Monitoring Distributed Component-Based Systems

Yliès Falcone[1], Hosein Nazarpour[2], Saddek Bensalem[2], and Marius Bozga[2]

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38000 Grenoble, France
FirstName.LastName@univ-grenoble-alpes.fr

Abstract. We monitor asynchronous distributed component-based systems with multi-party interactions. We consider independent components whose interactions are managed by several distributed schedulers. In this context, neither a global state nor the total ordering of the executions of the system is available at runtime. We instrument the system to retrieve local events from the local traces of the schedulers. Local events are sent to a global observer which reconstructs on-the-fly the set of global traces that are compatible with the local traces, in a concurrency-preserving fashion. The set of compatible global traces is represented in the form of an original lattice over partial states, such that each path of the lattice corresponds to a possible execution of the system.

1 Introduction

Component-based design consists in constructing complex systems using a set of predefined components. Each component is 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 with multiparty interactions (CBS) is defined according to the behavior of each component as well as their interactions. Each interaction is a set of simultaneously executed actions of the existing components [9]. In the distributed setting, for efficiency reasons, the execution of the interactions is distributed among several independent schedulers. 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 schedulers and the components [3].

Verification techniques can ensure the correctness of a distributed CBS. Runtime Verification (cf. [30,1,18]) consists in verifying the executions of the system against the desired properties. We consider properties referring to the global states of the system which can not be projected nor checked on individual components. In the following, we point out the problems that one encounters when monitoring distributed CBSs. We use neither a global clock nor a shared memory. This makes the execution of the system more dynamic and parallel by avoiding synchronization to take global snapshots, which would go against the distribution of the verified system. However, 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 makes the actual execution trace 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 behavior.

Our main contribution is an approach for the monitoring of distributed CBSs w.r.t. specifications referring to the global states of the system. First, we define a *monitoring hypothesis* that permits to rely on an abstract semantic model of distributed CBSs that encompasses a variety of distributed (component-based) systems. Our model only relies on the semantics of CBS, given in terms of Labeled Transition Systems (LTSs), thus it is not bound to any CBS framework. In a distributed CBS, due to the parallel executions in different schedulers (i) events (i.e., actions changing the state of the system) are not totally ordered, and (ii) the actual execution trace of a distributed system can not be obtained. Although each scheduler is aware of its local events, to evaluate the global behavior, it is necessary to find a set of possible ordering of the events of all schedulers, that is, the set of compatible execution traces. In our setting, schedulers do not communicate together but 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 account for existing causalities among events, we (i) employ *vector clocks* to define the ordering 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 algorithm that executes on a *global observer* to reconstruct a set of compatible execution traces that could possibly happen in the system with respect to the received events. We represent the set of compatible traces using a computation lattice tailored for CBSs. Such a computation lattice consists of a set of partially connected nodes. Created nodes are partial states and become global states during monitoring. Any path of the lattice projected on a scheduler represents the corresponding local partial trace according to that scheduler (*soundness*). All possible global traces are recorded (*completeness*).

An extended version of this paper with more details and proofs is available in [29].

2 Preliminaries and Notations

Sequences For a finite set E, a sequence s containing elements of E is formally defined by a total function $s : I \to E$ 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 i^{th} element of s and $s(i \cdots j)$ as the factor of s from the i^{th} to the j^{th} element; and $s(i \cdots j) = \epsilon$, if $i > j$. We define function $\mathrm{last} : E^+ \to E$ as $\mathrm{last}(e) = s(\mathrm{length}(s))$. 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 s from index i onwards. An n-tuple is an ordered list of n elements, where $n \in \mathbb{N}$. The i^{th} element of tuple t is denoted by $t[i]$.

Labeled transition systems (LTS) Labeled Transition Systems (LTSs) are used to define the semantics of CBSs. An LTS 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 say that a is *enabled* in q. 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'.

Vector Clock Mattern and Fidge's vector clocks [20,27] are a more powerful extension of Lamport's scalar logical clocks [23], i.e., strongly consistent with the ordering of events. In a distributed system with a set of schedulers $\{S_1, \ldots, S_m\}$, $VC = \{(c_1, \ldots, c_m) \mid \forall j \in [1\,..\,m]\,.\, c_j \in \mathbb{N}\}$ 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 $S_k \in \{S_1, \ldots, S_m\}$. A unique vector clock is associated each event in the system ([27], Sec. 7). For two vector clocks vc_1 and vc_2, $\max(vc_1, vc_2)$ is a vector clock vc_3 such that $\forall k \in [1\,..\,m]\,.\, vc_3[k] = \max(vc_1[k], vc_2[k])$. 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 [23] Relation $\rightarrowtail$ on the set of system events 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 [27] A computation lattice is represented as a directed graph with m (i.e., number of schedulers executed in distributed manner) axes. Each axis is dedicated to the state evolution of a scheduler. A computation lattice expresses all the possible traces. 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 happened-before relation among the nodes.

