $\mathrm{extend}(((\bot_1^1, \bot_2^1, d_3), (1, 0)), (Fill_3, (0, 1)))$ *is not defined which shows that Property 1 holds on the lattice.*

We define a relation between two vector clocks to distinguish the concurrent execution of two interactions such that both could happen from a specific global state of the system.

Definition 16 (Relation $\mathcal{J}_{\mathcal{L}}$) *Relation* $\mathcal{J}_{\mathcal{L}} \subseteq VC \times VC$ *is defined between two vector clocks as follows:* $\mathcal{J}_{\mathcal{L}} = \{(vc, vc') \in VC \times VC \mid \exists!\, k \in [1, |\mathbf{S}|] : vc[k] = vc'[k] + 1 \wedge \exists!\, l \in [1, |\mathbf{S}|] : vc'[l] = vc[l] + 1 \wedge \forall j \in [1, |\mathbf{S}|] \setminus \{k, l\} : vc[j] = vc'[j]\}$.

For two vector clocks vc and vc' to be in relation $\mathcal{J}_{\mathcal{L}}$, vc and vc' should agree on all but two clocks values related to two schedulers of indexes k and l. On one of these indexes, the value of one vector clock is equal to the value of the other vector clock plus 1, and the converse on the other index. Intuitively, $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$ means that nodes η and η' are associated to two concurrent events (caused by the execution of two interactions managed by two different schedulers) that both could happen from a unique global state of the system which is the meet of η and η' (see Property 2). Example 10 illustrates relation $\mathcal{J}_{\mathcal{L}}$.

Property 2 $\forall \eta, \eta' \in \mathcal{L}.nodes : (\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}} \implies \text{meet}(\eta, \eta', \mathcal{L}) \in \mathcal{L}.nodes.$

Property 2 states that for two nodes η and η' in lattice $\mathcal{L}$ such that $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$, there exists a node in lattice $\mathcal{L}$ as the meet of η and η', that is $\text{meet}(\eta, \eta', \mathcal{L}) \in \mathcal{L}.nodes$.

The joint node of η and η' is defined as follows.

Definition 17 (Joint node) *For two nodes $\eta, \eta' \in \mathcal{L}.nodes$ such that $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$, the joint node of η and η', denoted by* $\text{joint}(\eta, \eta', \mathcal{L}) = \eta''$, *is defined as follows:*

- $\forall i \in [1, |\mathbf{B}|] : \eta''.state[i] = \begin{cases} \eta.state[i] & \text{if } \eta.state[i] \neq \eta_m.state[i], \\ \eta'.state[i] & \text{otherwise;} \end{cases}$
- $\eta''.clock = \max(\eta.clock, \eta'.clock)$;

where $\eta_m = \text{meet}(\eta, \eta', \mathcal{L})$.

According to Property 2, for two nodes η and η' in relation $\mathcal{J}_{\mathcal{L}}$, their meet node exists in the lattice. The state of the joint node of η and η' is defined by comparing their states and the state of their meet. Since two nodes in relation $\mathcal{J}_{\mathcal{L}}$ are concurrent, the state of component B_i for $i \in [1, |\mathbf{B}|]$ in nodes η and η' is either equal to the state of component B_i in their meet, or only one of the nodes η and η' has a different state than their meet (components can not be both involved in two concurrent executions). The joint node of two nodes η and η' takes into account the latest changes of the state of the nodes η and η' compared to their meet. Note that $\text{joint}(\eta, \eta', \mathcal{L}) = \text{joint}(\eta', \eta, \mathcal{L})$, because joint is defined for nodes whose clocks are in relation $\mathcal{J}_{\mathcal{L}}$.

Example 10 (Relation $\mathcal{J}_{\mathcal{L}}$ and joint node) *To continue Examples 9 and 11, the reception of action event* $(Fill_3, (0, 1))$ *extends the lattice in the direction of scheduler S_2 because function* extend *is defined, that is:*

$$\text{extend}(((d_1, d_2, d_3), (0, 0)), (Fill_3, (0, 1))) = ((d_1, d_2, \bot_3^2), (0, 1)).$$

After this extension, the nodes of the lattice are $((d_1, d_2, d_3), (0, 0))$, $((\bot_1^1, \bot_2^1, d_3), (1, 0))$ *and* $((d_1, d_2, \bot_3^2), (0, 1))$. *According to Definition 16, the vector clocks of the nodes* $((\bot_1^1, \bot_2^1, d_3), (1, 0))$ *and* $((d_1, d_2, \bot_3^2), (0, 1))$ *are in relation $\mathcal{J}_{\mathcal{L}}$ (i.e.,* $((1, 0), (0, 1)) \in \mathcal{J}_{\mathcal{L}}$*). Hence, following Definition 17, the joint node of the two above nodes is* $((\bot_1^1, \bot_2^1, \bot_3^2), (1, 1))$, *and their meet is* $((d_1, d_2, d_3), (0, 0))$.

Update of the lattice. We define a function to update a node of the lattice which takes as input a node of the lattice and an update event and outputs the updated version of the input node.

Definition 18 (Node update) *Function* $\text{update} : (Q^l \times VC) \times E_\beta \to Q^l \times VC$ *is defined as follows. For a node* $\eta = ((q_1, \ldots, q_{|\mathbf{B}|}), vc)$ *and an update event* $e = (\beta_i, q_i') \in E_\beta$ *with* $i \in [1, |\mathbf{B}|]$ *which is sent by scheduler* S_k *with* $k \in [1, |\mathbf{S}|]$*:*

$$\text{update}(\eta, e) = ((q_1, \ldots, q_{i-1}, q_i'', q_{i+1}, \ldots, q_{|\mathbf{B}|}), vc),$$

with $q_i'' = \begin{cases} q_i' & \text{if } q_i = \bot_i^k, \\ q_i & \text{otherwise.} \end{cases}$

An update event (β_i, q_i') contains an updated state of some component B_i. By updating a node η in the lattice with an update event which is sent from scheduler S_k, we update the incomplete global state associated to η by adding the state information of that component, if the state of component B_i associated to node η is $\bot_i^k$. Intuitively means that a busy state which is caused by an execution of an action managed by scheduler S_k can only be replace by a ready state sent by the same scheduler S_k. Updating node η does not modify the associated vector clock vc.

Example 11 (Node update) *To continue Example 9, let us consider node* $((\bot_1^1, \bot_2^1, d_3), (1, 0))$ *whose associated global state is incomplete (because of the lack of the state information of* $Tank_1$ *and* $Tank_2$*), and update event* (β_1, f_1) *sent by scheduler S_1. To obtain the updated node, we apply function* update *over the node and the update event. We have:* $\text{update}(((\bot_1^1, \bot_2^1, d_3), (1, 0)), (\beta_1, f_1)) = ((f_1, \bot_2^1, d_3), (1, 0))$. *Concerning the initial node of the lattice and update event* (β_1, f_1), $\text{update}(((d_1, d_2, d_3), (0, 0)), (\beta_1, f_1)) = ((d_1, d_2, d_3), (0, 0))$.

Algorithm 1 MAKE

Global variables: $\mathcal{L}$ initialized to $init_{\mathcal{L}}$,
κ initialized to ϵ,
V initialized to $(0, \ldots, 0)$.

```
1: procedure MAKE(e, from-the-queue)
2:     if e ∈ E_a then                                   ▷ if e is an action event.
3:         ACTIONEVENT(e, from-the-queue)
4:     else if e ∈ E_β then                              ▷ if e is an update event.
5:         UPDATEEVENT(e, from-the-queue)
6:     end if
7: end procedure
```

Buffering events. The reception of an action event or an update event might not always lead to extending or updating the current computation lattice. Due to communication delay, an event which has happened before another event might be received later by the observer. It is necessary for the construction of the computation lattice to use events in a specific order. Such events must be kept in a waiting queue to be used later. For example, such a situation occurs when receiving action event e such that function extend is not defined over e and none of the existing nodes of the lattice. In this case event e must be kept in the queue until obtaining another configuration of the lattice in which function extend is defined. Moreover, an update event e' referring to an internal action of component B_i is kept in the queue if there exists an action event e'' in the queue such that component B_i is involved in e'', because we can not update the nodes of the lattice with an update event associated to an execution which is not yet taken into account in the lattice.

Definition 19 (Queue κ) *A queue of events is a finite sequence of events in E. Moreover, for a non-empty queue $\kappa = e_1 \cdot e_2 \cdots e_r$, $\mathrm{remove}(\kappa, e) = \kappa(1 \cdots z-1) \cdot \kappa(z+1 \cdots r)$ with $e = e_z \in \{e_1, e_2, \ldots, e_r\}$.*

Queue κ is initialized to an empty sequence. Function remove takes as input queue κ and an event in the queue and outputs the version of κ in which the given event is removed from the queue.

Example 12 (Event storage in the queue) *Consider trace t_2 in Example 8 such that all the events of scheduler S_2 are received by the observer earlier than the events of scheduler S_1. After the reception of action event $(Fill_3, (0,1))$, since $\mathrm{extend}(((d_1, d_2, d_3), (0,0)), (Fill_3, (0,1)))$ is defined, the lattice is extended in the direction of scheduler S_2 and the new node $((d_1, d_2, \bot_3^2)(0,1))$ is created. The reception of update event (β_3, f_3) updates the newly created node $((d_1, d_2, \bot_3^2)(0,1))$ to $((d_1, d_2, f_3)(0,1))$. After the reception of action event $(Drain_{23}, (1,2))$, since there is no node in the lattice where function extend is defined over, event $(Drain_{23}, (1,2))$ must be stored in the queue, therefore $\kappa = (Drain_{23}, (1,2))$.*

5.3 Algorithm Constructing the Computation Lattice

In the following, we define an algorithm based on the above definitions to construct the computation lattice based on the events received by the global observer.

The algorithm consists of a main procedure (see Algorithm 1) and several sub-procedures using global variables lattice $\mathcal{L}$ (Definition 14) and queue κ (Definition 19).

For an action event $e \in E_a$ with $e = (a, vc)$, $e.action$ denotes interaction a and $e.clock$ denotes vector clock vc. For an update event $e \in E_\beta$ with $e = (\beta_i, q_i)$, $e.index$ denotes index i.

Initially, after the reception of event e from a controller of a scheduler, the observer calls the main procedure MAKE$(e, \texttt{false})$. In the following, we describe each procedure in detail.

MAKE (Algorithm 1): Procedure MAKE takes two parameters as input: an event e and a boolean variable $from\text{-}the\text{-}queue$. Parameters e and $from\text{-}the\text{-}queue$ vary based on the type of event e. Boolean variable $from\text{-}the\text{-}queue$ is true when the input event e is picked up from the queue and false otherwise (i.e., event e is received from a controller of a scheduler). Procedure MAKE uses two sub-procedures, ACTIONEVENT and UPDATEEVENT. If the input event is an action event, sub-procedure ACTIONEVENT is called, and if the input event is an update event, sub-procedure UPDATEEVENT is called. Procedure MAKE updates the global variables.