3 Distributed CBS

We describe our assumptions on CBSs by providing them with a general semantics. The exact model and the system behavior are unknown. The architecture, the behaviors of the components and the schedulers, and the association between schedulers and components can be obtained by several techniques such as the ones in [10,7]. Our monitoring framework is independent from the technique used to obtain the system and its implementation. Inspiring from conformance-testing theory [32], we refer to this as the *monitoring hypothesis*.

3.1 Semantics

The system is composed of *components* in a set $\mathbf{B} = \{B_1, \ldots, B_{|\mathbf{B}|}\}$ and *schedulers* in a set $\mathbf{S} = \{S_1, \ldots, S_{|\mathbf{S}|}\}$. Each component B_i is endowed with a set of actions Act_i. Joint actions, 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: $\forall a \in Int\,.|a \cap Act_i| \leq 1$. In addition, each component B_i has internal actions modeled as a unique action β_i. Schedulers coordinate the execution of interactions and ensure that each multi-party interaction is jointly executed (Def. 2).

We assume some functions from the system architecture.

- Function $\mathrm{inv} : Int \to 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the components involved in an interaction. Moreover, we extend function inv to internal actions by setting $\mathrm{inv}(\beta_i) = i$, for any $\beta_i \in \{\beta_1, \ldots, \beta_{|\mathbf{B}|}\}$. Interaction $a \in Int$ is a joint action if and only if $|\mathrm{inv}(a)| \geq 2$.
- Function $\mathrm{mng} : Int \to \mathbf{S}$ indicates the scheduler managing an interaction: for an interaction $a \in Int$ $\mathrm{mng}(a) = S_j$ if a is managed by scheduler S_j.
- Function $\mathrm{scp} : \mathbf{S} \to 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the set of components in the scope of a scheduler s.t. $\forall j \in [1\,..|\mathbf{S}|]\,.\,\mathrm{scp}(S_j) = \bigcup_{a' \in \{a \in Int \mid \mathrm{mng}(a) = S_j\}} \mathrm{inv}(a')$.

We describe the behavior of components, schedulers, and their composition.

Definition 1 (Behavior of a component). *The behavior of a component B is an LTS $(Q_B, Act_B \cup \{\beta_B\}, \to_B)$ s.t.:*

- *Q_B is the set of states which has a partition $\{Q_B^{\mathrm{r}}, Q_B^{\mathrm{b}}\}$, where Q_B^{r} (resp. Q_B^{b}) is the so-called set of* ready *(resp.* busy*) states,*
- *Act_B is the set of actions, and β_B is the internal action,*
- *$\to_B \subseteq (Q_B^{\mathrm{r}} \times Act_B \times Q_B^{\mathrm{b}}) \cup (Q_B^{\mathrm{b}} \times \{\beta_B\} \times Q_B^{\mathrm{r}})$ is the set of transitions.*

The set of ready (resp. busy) states Q_B^{r} (resp. Q_B^{b}) is the set of states s.t. 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 s.t. $\beta_B \notin Act_B$ 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. Note that busy states permit the modelling of distributed (decentralized) execution of components: after an interaction, components stay in busy states until the internal computation related to the interaction terminates (after which they get ready for the next interaction and so on). The state of components is only modified by their internal actions; other actions are dedicated to synchronisation.

We assume that each component $B_i \in \mathbf{B}$ is defined by the LTS $(Q_{B_i}, Act_{B_i} \cup \{\beta_{B_i}\}, \to_{B_i})$ where Q_{B_i} has a partition $\{Q_{B_i}^{\mathrm{r}}, Q_{B_i}^{\mathrm{b}}\}$ of ready and busy states.

Definition 2 (Behavior of a scheduler). *The behavior of a scheduler S is an LTS $(Q_S, Act_S, \to_S)$ s.t.:*

- *Q_S is the set of states,*

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

$Act^{\gamma}_S \subseteq Int$ 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 $Act_{S_j} = Act^{\gamma}_{S_j} \cup Act^{\beta}_{S_j}$; as per Def. 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. Since one scheduler is associated with each interaction, 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, by construction, schedulers are always ready to receive such a state update.