Algorithm 2 ACTIONEVENT

1: **procedure** ACTIONEVENT(e, *from-the-queue*)
2: $\quad$ *lattice-extend* ← `false`
3: $\quad$ **for** all $\eta \in \mathcal{L}.nodes$ **do**
4: $\quad\quad$ **if** $\exists \eta' \in Q^l \times VC : \eta' = \text{extend}(\eta, e)$ **then**
5: $\quad\quad\quad$ $\mathcal{L}.nodes \leftarrow \mathcal{L}.nodes \cup \{\eta'\}$ ▷ extend the lattice with the new node.
6: $\quad\quad\quad$ MODIFYQUEUE(e, *from-the-queue*, `true`) ▷ event e is removed from the queue if it was picked up from the queue.
7: $\quad\quad\quad$ *lattice-extend* ← `true`
8: $\quad\quad\quad$ **break** ▷ stop iteration when the lattice is extended (Property 1).
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad$ **if** ¬ *lattice-extend* **then**
12: $\quad\quad$ MODIFYQUEUE(e, *from-the-queue*, `false`) ▷ event e is added to the queue if it was not picked up from the queue.
13: $\quad\quad$ **return**
14: $\quad$ **end if**
15: $\quad$ JOINTS( ) ▷ extend the lattice with joint nodes.
16: $\quad$ REMOVEEXTRANODES() ▷ lattice size reduction.
17: $\quad$ **if** ¬ *from-the-queue* **then**
18: $\quad\quad$ CHECKQUEUE( ) ▷ recall the events stored in the queue.
19: $\quad$ **end if**
20: **end procedure**

ACTIONEVENT (Algorithm 2): Procedure ACTIONEVENT is associated to the reception of action events and takes as input an action event e and a boolean parameter *from-the-queue*, which is false when event e is received from a controller of a scheduler and true when event e is picked up from the queue. Procedure ACTIONEVENT modifies global variables $\mathcal{L}$ and κ.

Procedure ACTIONEVENT has a local boolean variable *lattice-extend* which is true when an input action event could extend the lattice (i.e., the current computation lattice is extendable by the input action event) and false otherwise.

By iterating over the existing nodes of lattice $\mathcal{L}$, ACTIONEVENT checks if there exists a node η in $\mathcal{L}.nodes$ such that function extend is defined over event e and node η (Definition 15). If such a node η is found, ACTIONEVENT creates the new node $\text{extend}(\eta, e)$, adds it to the set of the nodes of the lattice, invokes procedure MODIFYQUEUE, and stops iteration. Otherwise, ACTIONEVENT invokes procedure MODIFYQUEUE and terminates.

In the case of extending the lattice by a new node, it is necessary to create the (possible) joint nodes. To this end, in Line 15 procedure JOINTS is called to evaluate the current lattice and create the joint nodes. For optimization purposes, after making the joint nodes procedure REMOVEEXTRANODES is called to eliminate unnecessary nodes to optimize the lattice size.

After making the joint nodes and (possibly) reducing the size of the lattice, if the input action event is not picked from the queue, ACTIONEVENT invokes procedure CHECKQUEUE in Line 18, otherwise it terminates.

UPDATEEVENT (Algorithm 3): Procedure UPDATEEVENT is associated to the reception of update events. Recall that an update event e contains the state update of some component B_i with $i \in [1, n]$ ($e.index = i$). Procedure UPDATEEVENT takes as input an update event e and a boolean value associated to parameter *from-the-queue*. Procedure UPDATEEVENT modifies global variables $\mathcal{L}$ and κ.

First, UPDATEEVENT checks the events in the queue. If there exists an action event e' in the queue such that component B_i is involved in $e'.action$, UPDATEEVENT adds update event e to the queue using MODIFYQUEUE and terminates. Indeed, one can not update the nodes of the lattice with an update event associated to an execution which is not yet taken into account in the lattice.

If no action event in the queue concerned component B_i, UPDATEEVENT updates all the nodes of the lattice (Lines 8-10) according to Definition 18.

Finally, the input update event is removed from the queue if it is picked from the queue, using MODIFYQUEUE.

Algorithm 3 UPDATEEVENT

1: **procedure** UPDATEEVENT(e, *from-the-queue*)
2: **for all** $e' \in \kappa$ **do**
3: **if** $e' \in E_{\mathrm{a}} \wedge e.index \in \mathrm{involved}(e'.action)$ **then** ▷ check if there exists an action event in the queue concerning component $B_{e.index}$.
4: MODIFYQUEUE(e, *from-the-queue*, `false`) ▷ event e is added to the queue if it was not picked up from the queue.
5: **return**
6: **end if**
7: **end for**
8: **for all** $\eta \in \mathcal{L}.nodes$ **do**
9: $\eta \leftarrow \mathrm{update}(\eta, e)$ ▷ update nodes according to Definition 18.
10: **end for**
11: MODIFYQUEUE(e, *from-the-queue*, `true`)
12: **end procedure**

Algorithm 4 MODIFYQUEUE

1: **procedure** MODIFYQUEUE(e, *from-the-queue*, *event-is-used*)
2: **if** *from-the-queue* $\wedge$ *event-is-used* **then**
3: $\kappa \leftarrow \mathrm{remove}(\kappa, e)$ ▷ event e is removed from the queue if it is picked from queue and used.
4: **else if** $\neg$ *from-the-queue* $\wedge \neg$ *event-is-used* **then**
5: $\kappa \leftarrow \kappa \cdot e$ ▷ event e is added to the queue if it is not picked from queue and could not be used.
6: **end if**
7: **end procedure**

MODIFYQUEUE (Algorithm 4): Procedure MODIFYQUEUE takes as input an event e and boolean variables *from-the-queue* and *event-is-used*. Procedure MODIFYQUEUE adds (*resp.* removes) event e to (*resp.* from) queue κ according to the following conditions. If event e is picked up from the queue (i.e., *from-the-queue* = `true`) and e is used in the algorithm to extend or update the lattice (i.e., *event-is-used* = `true`), event e is removed from the queue (Line 3). Moreover, if event e is not picked up from the queue and it is not used in the algorithm, event e is stored in the queue (Line 5).

JOINTS (Algorithm 5): Procedure JOINTS extends lattice $\mathcal{L}$ in such a way that all the possible joints have been created. First, procedure JCOMPUTE is invoked to compute relation $\mathcal{J}_{\mathcal{L}}$ (Definition 16) among the existing nodes of the lattice and then creates the joint nodes and adds them to the set of the nodes of the lattice. Then, after the creation of the joint node of two nodes η and η', $(\eta.clock, \eta'.clock)$ is removed from relation $\mathcal{J}_{\mathcal{L}}$. It is necessary to compute relation $\mathcal{J}_{\mathcal{L}}$ again after the creation of joint nodes, because new nodes can be in relation $\mathcal{J}_{\mathcal{L}}$. This process terminates when $\mathcal{J}_{\mathcal{L}}$ is empty.

JCOMPUTE (Algorithm 6): Procedure JCOMPUTE computes relation $\mathcal{J}_{\mathcal{L}}$ by pairwise iteration over all the nodes of the lattice and checks if the vector clocks of any two nodes satisfy the conditions in Definition 16. The pair of vector clocks satisfying the above conditions are added to relation $\mathcal{J}_{\mathcal{L}}$.

CHECKQUEUE (Algorithm 7): Procedure CHECKQUEUE recalls the events stored in the queue $e \in \kappa$ and executes MAKE(e, `true`), to check whether the conditions for taking them into account to update the lattice hold.

Procedure CHECKQUEUE checks the events in the queue until none of the events in the queue can be used either to extend or to update the lattice. To this end, before checking queue κ, in Line 3 a copy of queue κ is stored in κ', and after iterating all the events in queue κ, the algorithm checks the equality of current queue and the copy of the queue before checking. If the current queue κ and copied queue κ' have the same events, it means that none of the events in queue κ has been used (thus removed), therefore the algorithm stops checking the queue again by breaking the loop in Line 8.

Note, when the algorithm is iterating over the events in the queue, i.e., when the value of variable *from-the-queue* is true, it is not necessary to iterate over the queue again (Algorithm 2, Line 17). Moreover, events in the queue

Algorithm 5 JOINTS

1: **procedure** JOINTS
2: $\mathcal{J}_{\mathcal{L}} \leftarrow$ JCOMPUTE ▷ compute the pairs of the vector clocks of the nodes which are in $\mathcal{J}_{\mathcal{L}}$.
3: **while** $\mathcal{J}_{\mathcal{L}} \neq \emptyset$ **do**
4: **for** all $\eta, \eta' \in \mathcal{L}.nodes$ such that $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$ **do**
5: $\mathcal{L}.nodes \leftarrow \mathcal{L}.nodes \cup \{\text{joint}(\eta, \eta', \mathcal{L})\}$ ▷ extend the lattice with the new joint node.
6: $\mathcal{J}_{\mathcal{L}} \leftarrow \mathcal{J}_{\mathcal{L}} \setminus \{(\eta.clock, \eta'.clock)\}$
7: **end for**
8: $\mathcal{J}_{\mathcal{L}} \leftarrow$ JCOMPUTE
9: **end while**
10: **end procedure**

Algorithm 6 JCOMPUTE

1: **procedure** JCOMPUTE
2: **for** all $\eta, \eta', \eta'' \in \mathcal{L}.nodes$ **do**
3: **if** $\eta'' \rightarrowtail \eta \wedge \eta'' \rightarrowtail \eta'$ **then** ▷ if η and η' are associated to two concurrent events.
4: $\mathcal{J}_{\mathcal{L}} = \mathcal{J}_{\mathcal{L}} \cup \{(\eta.clock, \eta'.clock)\}$ ▷ $\eta.clock$ and $\eta'.clock$ are added to relation $\mathcal{J}_{\mathcal{L}}$.
5: **end if**
6: **end for**
7: **return** $\mathcal{J}_{\mathcal{L}}$
8: **end procedure**

are picked up in the same order as they have been stored in the queue (FIFO queue).

REMOVEEXTRANODES (Algorithm 8): For optimization reasons, after extending the lattice by an action event, procedure REMOVEEXTRANODES is called to eliminate some (possibly existing) nodes of the lattice. A node in the lattice can be removed if the lattice no longer can be extended from that node. Having two nodes of the lattice η and η' such that every clock in the vector clock of η' is strictly greater than the respective clock of η, one can remove node η. This is due to the fact that the algorithm never receives an action event which could have extended the lattice from η where the lattice has already took into account an occurrence of event which has greater clocks stamp than $\eta.clock$.

Example 13 (Lattice construction) *Figure 8a depicts the computation lattice according to the received sequence of events concerning trace t_2 of Example 8. Node $((d_1, d_2, f_3), (0,1))$ is associated to event $(Fill_{12}, (1,0))$ and node $((f_1, f_2, d_3), (1,0))$ is associated to event $(Fill_3, (0,1))$. Since these two events are concurrent, joint node $((f_1, f_2, f_3), (1,1))$ is made. Node $\left((f_1, \bot_2^2, \bot_3^2), (1,2)\right)$ is associated to event $(Drain_{23}, (1,2))$. Due to vector clock update technique, the node with vector clock of $(0,2)$ is not created.*

5.4 Insensibility of Algorithm MAKE to the Communication Delay

Algorithm MAKE can be defined over a sequence of events received by the observer $\zeta = e_1 \cdot e_2 \cdot e_3 \cdots e_z \in E^*$ in the sense that one can apply MAKE sequentially from e_1 to e_z initialized by taking event e_1, the initial lattice $init_{\mathcal{L}}$ and an empty queue.

***Proposition* 2 (Insensitivity to the reception order)** *For any two sequences of events $\zeta, \zeta' \in E^*$, we have $\left(\forall S_j \in \mathbf{S} : \zeta \downarrow_{S_j} = \zeta' \downarrow_{S}\right.$ MAKE(ζ) = MAKE(ζ'), where $\zeta \downarrow_{S_j}$ is the projection of ζ on scheduler S_j which results the sequence of events generated by S_j.*

Property 2 states that different ordering of the events does not affect the output result of Algorithm MAKE. Note, Proposition 2 assumes that all events in ζ and ζ' can be distinguished. For a sequence of events $\zeta \in E^*$, MAKE$(\zeta).lattice$ denotes the constructed computation lattice $\mathcal{L}$ by algorithm MAKE.

5.5 Correctness of Lattice Construction

Computation lattice $\mathcal{L}$ has an initial node $init_{\mathcal{L}}$ which is the node with the smallest vector clock, and a *frontier* node which is the node with the greatest vector clock. A path of the constructed computation lattice $\mathcal{L}$ is a sequence

Algorithm 7 CHECKQUEUE

```
1:  procedure CHECKQUEUE
2:      while true do
3:          κ′ ← κ
4:          for all z ∈ [1, length(κ)] do
5:              MAKE(κ(z), true)                                   ▷ recall the events of the queue.
6:          end for
7:          if κ = κ′ then
8:              break                       ▷ break if none of the events in the queue is used.
9:          end if
10:     end while
11: end procedure
```

Algorithm 8 REMOVEEXTRANODES

```
1: procedure REMOVEEXTRANODES
2:     for all η ∈ L.nodes do
3:         if ∀j ∈ [1, m], ∃η′ ∈ L.nodes : η′.clock[j] > η.clock[j] then    ▷ if there exists a node with a strictly
   greater clocks in the vector clock.
4:             remove(L.nodes, η)                 ▷ the node with the smaller vector clock is removed.
5:         end if
6:     end for
7: end procedure
```

of causally-related nodes of the lattice, starting from the initial node and ending up in the frontier node.

Definition 20 (Set of the paths of a lattice) *The set of the paths of a constructed computation lattice $\mathcal{L}$ is* $\Pi(\mathcal{L}) = \Big\{\eta_0 \cdot \alpha_1 \cdot \eta_1 \cdot \alpha_2 \cdot \eta_2 \cdots \alpha_z \cdot \eta_z \mid \eta_0 = init_{\mathcal{L}} \wedge \forall r \in [1, z] : \Big(\eta_{r-1} \overset{\alpha_r}{\rightarrowtail\!\!\twoheadrightarrow} \eta_r \vee (\exists N \subseteq \mathcal{L}.nodes : \eta_{r-1} = \mathrm{meet}(N, \mathcal{L}) \wedge \eta_r = \mathrm{joint}(N, \mathcal{L}) \wedge \forall \eta \in N : \eta_{r-1} \overset{a_\eta}{\rightarrowtail\!\!\twoheadrightarrow} \eta \wedge \alpha_r = \bigcup_{\eta \in N} a_\eta)\Big)\Big\}$*, where the notions of* meet *and* joint *are naturally extended over a set of nodes.*

A path crosses over a series of nodes of the lattice either (i) the prior node is in $\rightarrowtail\!\!\twoheadrightarrow$ relation with the next node or (ii) the prior and the next node are the meet and the joint of a set of existing nodes respectively. A path from a meet node to the associated joint node represents an execution of a set of concurrent joint actions.

Example 14 (Set of the paths of a lattice) *In the computation lattice $\mathcal{L}$ depicted in Figure 8a, there are three distinct paths that begin from the initial node $((d_1, d_2, d_3), (0, 0))$ and end up to the frontier node $\big((f_1, \perp_2^2, \perp_3^2), (1, 2)\big)$. The set of paths is $\Pi(\mathcal{L}) = \{\pi_1, \pi_2, \pi_3\}$, where:*

- $\pi_1 = ((d_1, d_2, d_3), (0, 0)) \cdot \{Fill_{12}\} \cdot ((f_1, f_2, d_3), (1, 0)) \cdot \{\{Fill_3\}\} \cdot ((f_1, f_2, f_3), (1, 1)) \cdot \{Drain_{23}\} \cdot ((f_1, \perp_2^2, \perp_3^2), (1, 2))$,
- $\pi_2 = ((d_1, d_2, d_3), (0, 0)) \cdot \{\{Fill_3\}\} \cdot ((d_1, d_2, f_3), (0, 1)) \cdot \{Fill_{12}\} \cdot ((f_1, f_2, f_3), (1, 1)) \cdot \{Drain_{23}\} \cdot ((f_1, \perp_2^2, \perp_3^2), (1, 2))$,
- $\pi_3 = ((d_1, d_2, d_3), (0, 0)) \cdot \{Fill_{12}, \{Fill_3\}\} \cdot ((f_1, f_2, f_3), (1, 1)) \cdot \{Drain_{23}\} \cdot ((f_1, \perp_2^2, \perp_3^2), (1, 2))$.

Let us consider a distributed CBS consisting of a set of components $\mathbf{B}$ (as per Definition 1) and a set of schedulers $\mathbf{S}$ (as per Definition 2) with the global behavior as per Definition 4. At runtime, the execution of such a system produces a global trace $t = q^0 \cdot (\alpha^1 \cup \beta^1) \cdot q^1 \cdot (\alpha^2 \cup \beta^2) \cdots (\alpha^k \cup \beta^k) \cdot q^k$ which consists of (incomplete) global states and global actions (as per Definition 6). Due to the occurrence of simultaneous interactions and internal actions, each global trace t can be represented as a set of compatible global traces, which could have happened in the system at runtime.

Definition 21 (Compatible traces of a global trace) *The set of all compatible global traces of global trace t is* $\mathcal{P}(t) = \{t' \in Q \cdot (GAct \cdot Q)^* \mid \forall j \in [1, |\mathbf{S}|], t' \downarrow_{S_j} = t \downarrow_{S_j} = s_j(t)\}$.

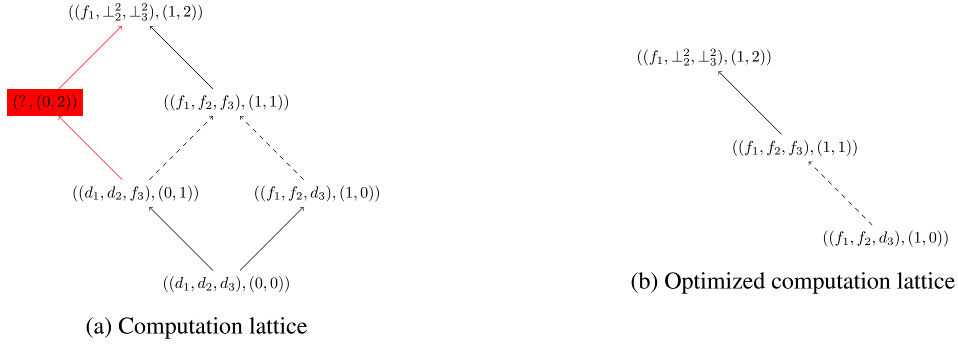


Figure 8: Computation lattice associated to trace t_2 in Example 4

Trace t' is compatible with trace t if the projection of both t and t' on scheduler S_j, for $j \in [1, |\mathbf{S}|]$, results the local trace of scheduler S_j. In a global trace, for each global action which consists of several concurrent interactions and internal actions of different schedulers, one can define different ordering of those concurrent interactions, each of which represents a possible execution of that global action. Consequently, several compatible global traces can be encoded from a global trace.

Note that two compatible traces with only difference in the ordering of their internal actions are considered as a unique compatible trace. What matters in the compatible traces of a global trace is the different ordering of interactions.

Example 15 (The set of compatible global traces) *Let us consider the incomplete global trace t_1 described in Example 8, that is $t_1 = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{Drain_1\}, \{Fill_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$. The projection of t_1 on each scheduler is represented as follow:*

- $t_1 \downarrow_{S_1} = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, ?) \cdot \{\beta_1\} \cdot (f_1, \bot, ?) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, ?) \cdot \{\beta_2\} \cdot (\bot, f_2, ?)$,
- $t_1 \downarrow_{S_2} = (d_1, d_2, d_3) \cdot \{Fill_3\} \cdot (?, d_2, \bot)$.

The set of compatible global traces is $\mathcal{P}(t_1) = \{t_1^1, t_1^2, t_1^3, t_1^4, t_1^5\}$ where:

- $t_1^1 = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{Fill_3\}, \{Drain_1\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$,
- $t_1^2 = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, d_3) \cdot \{\{Fill_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$,
- $t_1^3 = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{Fill_3\}\} \cdot (f_1, \bot, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$,
- $t_1^4 = (d_1, d_2, d_3) \cdot \{Fill_{12}, \{Fill_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_1\} \cdot (f_1, \bot, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$,
- $t_1^5 = (d_1, d_2, d_3) \cdot \{\{Fill_3\}\} \cdot (d_1, d_2, \bot) \cdot \{Fill_{12}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_1\} \cdot (f_1, \bot, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (\bot, f_2, \bot)$.

For monitoring purposes it is necessary to represent the run of the system by a sequence of complete global states (recall that the monitored property is defined over the complete global states). To this end, by inspiring the technique introduced in [17], we define a function which takes as input a global trace of the distributed system (i.e., a sequence of incomplete global states) and outputs an equivalent global trace in which all the internal actions (β) are removed from the trace and instead the updated state after each internal action is used to complete the states of the global trace.

Definition 22 (Function refine $\mathcal{R}_\beta$) *Function $\mathcal{R}_\beta : Q \cdot (GAct \cdot Q)^* \longrightarrow Q \cdot (Int \cdot Q)^*$ is defined as:*

- $\mathcal{R}_\beta(init) = init$,
- $\mathcal{R}_\beta(\sigma \cdot (\alpha \cup \beta) \cdot q) = \begin{cases} \mathcal{R}_\beta(\sigma) \cdot \alpha \cdot q & \textit{if } \beta = \emptyset, \\ \text{map } [x \mapsto \text{upd}(q, x)] \ (\mathcal{R}_\beta(\sigma)) & \textit{if } \alpha = \emptyset, \\ \text{map } [x \mapsto \text{upd}(q, x)] \ (\mathcal{R}_\beta(\sigma) \cdot \alpha \cdot q) & \textit{otherwise;} \end{cases}$

with $\text{upd} : Q \times (Q \cup 2^{Int}) \longrightarrow Q \cup 2^{Int}$ *defined as:*

– $\text{upd}((q_1, \ldots, q_{|\mathbf{B}|}), \alpha) = \alpha$,

– $\text{upd}\Big((q_1, \ldots, q_{|\mathbf{B}|}), (q'_1, \ldots, q'_{|\mathbf{B}|})\Big) = (q''_1, \ldots, q''_{|\mathbf{B}|})$,

where $\forall k \in [1, |\mathbf{B}|], q''_k = \begin{cases} q_k & \text{if } (q_k \notin Q_k^{\text{b}}) \wedge (q'_k \in Q_k^{\text{b}}) \\ q'_k & \text{otherwise.} \end{cases}$

Function $\mathcal{R}_\beta$ uses the (information in the) state after internal actions in order to update the incomplete states using function upd.

By applying function $\mathcal{R}_\beta$ over the set of compatible global traces $\mathcal{P}(t)$, we obtain a new set of global traces which is (i) equivalent to $\mathcal{P}(t)$ (according to [17], Definition 5), (ii) internal actions are discarded in the presentation of each global trace and (iii) contains maximal complete states that can be built with the information contained in the incomplete states observed so far.

Example 16 (Applying function $\mathcal{R}_\beta$) *By applying function $\mathcal{R}_\beta$ over the set of compatible global traces in Example 15 we have the refined traces:*

- $\mathcal{R}_\beta(t_1^1) = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (f_1, f_2, d_3) \cdot \{\{Drain_1\}, \{Fill_3\}\} \cdot (\bot, f_2, \bot)$,
- $\mathcal{R}_\beta(t_1^2) = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (f_1, f_2, d_3) \cdot \{\{Drain_1\}\} \cdot (\bot, f_2, d_3) \cdot \{\{Fill_3\}\} \cdot (\bot, f_2, \bot)$,
- $\mathcal{R}_\beta(t_1^3) = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (f_1, f_2, d_3) \cdot \{\{Fill_3\}\} \cdot (f_1, f_2, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, f_2, \bot)$,
- $\mathcal{R}_\beta(t_1^4) = (d_1, d_2, d_3) \cdot \{Fill_{12}, \{Fill_3\}\} \cdot (f_1, f_2, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, f_2, \bot)$,
- $\mathcal{R}_\beta(t_1^5) = (d_1, d_2, d_3) \cdot \{\{Fill_3\}\} \cdot (d_1, d_2, \bot) \cdot \{Fill_{12}\} \cdot (f_1, f_2, \bot) \cdot \{\{Drain_1\}\} \cdot (\bot, f_2, \bot)$.

In Sec. 3.2 (Definition 7) we define $\{s_1(t), \ldots, s_m(t)\}$, the set of observable local traces of the schedulers obtained from global trace t. According to Definition 13, from each local trace we can obtain the sequences of events generated by the controller of each scheduler, such that the set of all the sequences of the events is $\{\text{event}(s_1(t)), \ldots, \text{event}(s_m(t))\}$ with $\text{event}(s_j(t)) \in E^*$ for $j \in [1, |\mathbf{S}|]$.

In the following, we define the set of all possible sequences of events that could be received by the observer.

Definition 23 (Possible events ordering) *Considering global trace t, the set of all possible sequences of events that could be received by the observer is* $\Theta(t) = \{\zeta \in E^* \mid \forall j \in [1, |\mathbf{S}|] : \zeta \downarrow_{S_j} = \text{event}(s_j(t))\}$.

Events are received by the observer in any order just under a condition in which the ordering among the local events of a scheduler is preserved.

***Proposition* 3 (Soundness)** $\forall \zeta \in \Theta(t), \forall \pi \in \Pi(\text{MAKE}(\zeta).\mathit{lattice}), \forall j \in [1, |\mathbf{S}|] : \pi \downarrow_{S_j} = \mathcal{R}_\beta(s_j(t))$.

Property 3 states that the projection of all paths in the lattice on a scheduler S_j for $j \in [1, |\mathbf{S}|]$ results in the refined local trace of scheduler S_j. The following proposition states the correctness of the construction in the sense that applying Algorithm MAKE over a sequence of observed events (i.e., $\zeta \in \Theta$) at runtime, results a computation lattice which encodes a set of the sequences of global states, such that each sequence represents a global trace of the system.

***Proposition* 4 (Completeness)** *Given a global trace t as per Definition 6, we have*

$\forall \zeta \in \Theta(t), \forall t' \in \mathcal{P}(t), \exists! \pi \in \Pi\Big(\text{MAKE}(\zeta).\mathit{lattice}\Big) : \pi = \mathcal{R}_\beta(t')$.

π said to be the associated path of the compatible trace t'.

Applying algorithm MAKE over any of the sequence of events constructs a computation lattice whose set of paths consists on all the compatible global traces.

Example 17 (Existence of the set of compatible global traces in the constructed lattice) *Let us consider global trace t_1 presented in Example 4 and the set of all associated event of global trace t_1 that is presented in Example 8. Events are received by the observer in order to make the lattice. Figure 10, illustrates the associated constructed computation lattice using algorithm* MAKE *consists of 5 paths π_1 to π_5. The set of refined compatible global traces (presented in Example 16) can be extracted from the reconstructed lattice, where $\pi_k = \mathcal{R}_\beta(t_1^k)$ for $k \in [1, 5]$. Paths π_1 to π_5 are associated paths of the compatible traces t_1^1 to t_1^5 respectively.*

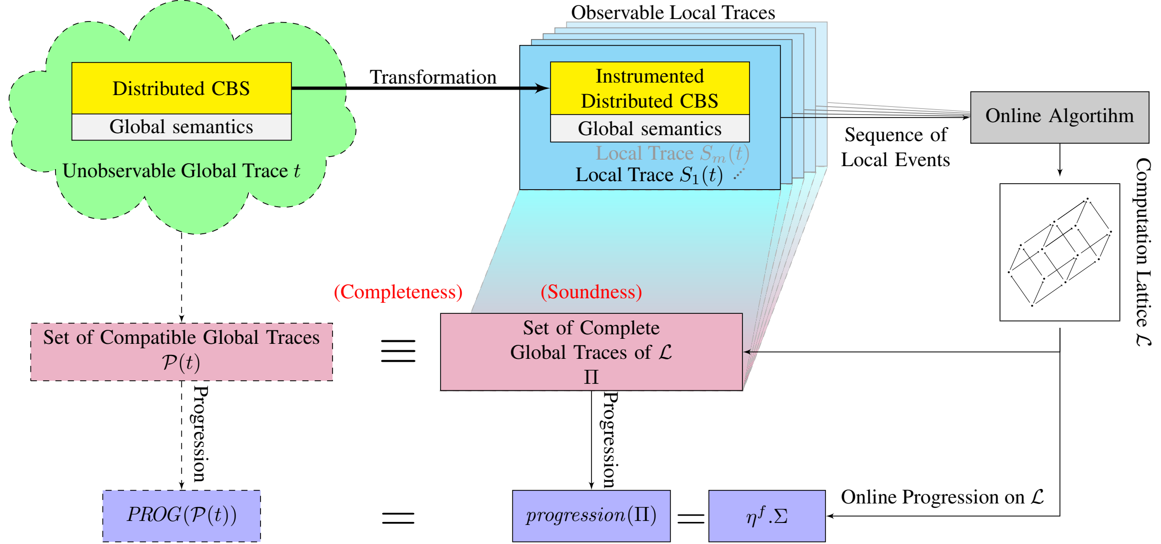


Figure 9: Approach overview

6 LTL Runtime Verification by Progression on the Reconstructed Computation Lattice

In this section, we address the problem of monitoring an LTL formula specifying the desired global behavior of the system.

In the usual case, evaluating whether an LTL formula holds requires the monitoring procedure to have access to the complete information about the system state, that is ready states of CBSs. Since components have busy states as well as ready states, the global trace of the system is a sequence of incomplete global states (*cf.* Definition 6). An incomplete global state is a sort of global state in which the state of some components are possibly unknown. Although by stabilizing the system to have all the components' ready-states we could obtain a complete computation lattice using algorithm MAKE (*cf.* Sec. 5.3), that is the state of each node is a complete global state, instead, we propose an on-the-fly verification of an LTL property during the construction of computation lattice.

There are many approaches to monitor LTL formulas based on various finite-trace semantics (*cf.* [3]). One way of looking at the monitoring problem for some LTL formula φ is described in [4] based on formula rewriting, which is also known as formula progression, or just *progression*. Progression splits a formula into (i) a formula expressing what needs to be satisfied by the observed events so far and (ii) a new formula (referred to a *future goal*), which has to be satisfied by the trace in the future. We apply progression over a set of finite traces, where each trace consists in a sequence of (possibly) incomplete global states, encoded from the constructed computation lattice. An important advantage of this technique is that it often detects when a formula is violated or validated before the end of the execution trace, that is, when the constructed lattice is not complete, so it is suitable for online monitoring.

To monitor the execution of a distributed CBS with multi-party interaction with respect to an LTL property φ, we introduce a more informative computation lattice by attaching to the each node of lattice $\mathcal{L}$ a set of formula. Given a computation lattice $\mathcal{L} = (N, \rightarrowtail\!\!\!\!\rightarrow)$ (as per Definition 14), we define an augmented computation lattice $\mathcal{L}^\varphi$ as follow.

Definition 24 (Computation lattice augmentation) *$\mathcal{L}^\varphi$ is a pair $(N^\varphi, \rightarrowtail\!\!\!\!\rightarrow)$, where $N^\varphi \subseteq Q^l \times VC \times 2^{LTL}$ is the set of nodes augmented by 2^{LTL}, that is the set of LTL formulas. The initial node is $init^\varphi_\mathcal{L} = (init, (0, \ldots, 0), \{\varphi\})$ with $\varphi \in LTL$ the global desired property.*

In the newly defined computation lattice, a set of LTL formulas is attached to each node. The set of formulas attached to a node represents the different evaluation of the property φ with respect to different possible paths form the initial node to the node. The state and the vector clock associated to each node and the happened-before relation are defined similar to the initial definition of computation lattice (*cf.* Definition 14).

The construction of the augmented computation lattice requires some modifications to algorithm MAKE:

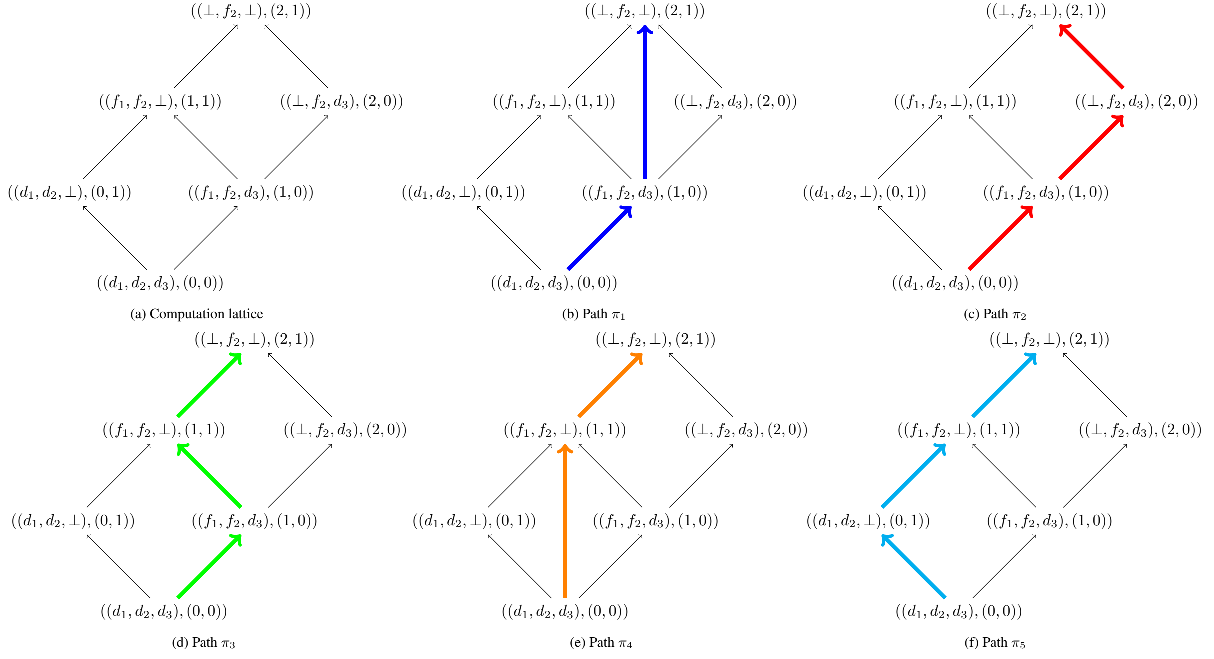


Figure 10: Computation lattice, all the associated paths and compatible traces associated to trace t_1 in Example 4

- Lattice $\mathcal{L}^\varphi$ initially has node $init_{\mathcal{L}}^{\varphi} = (init, (0, \ldots, 0), \{\varphi\})$.
- The creation of a new node η in the lattice with $\eta.state = q$ and $\eta.clock = vc$, calculates the set of formulas Σ associated to η using the progression function (see Definition 25). The augmented node is $\eta = (q, vc, \Sigma)$, where $\Sigma = \{\mathrm{prog}(LTL', q) \mid LTL' \in \eta'.\Sigma \wedge (\eta' \rightarrowtail\!\!\!\!\twoheadrightarrow \eta \vee \exists N \subseteq \mathcal{L}^\varphi.nodes : \eta' = \mathrm{meet}(N, \mathcal{L}) \wedge \eta = \mathrm{joint}(N, \mathcal{L}))\}$. We denote the set of formulas of node $\eta \in \mathcal{L}^\varphi.nodes$ by $\eta.\Sigma$.
- Updating node $\eta = (q, vc, \Sigma)$ by update event $e = (\beta_i, q_i) \in E_\beta, i \in [1, n]$ which is sent by scheduler $S_j, j \in [1, m]$ updates all associated formulas Σ to Σ' using the update function (see Definition 26), where $\Sigma' = \left\{\mathrm{upd}_\varphi(LTL, q_i, j) \mid LTL \in \Sigma\right\}$.

Definition 25 (Progression function) *prog : $LTL \times Q^l \longrightarrow LTL$ is defined using a pattern-matching with $p \in AP_{i\in[1,n]}$ and $q = (q_1, \ldots, q_n) \in Q^l$.*

$$
\begin{array}{rcl}
\mathrm{prog}(\varphi, q) & = & \mathtt{match}(\varphi)\ \mathtt{with} \\
& | & p \in AP_{i\in[1,n]} \rightarrow \begin{cases} T & \text{if } q_i \in Q_i^{\mathrm{r}} \wedge p \in q_i \\ F & \text{if } q_i \in Q_i^{\mathrm{r}} \wedge p \notin q_i \\ \boldsymbol{X}_\beta^k p & \text{otherwise } (q_i = \bot_i^k, k \in [1, m]) \end{cases} \\
& | & \boldsymbol{X}_\beta^k p \rightarrow \boldsymbol{X}_\beta^k p \\
& | & \varphi_1 \vee \varphi_2 \rightarrow \mathrm{prog}(\varphi_1, q) \vee \mathrm{prog}(\varphi_2, q) \\
& | & \varphi_1 \boldsymbol{U} \varphi_2 \rightarrow \mathrm{prog}(\varphi_2, q) \vee \mathrm{prog}(\varphi_1, q) \wedge \varphi_1 \boldsymbol{U} \varphi_2 \\
& | & \boldsymbol{G}\varphi \rightarrow \mathrm{prog}(\varphi, q) \wedge \boldsymbol{G}\varphi \\
& | & \boldsymbol{F}\varphi \rightarrow \mathrm{prog}(\varphi, q) \vee \boldsymbol{F}\varphi \\
& | & \boldsymbol{X}\varphi \rightarrow \varphi \\
& | & \neg\varphi \rightarrow \neg\, \mathrm{prog}(\varphi, q) \\
& | & T \rightarrow T
\end{array}
$$

We define a new modality $\boldsymbol{X}_\beta$ such that $\boldsymbol{X}_\beta^k p$ for $p \in AP_{i\in[1,n]}$ and $k \in [1, m]$ means that atomic proposition p has to hold at next ready state of component B_i which is sent by scheduler S_k. For a sequence of global states obtained at runtime $\sigma = q_0 \cdot q_1 \cdot q_2 \cdots$ such that $\sigma_j = q_j$, we have $\sigma_j \models \boldsymbol{X}_\beta^k p \Leftrightarrow \sigma_z \models p$ where $z = \min\left(\left\{r > j \;\middle|\; (\sigma_{r-1} \downarrow_{S_k}) \xrightarrow{\beta_i}_{S_k} (\sigma_r \downarrow_{S_k})\right\}\right)$.

The truth value of the progression of an atomic proposition $p \in AP_i$ for $i \in [1, n]$ with a global state $q = (q_1, \ldots, q_n)$ is evaluated by true (*resp.* false) if the state of component B_i (that is q_i) is a ready state and satisfies (*resp.* does not satisfy) the atomic proposition p. If the state of component B_i is not a ready state, the evaluation of the atomic proposition p is postponed to the next ready state of component B_i.

Definition 26 (Formula update function) $\mathrm{upd}_\varphi : LTL \times \{Q_i^{\mathrm{r}}\}_{i=1}^{n} \times [1, m] \rightarrow LTL$ *is defined using a pattern-matching with* $q_i \in Q_i^{\mathrm{r}}$ *for* $i \in [1, n]$.

$$
\begin{array}{rcl}
\mathrm{upd}_\varphi(\varphi, q_i, j) & = & \texttt{match}(\varphi)\ \texttt{with} \\
& | & \boldsymbol{X}_\beta^k p \rightarrow \begin{cases} T & \text{if } p \in AP_i \cap q_i \wedge k = j \\ F & \text{if } p \in AP_i \cap \overline{q_i} \wedge k = j \\ \boldsymbol{X}_\beta^k p & \text{otherwise } (p \notin AP_i \vee k \neq j) \end{cases} \\
& | & \varphi_1 \vee \varphi_2 \rightarrow \mathrm{upd}_\varphi(\varphi_1, q_i) \vee \mathrm{upd}_\varphi(\varphi_2, q_i) \\
& | & \varphi_1 \wedge \varphi_2 \rightarrow \mathrm{upd}_\varphi(\varphi_1, q_i) \wedge \mathrm{upd}_\varphi(\varphi_2, q_i) \\
& | & \varphi_1 \boldsymbol{U} \varphi_2 \rightarrow \mathrm{upd}_\varphi(\varphi_1, q_i) \boldsymbol{U}\, \mathrm{upd}_\varphi(\varphi_2, q_i) \\
& | & \boldsymbol{G}\varphi \rightarrow \boldsymbol{G}\, \mathrm{upd}_\varphi(\varphi, q_i) \\
& | & \boldsymbol{F}\varphi \rightarrow \boldsymbol{F}\, \mathrm{upd}_\varphi(\varphi, q_i) \\
& | & \boldsymbol{X}\varphi \rightarrow \boldsymbol{X}\, \mathrm{upd}_\varphi(\varphi, q_i) \\
& | & \neg\varphi \rightarrow \neg\, \mathrm{upd}_\varphi(\varphi, q_i) \\
& | & T \rightarrow T \\
& | & p \in AP_{i \in [1,n]} \rightarrow p
\end{array}
$$

Update function updates a progressed LTL formula with respect to a ready state of a component. Intuitively, a formula consists in an atomic proposition whose truth or falsity depends on the next ready state of component B_i sent by scheduler S_k, that is $\mathbf{X}_\beta^k p$ where $p \in AP_i$, can be evaluated using update function by taking the first ready state of component B_i received from scheduler S_k after the formula rewrote to $\mathbf{X}_\beta^k p$.

Example 18 (Formula progression and formula update over an augmented computation lattice) *Let consider the system in Example 4 with global trace t_2 and the received sequence of events presented in Example 8 and desired property $\varphi = \boldsymbol{G}(d_3 \vee f_1)$. $\mathcal{L}^\varphi$ initially has node $init_{\mathcal{L}}^{\varphi} = ((d_1, d_2, d_3), (0, 0), \{\varphi\})$. By observing the action event $(Fill_{12}, (1, 0))$, algorithm* Make *creates new node $\eta_1 = ((\bot, \bot, d_3), (1, 0), \{\varphi\})$, because* $\mathrm{prog}(\boldsymbol{G}(d_3 \vee f_1), (\bot, \bot, d_3)) = \mathrm{prog}((d_3 \vee f_1), (\bot, \bot, d_3)) \wedge \boldsymbol{G}(d_3 \vee f_1) = T \wedge \boldsymbol{G}(d_3 \vee f_1) = \boldsymbol{G}(d_3 \vee f_1) = \varphi$.

By observing the action event $(Fill_3, (0, 1))$, new node $\eta_2 = ((d_1, d_2, \bot), (0, 1), \{\boldsymbol{X}_\beta d_3 \wedge \varphi\})$ is created, because $\mathrm{prog}(\boldsymbol{G}(d_3 \vee f_1), (d_1, d_2, \bot)) = \mathrm{prog}((d_3 \vee f_1), (d_1, d_2, \bot)) \wedge \boldsymbol{G}(d_3 \vee f_1) = \boldsymbol{X}_\beta d_3 \wedge \boldsymbol{G}(d_3 \vee f_1) = \boldsymbol{X}_\beta d_3 \wedge \varphi$. *Formula $\boldsymbol{X}_\beta d_3$ means that the evaluation of incomplete state $(d_1, d_2, \bot)$ with respect to formula $(d_3 \vee f_1)$ is on hold until the next ready state of component $Tank_3$.*

Consequently joint node $\eta_3 = ((\bot, \bot, \bot), (1, 1), \{(\boldsymbol{X}_\beta d_3) \wedge (\boldsymbol{X}_\beta d_3 \vee \boldsymbol{X}_\beta f_1) \wedge \varphi, (\boldsymbol{X}_\beta d_3 \vee \boldsymbol{X}_\beta f_1) \wedge \varphi, (\boldsymbol{X}_\beta d_3 \vee \boldsymbol{X}_\beta f_1) \wedge \varphi\})$ is made. The update event (β_3, f_3) updates both state and the set of formulas associated to each node as follows. Although $init_{\mathcal{L}}^{\varphi}$ and η_1 remain intact, but node η_2 is updated to $((d_1, d_2, f_3), (0, 1), \{F\})$ because $\mathrm{upd}_\varphi(\boldsymbol{X}_\beta d_3 \wedge \varphi, f_3) = F$. *Moreover, node η_3 is updated to $\eta_3 = ((\bot, \bot, f_3), (1, 1), \{F, (\boldsymbol{X}_\beta f_1) \wedge \varphi, (\boldsymbol{X}_\beta f_1) \wedge \varphi\})$.*

The update event (β_2, f_2) updates nodes η_1 and η_3 such that $\eta_1 = ((\bot, f_2, d_3), (1, 0), \{\varphi\})$ and $\eta_3 = ((\bot, f_2, f_3), (1, 1), \{F, (\boldsymbol{X}_\beta f_1) \wedge \varphi, (\boldsymbol{X}_\beta f_1) \wedge \varphi\})$.

By observing the action event $(Drain_{23}, (1, 2))$, the new node $\eta_4 = ((\bot, \bot, \bot), (1, 2), \{F, (\boldsymbol{X}_\beta d_3 \vee \boldsymbol{X}_\beta f_1) \wedge (\boldsymbol{X}_\beta f_1) \wedge \varphi, (\boldsymbol{X}_\beta d_3 \vee \boldsymbol{X}_\beta f_1) \wedge (\boldsymbol{X}_\beta f_1) \wedge \varphi\})$ is created.

The update event (β_1, f_1) updates nodes η_1, η_3 and η_4 such that $\eta_1 = ((f_1, f_2, d_3), (1, 0), \{\varphi\})$, $\eta_3 = ((f_1, f_2, f_3), (1, 1), \{F, \varphi, \varphi\})$ and $\eta_4 = ((f_1, \bot, \bot), (1, 2), \{F, \varphi, \varphi\})$.

6.1 Correctness of Formula Progression on the Lattice

In Sec. 5, we introduced how from an unobservable global trace t of a distributed CBS with multi-party interactions one can construct a set of paths compatible with t in form of a lattice. Furthermore, in Sec. 6 we adapted formula progression over the constructed lattice with respect to a given LTL formula φ. What we obtained is a directed lattice $\mathcal{L}^\varphi$ starting from the initial node and ending up with frontier node η^f. The set of formulas attached to the frontier node, that is $\eta^f.\Sigma$, represents the progression of the initial formula over the set of path of the lattice.

Definition 27 (Progression on a global trace) *Function* $PROG : LTL \times Q \cdot (GAct \cdot Q)^* \rightarrow LTL$ *is defined as:*

Table 1: On-the-fly construction and verification of computation lattice

step	event	lattice
0	ϵ	$((d_1, d_2, d_3), (0,0), \{\varphi\})$
1	- receiving event $Fill_{12}(1, 0)$ - extending by making a new node - the formula projection of new node	$((\bot, \bot, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$
2	- receiving event $Fill_3(0, 1)$ - extending by making a new node - extending by making the joint node - the formula projection of new nodes - three formulas attached to the frontier represent the evaluation of three paths	$((\bot, \bot, \bot), (1,1), \{(\mathbf{X}_\beta d_3) \wedge (\mathbf{X}_\beta d_3 \vee \mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta d_3 \vee \mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta d_3 \vee \mathbf{X}_\beta f_1) \wedge \varphi\})$ $((d_1, d_2, \bot), (0,1), \{\mathbf{X}_\beta d_3 \wedge \varphi\})$ $((\bot, \bot, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$
3	- receiving event (β_3, f_3) - updating states of the existing nodes - updating the formulas of existing nodes	$((\bot, \bot, f_3), (1,1), \{F, (\mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta f_1) \wedge \varphi\})$ $((d_1, d_2, f_3), (0,1), \{F\})$ $((\bot, \bot, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$
4	- receiving event (β_2, f_2) - updating states of the existing nodes - updating the formulas of existing nodes	$((\bot, f_2, f_3), (1,1), \{F, (\mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta f_1) \wedge \varphi\})$ $((d_1, d_2, f_3), (0,1), \{F\})$ $((\bot, f_2, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$
5	- receiving event $Drain_{23}(1, 2)$ - extending by making a new node - the formula projection of new node	$((\bot, \bot, \bot), (1,2), \{F, (\mathbf{X}_\beta d_3 \vee \mathbf{X}_\beta f_1) \wedge (\mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta d_3 \vee \mathbf{X}_\beta f_1) \wedge (\mathbf{X}_\beta f_1) \wedge \varphi\})$ $((\bot, f_2, f_3), (1,1), \{F, (\mathbf{X}_\beta f_1) \wedge \varphi, (\mathbf{X}_\beta f_1) \wedge \varphi\})$ $((d_1, d_2, f_3), (0,1), \{F\})$ $((\bot, f_2, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$
6	- receiving event (β_1, f_1) - updating states of the existing nodes - updating the formulas of existing nodes	$((f_1, \bot, \bot), (1,2), \{F, \varphi, \varphi\})$ $((f_1, f_2, f_3), (1,1), \{F, \varphi, \varphi\})$ $((d_1, d_2, f_3), (0,1), \{F\})$ $((f_1, f_2, d_3), (1,0), \{\varphi\})$ $((d_1, d_2, d_3), (0,0), \{\varphi\})$

- $PROG(\varphi, init) = \varphi$,
- $PROG(\varphi, \sigma) = \varphi'$
- $PROG\left(\varphi, \sigma \cdot (\alpha \cup \beta) \cdot q\right) = \mathrm{prog}\left(UPD\left(PROG\left(\varphi, \sigma\right), \mathcal{Q}\right), q\right)$ *where*
  - $\mathcal{Q} = \{q[i] \mid \beta_i \in \beta\}$ *is the set of updated states,*
  - *function* $UPD : LTL \times Q^R \rightarrow LTL$ *is defined as:*
    - $UPD\left(\varphi, \{\epsilon\}\right) = \varphi$,
    - $UPD\left(\varphi, Q^r \cup \{q_i\}\right) = \mathrm{upd}_\varphi\left(UPD\left(\varphi, Q^r\right), q_i\right)$.

  with $Q^R \subseteq \{q \in Q_i^{\mathrm{r}} \mid i \in [1, n]\}$ *the set of subsets of ready states of the components.*

Function $PROG$ uses functions prog, upd_φ (Definitions 25 and 26) and function UPD. Since after each global action we only have one global state (likely incomplete), function upd_φ does not need to check among multiple global states to find whose formula must be updated. That is why $PROG$ uses the simplified version of function upd_φ by eliminating the scheduler index input. Moreover functions prog modified in such way to take as input a global state in Q instead of a global state in Q^l because as we above mentioned, the index of schedulers does not play a role in the progression of an LTL formula on a global trace.

Given a global state t as per Definition 6 and an LTL property φ, by $\eta^f.\Sigma$ we denote the set of LTL formulas of the frontier node of the constructed computation lattice $\mathcal{L}^\varphi$, we have the two following proposition and theorems:

***Proposition* 5** *Given an LTL formula φ and a global trace t, we have:*

$$PROG(\varphi, t) = \begin{cases} progression(\varphi, \mathcal{R}_\beta(t)) & \text{if } \mathrm{last}(t)[i] \in Q_i^{\mathrm{r}} \text{ for all } i \in [1, n], \\ progression(\varphi, \mathcal{R}_\beta(t')) & \text{otherwise.} \end{cases}$$

where $\exists \beta \subseteq \bigcup_{i \in [1,n]} \{\beta_i\} : t' = t \cdot \beta \cdot q, q[i] \in Q_i^{\mathrm{r}}, i \in [1, n]$ and progression is the standard progression function described in [4].

Proposition 5 states that progression of an LTL formula on a global trace of a distributed system, defined in Definition 6, using $PROG$ is similar to the standard progression of the LTL formula on the corresponding refined global trace using $progression$. Intuitively, progression of stabilized global trace t, results similar to the standard progression of refined t.

***Theorem* 1 (Soundness)** *For a global trace t and LTL formula φ, we have*

$$\forall \varphi' \in \eta^f.\Sigma, \exists t' \in \mathcal{P}(t) : PROG(\varphi, t') = \varphi'.$$

Theorem 1 states that each formula of the frontier node is derived from the progression of formula φ on a compatible trace of trace t.

***Theorem* 2 (Completeness)** *For a global trace t and an LTL formula φ, we have:*

$$\eta^f.\Sigma = \left\{PROG(\varphi, t') \mid t' \in \mathcal{P}(t)\right\}.$$

Theorem 2 states that the set of formulas in the frontier node is equal to the set of progression of φ on all the compatible traces of t.

7 Implementation

We present an implementation of our monitoring approach in a tool called RVDIST. RVDIST is a prototype tool implementing algorithm MAKE presented in Sec. 6, written in the C++ programming language. RVDIST takes as input a configuration file describing the architecture of the distributed system and a list of events. The configuration file has the parameters of system such as the number of schedulers, the number of components, the initial state of the system, the LTL formula to be monitored, the mapping of each atomic propositions to the components. The formula is monitored against the sequence of events by progression over the constructed computation lattice. RVDIST outputs the evaluation of the constructed lattice by reporting the number of observed events, the number of existing nodes of the constructed lattice, the number of nodes which have been removed from the lattice due to optimizing the size of the lattice, the vector clock of the frontier node, the number of paths from the initial node to the frontier node which have been monitored (the set of all compatible traces), the set of formulas associated to the frontier node. Figure 11 depicts the work-flow of RVDIST.

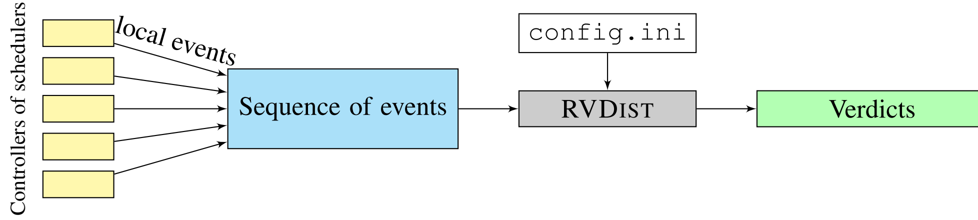


Figure 11: Overview of RVDIST work-flow

8 Evaluation

We present the evaluation of our monitoring approach on two case studies carried out with RVDIST.

8.1 Case Studies

We present a realistic example of a robot navigation and a model of two phase commit protocol (TPC).

8.1.1 Deadlock Freedom of Robotic Application ROBLOCO

The functional level of this navigating robot consists of a set of modules. ROBLOCO is in charge of the robot low-level controller. It has a *track* task associated to the activities *TSStart/TSStop* (TrackSpeedStart, TrackSpeedStop). *TSStart* reads data from the dedicated *speed* port and sends it to the motor *controller*. In parallel, the *manager* module, which is associated to the *odo* task activities (OdoStart and OdoStop) reads the signals from the encoders on the wheels and produces a current position on the *pos* port. ROBLASER is in charge of the laser. It has a *scan* task associated to the *StartScan/StopScan* activities. They produce the free space in the laser's range tagged with the position where the scan has been made. ROBMAP aggregates the successive *scan* data in the *map* port. ROBMOTION has one task plan which, given a goal position, computes the appropriate speed to reach it and writes it on *speed*, using the current position, and avoiding obstacles. We deal with the most complex module, i.e., ROBLOCO, involving three schedulers in charge of the execution of the dedicated actions. ROBLOCO has 34 components and 117 multi-party interactions synchronizing the actions of components. Since tasks are based on the specific sequence of the execution of interactions and tasks are not totally independent, there exist some share components which are involved in more than one task. To prevent deadlocks in the system, it is required that whenever the *controller* is in *free* state, at some point in future, the *signal* module must reach the *start* state before the *manager* starts managing a new *odo* task. The deadlock freedom requirement can be defined as LTL formula φ_1.

$$\varphi_1\text{: } \mathbf{G}\Big(\texttt{ControlFree} \implies (\mathbf{X}\neg\texttt{ManagerStartodo}\ \mathbf{U}\texttt{SignalStart})\Big)$$

8.1.2 Protocol Correctness of Two Phase Commit (TPC):

We consider the distributed transaction commit [11] problem where a set of nodes called *resource managers* $\{rm_1, rm_n, \ldots, rm_n\}$ have to reach agreement on whether to *commit* or *abort* a transaction. Resource managers are able to locally *commit* or *abort* a transaction based on a local decision. In a fault-free system, it is required the global system to commit as a whole if each resource manager has locally committed, and that it aborts as a whole if any of the resource managers has locally aborted. In case of global abort, locally-committed resource managers may perform roll-back steps to undo the effect of the last transaction [25].

Two phase commit protocol is a solution proposed by [10] to solve the transaction commit problem. It uses a *transaction manager* that coordinates between resource managers to ensure they all reach one global decision regarding a particular transaction. The global decision is made by the transaction manager based on the feedback it gets from resource managers after making their local decision (`LocalCommit/LocalAbort`).

The protocol, running on a transaction, uses a *client*, a transaction manager and a non-empty set of resource managers which are the active participants of the transaction. The protocol starts when client sends remote procedure to all the participating resource managers. Then each participating resource manager rm_i makes its local decision based on its local criteria and reports its local decision to transaction manager. $\texttt{LocalCommit}_i$ is true if resource manager rm_i can locally-commit the transaction, and $\texttt{LocalAbort}_i$ is true if resource manager rm_i

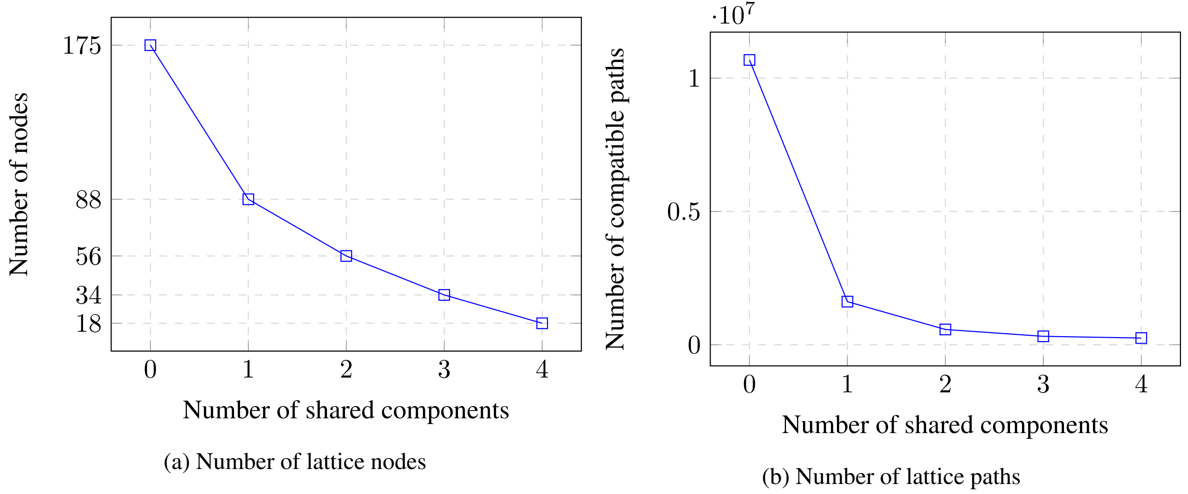


Figure 12: Lattice construction vs. number of shared components

cannot locally-commit the transaction. Each participant resource managers stays in wait location until it hears back from transaction manager whether to perform a global commit or abort for the current transaction. After all local decisions have been made and reported to transaction manager, the latter makes a global decision (`GlobalCommit`/`GlobalAbort`) that all the system will agree upon. When `GlobalCommit` is true, the system will globally-commit as a whole, and it will abort as a whole when `GlobalAbort` is true. We consider two specifications related to TPC protocol correctness:

$$\varphi_2\text{: } \mathbf{G}\Big(\bigwedge_{i=1}^{n}(\texttt{LocalAbort}_i \implies \mathbf{X}(\neg\texttt{LocalAbort}_i \wedge \neg\texttt{LocalCommit}_i)\ \mathbf{U}\texttt{GlobalAbort})\Big),$$

$$\varphi_3\text{: } \mathbf{G}\Big(\bigwedge_{i=1}^{n}\texttt{LocalCommit}_i \implies \mathbf{X}\Big(\bigwedge_{i=1}^{n}(\neg\texttt{LocalAbort}_i\wedge\neg\texttt{LocalCommit}_i)\Big)\ \mathbf{U}\texttt{GlobalCommit}\Big).$$

Property φ_2 states that, sending locally abort in any resource managers for a current transaction implies the global abort (`GlobalAbort`) on that transaction before the resource manager locally aborts or commits the next transaction, that is, none of the resource managers commit. Property φ_3 states that, if all the resource managers send locally commit for a current transaction, then all the resource managers commit the transaction (`GlobalCommit`) before the resource managers locally aborts or commits the next transaction.

For each system we applied the model transformation defined in Section 4.1 and run them in a distributed setting. Each instrumented system produces a sequence of event which is generated and sent from the controllers of its schedulers. The events are sent to the RVDIST where the associated configuration file is already given. Upon the reception of each event, RVDIST applies the online monitoring algorithm introduced in Section 6 and outputs the result consists of the information stored in the constructed computation lattice and evaluation of the desired LTL property so far.

In the following we investigate how the number of shared components effects on the size of the computation lattice over a very simple example of a distributed CBS with multiparty interaction.

Example 19 (Shared component, lattice size) *Let us consider a component-based system consists of four independent components $Comp_1, \ldots, Comp_4$. Each component has two actions $Action_1, Action_2$ which are designed to only be executed with the following order: $Action_1.Action_2.Action_1$ and then the component terminates. We distribute the execution of actions using four schedulers $Sched_1, \ldots, Sched_4$. For the sake of simplicity, we consider each action of the components as a singleton interaction of the system such that $Act = \{Comp_i.Action_1, Comp_i.Action_2 \mid i \in [1,4]\}$. Each scheduler manages a subset of Act. We define various partitioning of the interactions to obtain the following settings:*

1. *Each scheduler is dedicated to manage the actions of only one component, such that actions $Comp_i.Action_1$ and $Comp_i.Action_2$ are managed by Scheduler $Sched_i$ for $i \in [1,4]$. In this setting, no component has shared its actions to more that one scheduler.*

2. *Considering the previous setting with the only difference that action $Comp_1.Action_2$ by scheduler $Sched_2$. In this setting, component $Comp_1$ is a shared component.*

Table 2: Results of lattice construction w.r.t different settings of Example 19

shared component	lattice nodes	removed nodes	paths
0	175	81	10,681,263
1	88	72	1,616,719
2	56	60	572,847
3	34	52	316,035
4	18	47	251.177

3. *Considering the previous setting with the only difference that action* $Comp_2.Action_2$ *by scheduler* $Sched_3$*. In this setting, components* $Comp_1$ *and* $Comp_2$ *are shared component.*

4. *Considering the previous setting with the only difference that action* $Comp_3.Action_2$ *by scheduler* $Sched_4$*. In this setting, components* $Comp_1$*,* $Comp_2$ *and* $Comp_3$ *are shared component.*

5. *Considering the previous setting with the only difference that action* $Comp_4.Action_2$ *by scheduler* $Sched_1$*. In this setting, all the components are shared component.*

Since the components are designed to be involved only in three actions, the number of generated action/update events is equal in different settings (24 events in total), no matter which scheduler manages which action, and the only differences of events obtained through those setting are the vector clock of the action events and the sender of action/update events. Table 2 and Figure 12 represent the results with respect to the above-mentioned settings. Columns in Table 2 have the following meaning:

- *Column shared component indicates the number of shared components in each setting.*
- *Column lattice nodes shows the number of the nodes of the constructed lattice in each setting.*
- *Column removed nodes indicates the number of removed nodes in the lattice using the optimization algorithm.*
- *Column path indicates the number of paths of the constructed computation lattice.*

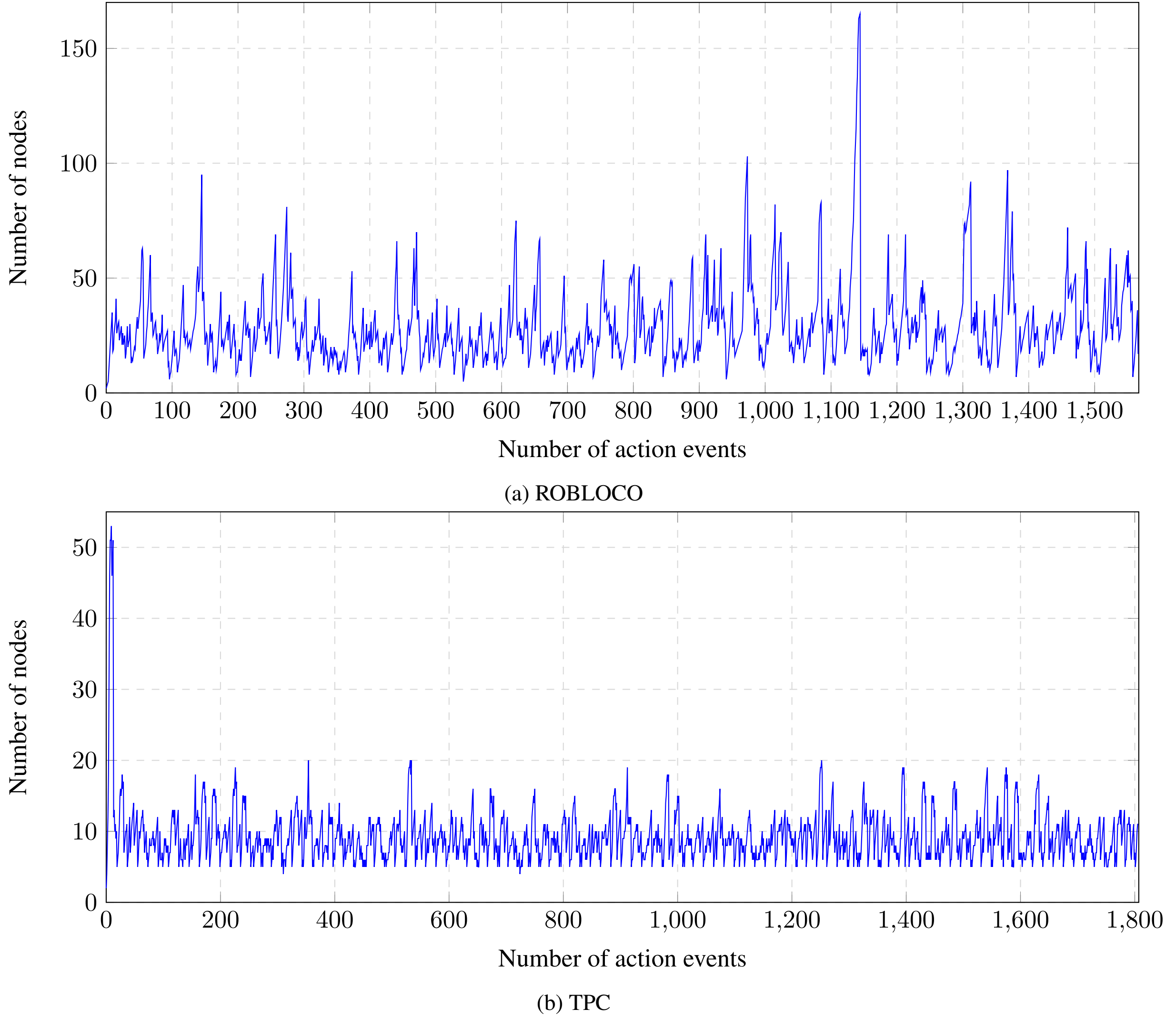
Considering the first setting where there is no shared component in the system, results a set of independent action events where none of the two action events from two different schedulers are causally related, so that we construct a complete (maximal) computation lattice in order to cover all the compatible global traces. The size of constructed lattice as well as the number of paths of the lattice is decreased by considering more shared components (see Figure 12a and Figure 12b).

8.2 Results and Conclusion

Table 3 and Figure 13 present the results checking specifications *deadlock freedom* on ROBLOCO and *protocol correctness* on TPC. The columns of the table have the following meanings:

- Column $|\varphi|$ shows the size of the monitored LTL formula. Note, the size of formulas are measured in terms of the operators entailment inside it, e.g., $\mathbf{G}(a \wedge b) \vee \mathbf{X}c$ is of size 2.
- Column *observed event* indicates the number of action/update events sent by the controllers of the schedulers.
- Column *lattice size* reports the size of constructed lattice using optimization algorithm is used vs. the size of constructed lattice when non-optimized algorithm is used.
- Column *frontier node VC* indicated the vector clock associated to the frontier node of the constructed lattice.

Figures 13a, 13b show how the size of constructed lattice varies in two systems as they evolve. Having shared components in system is not the only reason to have a small lattice size, what is more important is how often



(a) ROBLOCO

(b) TPC

Figure 13: Optimization algorithm effect on the size of the constructed lattice

the shared components are used as a part of executed interactions. The more execution with shared components results the more dependencies in the generated events and thus the smaller lattice size.

In ROBLOCO system, after receiving 3463 events, the size of the obtained computation lattice is 17, whereas the size of non-optimized lattice is 10602 which is a quite large in terms of storage space and iteration process. It shows how efficient our optimization algorithm minimize and optimize the monitoring process. Figures 13a shows how the size of constructed lattice varies by the time the ROBLOCO systems evolves. Although what we need in the constructed computation lattice as the verdicts of the monitor output is only stored in the frontier node, but the rest of the nodes are necessary to be kept at runtime in order to extend the lattice in case of reception of new events.

In TPC system we also obtained a very small size of the lattice after the reception of 4709 events. As it is shown in Table 3 the size and complexity of the LTL property does not change the structure of the constructed lattice, it only effects on the progression process. The frontier-node vector clock shows that how many interactions have been executed by each scheduler at the end of the system run.

Our monitoring algorithm implemented in RVDIST provide a lightweight tool to runtime monitor the behavior of a distributed CBS. RVDIST keeps the size of the lattice as small as possible even for a long run.

Table 3: Results of monitoring ROBLOCO and TPC with RVDIST

System	property	$\lvert\varphi\rvert$	# observed events	# lattice size		frontier node VC
				optimized	not-optimized	
ROBLOCO	φ_1	3	3463	17	10602	(730,352,485)
TPC	φ_2	10	4709	11	2731	(402,402,402,601)
	φ_3	26				

9 Related Work

A close work to the approach presented in this paper has been exposed in [4]. In this setting, multiple components in a system each observe a subset of some global event trace. Given an LTL property φ, their goal is to create sound formula derived from φ that can be monitored on each local trace, while minimizing inter-component communication. Similar to our approach, the monitor synthesis is based on the internal structure of the monitored system and the projection of the global trace upon each component is well-defined and known in advance. Moreover, all components consume events from the trace synchronously. Compare to our setting, we target a distributed component system with asynchronous executions. Hence, instead of having a global trace at runtime, we are dealing with a set of possible global traces which possibly could happen during the run of the system.

In [7], Cooper and Marzullo present three algorithms for detecting global predicates based on the construction of the lattice associated with a distributed execution. The first algorithm determined that the predicate was *possibly* true at some point in the past; the second algorithm determines that the predicate was *definitely* true in the past; while the third algorithm establishes that the predicate is *currently* true, but to do so it may delay the execution of certain processes.

In [8], Diehl, Jard and Rampon present basic algorithm for trace checking of distributed programs by building the lattice of all reachable states of the distributed system under test, based on the on-the-fly observation of the partial order of message causality. Compare to our approach, in our distributed setting schedulers don't communicate directly by sending-receiving messages. Moreover, no monitor has been proposed in [8] for the purpose of verification whereas in our algorithm we synthesize a runtime monitor which evaluate on-the-fly the behavior of the system based on the reconstructed computation lattice of partial-states.

In [24], Sen and Vardhan design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). The distributed monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. That is, if processes rarely communicate, then monitors exchange very little information and, hence, some violations of properties may remain undetected. In that paper, a tool called DIANA (distributed analysis) introduce in order to implement the proposed monitoring method. The main noteworthy difference between [24] and our work is that we evaluate the behavior of the distributed system based on all of the possible global traces of the distributed system.

In [13], Massart and Meuter define an online monitoring method which collect the trace and checks on the fly that is satisfies a requirement, given by any LTL property on finite sequence. Their method explores the possible configurations symbolically, as it handles sets of configurations. Our approach mainly differs from [13] in that we target distributed CBSs with multi-party interactions where the execution traces are defined over the set of the partial states of the system.

In [20], Scheffel and Schmitz studied runtime verification of distributed asynchronous systems against Distributed Temporal Logic (DTL) properties. DTL combines the three-valued Linear Temporal Logic (LTL_3) with past-time Distributed Temporal Logic (ptDTL). In that paper, a distributed system is modeled as n agents and each agent has a local monitor. These monitors work together to check a property, but they only communicate by adding some data to the messages already sent by the agents. They can not force their agent to send a message or even communicate on their own.

In [16], a decentralized algorithm for runtime verification of distributed programs is proposed. Proposed algorithm conducts runtime verification for the 3-valued semantics of the linear temporal logic (LTL_3). In that paper, they adapt the distributed computation slicing algorithm for distributed online detection of conjunctive predicates, and also the lattice-theoretic technique is adapted for detecting global-state predicates at run time.

In [23], Sen and Garg use a temporal logic, CTL, for specifying properties of distributed computation and interpret it on a finite lattice of global states and check that a predicate is satisfied for an observed single execution trace of the program. Compare to our approach, we deal with a set of events at runtime generated by the schedulers which results in a infinite lattice of partial-states. Although the computation lattice in our method is made based

of the observed partial states, we could check the satisfaction of temporal predicates defined over the global states of the system, which mean that we could monitor the system even if the global state of the is not defined.

In [21], Sen and Garg used computation slicing for offline predicate detection in the subset of CTL with the following three properties; i) temporal operators, ii) atomic propositions are regular predicates and iii) negation operator has been pushed onto atomic propositions. They called this logic *Regular CTL plus* (RCTL+), where plus denotes that the disjunction and negation operators are included in the logic. In that paper, authors gave the formal definition of RCTL+ which uses regular predicates as atomic propositions and implemented their predicate detection algorithms, which use computation slicing, in a prototype tool called Partial Order Trace Analyzer (POTA).

In [22], Sen and Gerg present temporal slicing centralized online algorithm with respect to properties in temporal logic RCTL+.

10 Conclusions and Future Work

We draw conclusions and outline avenues for future work.

10.1 Conclusions

In this paper, we have presented a technique to enable runtime verification on a distributed component-based system with multi-party interactions. In our setting, global joint actions are partitioned among a set of distributed scheduler. Each scheduler is in charge of execution of the dedicated subset of global joint actions. execution of each global actions, triggers the set of actions of corresponding component involved in the joint action. Our proposed technique consists in (i) transformation of the given distributed system to generate locally observed events by each distributed scheduler, (ii) synthesizing a centralized observer which collects the local events of all schedulers (iii) introducing an algorithm to reconstruct on-the-fly the set of possible ordering among the received events which forms a computation lattice, (iv) augmentation of the reconstructed computation lattice with a verification method in the sense that the observer plays the role of a runtime monitor while it is building the lattice. We showed that the set of paths of the constructed lattice represents the set of compatible traces, such that each of them could have occurred as the actual run of the system. The experimental results show that even for a long run of a system, that is having many generated events, using the optimization algorithm keeps the size of the lattice minimal. Moreover, the set of formulas attached to the frontier node of the constructed lattice represents the evaluation of all the compatible traces with respect to the given LTL formula.

10.2 Future Work

Several research perspectives can be considered.

A first direction is to distribute the runtime monitor, such that the satisfaction or violation of specifications can be detected by local monitors alone. By distributing the monitors we indeed decrease the load of monitoring process on a single entity.

Another possible direction is to extend the proposed framework for timed components and timed specifications as presented in [2, 27].

References

[1] Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Formal Techniques for Networked and Distributed Systems - FORTE, 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings. pp. 116–133 (2008) 1

[2] Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India. pp. 3–12 (2006) 10.2

[3] Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010) 1, 6

[4] Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. pp. 85–100 (2012) 6, 5, 9

[5] Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: International Conference on Concurrency Theory. pp. 508–522. Springer (2008) 1

[6] Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3(1), 63–75 (1985) 2

[7] Cooper, R., Marzullo, K.: Consistent detection of global predicates. ACM (1991) 9

[8] Diehl, C., Jard, C., Rampon, J.X.: Reachability analysis on distributed executions. Springer (1993) 9

[9] Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress proper6ties. In: Proceedings of the 9th International Workshop on Runtime Verification (RV 2009), Selected Papers. pp. 40–59. Springer (2009) 1

[10] Gray, J.N.: Notes on data base operating systems. In: Operating Systems, pp. 393–481. Springer (1978) 8.1.2

[11] Gray, J., Lamport, L.: Consensus on transaction commit. ACM Transactions on Database Systems (TODS) 31(1), 133–160 (2006) 8.1.2

[12] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978) 2, 2

[13] Massart, T., Meuter, C.: Efficient online monitoring of LTL properties for asynchronous distributed systems. Université Libre de Bruxelles, Tech. Rep (2006) 9

[14] Mattern, F.: Virtual time and global states of distributed systems. Parallel and Distributed Algorithms 1(23), 215–226 (1989) 2

[15] Milner, R.: Communication and concurrency. Prentice Hall International (UK) Ltd. (1995) 2

[16] Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015. pp. 494–503 (2015) 9

[17] Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., Combaz, J.: Monitoring multi-threaded component-based systems. In: International Conference on Integrated Formal Methods. pp. 141–159. Springer (2016) 5.5, 5.5

[18] Pnueli, A.: The temporal logic of programs. In: SFCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society (1977) 1, 2

[19] Runtime Verification: http://www.runtime-verification.org (2001-2016) 1

[20] Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on. pp. 52–61. IEEE (2014) 9

[21] Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: Principles of Distributed Systems, pp. 171–183. Springer (2004) 9

[22] Sen, A., Garg, V.K.: Formal verification of simulation traces using computation slicing. IEEE Trans. Computers 56(4), 511–527 (2007) 9

[23] Sen, A., Garg, V.K.: Detecting temporal logic predicates on the happened-before model. In: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM. pp. 8–pp. IEEE (2001) 9

[24] Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of the 26th International Conference on Software Engineering. pp. 418–427. IEEE Computer Society (2004) 9

[25] Tanenbaum, A.S., van Steen, M.: Fault tolerance. Distributed Systems: Principles and Paradigms, Upper Saddle River, New Jersey, Prentice-Hall, Inc pp. 361–412 (2002) 8.1.2

[26] Tretmans, J.: A formal approach to conformance testing. In: Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems. pp. 257–276 (1993) 3

[27] Triki, A., Combaz, J., Bensalem, S.: Optimized distributed implementation of timed component-based systems. In: Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on. pp. 30–35. IEEE (2015) 10.2