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

Definition 3 (Shared component). $\mathbf{B}_{\mathrm{s}} = \{B \in \mathbf{B} \mid |\{S \in \mathbf{S} \mid B \in \mathrm{scp}(S)| \geq 2\}\}$.

A shared component is in the scope of more than one scheduler. Thus, the execution of its actions are managed by more than one scheduler. The global execution of the system can be described as the parallel execution of interactions managed by the schedulers.

Definition 4 (Global behavior). *The system behavior 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,*
- $GAct \subseteq 2^{Int \cup \bigcup_{i=1}^{|\mathbf{B}|}\{\beta_i\}} \setminus \{\emptyset\}$ *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 by 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: $\forall a \in \alpha \,.\, (\exists S_j \in \mathbf{S} \,.\, \mathrm{mng}(a) = S_j) \implies \Big(q_{s_j} \xrightarrow{a}_{S_j} q'_{s_j} \wedge \forall B_i \in \mathrm{inv}(a) \,.\, q_i \xrightarrow{a \cap Act_i}_{B_i} q'_i\Big)$,

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

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

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

where functions inv *and* mng *are extended to sets of interactions and internal actions.*

The system components execute according to the schedulers decisions.

- C_1 states that a component performs at most one execution step at a time. Executed global actions ($\alpha \cup \beta$) contains at most one interaction involving each component.
- *Condition* C_2 states that whenever an interaction a managed by scheduler S_j is executed, a is enabled in S_j and the corresponding action (in $a \cap Act_i$) is enabled in each component involved in this interaction.
- *Condition* C_3 states that internal actions are executed whenever they are enabled in the corresponding components. Schedulers are aware of internal actions of components in their scope. This results in transferring the updated state 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.

Remark 2. The operational description of a CBS is usually more detailed. 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, i.e., 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 execution traces, we assume that these are correct w.r.t. the conflicts and priorities. Therefore, defining the other modules is out of our scope. Moreover, schedulers could interact together as part of some coordination protocol, but our model does not account for it.

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

3.2 Traces

Running the system produces a trace. Intuitively, a 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 system states to states of the components.

We consider a CBS consisting of a set $\mathbf{B}$ of components (as per Def. 1) and a set $\mathbf{S}$ of schedulers (as per Def. 2) with the global behavior as per Def. 4.

Definition 6 (Trace). *A trace is a sequence* $(q_1^0, \ldots, q_{|\mathbf{B}|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \ldots, q_{|\mathbf{B}|}^1) \cdots (q_1^k, \ldots, q_{|\mathbf{B}|}^k) \cdots$*, s.t.* $q_1^0, \ldots, q_{|\mathbf{B}|}^0$ *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 system and scheduler states are discarded.*

Since a trace t has partial states where at least one component is busy with its internal computation, t is referred to as a *partial trace*. Although the partial 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 in the system

would require all components to synchronize, and would defeat the purpose of building a distributed system. Instead, we shall instrument the system to observe the sequence of states through schedulers.

In the sequel, we consider a partial 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 Def. 6. Each scheduler $S_j \in \mathbf{S}$, observes a local partial trace $s_j(t)$ which consists in the sequence of state-evolutions of the components it manages.

Definition 7 (Observable local partial-trace). *The local partial-trace $s_j(t)$ observed by scheduler S_j is defined on the partial 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*
- $s_j\left(t \cdot (\alpha \cup \beta) \cdot q\right) = \begin{cases} t & \text{if } S_j \notin \mathrm{mng}(\alpha) \wedge (\mathrm{inv}(\beta) \cap \mathrm{scp}(S_j) = \emptyset) \\ t \cdot \gamma \cdot q' & \text{otherwise} \end{cases}$

  where
  - $q = \left(q_1, \ldots, q_{|\mathbf{B}|}\right)$,
  - $\gamma = \left(\alpha \cap \{a \in \mathit{Int} \mid \mathrm{mng}(a) = S_j\}\right) \cup \left(\beta \cap \{\beta_i \mid B_i \in \mathrm{scp}(S_j)\}\right)$
  - $q' = (q'_1, \ldots, q'_{|\mathbf{B}|})$ *with* $q'_i = \begin{cases} \mathrm{last}(s_j(t))[i] & \text{if } B_i \in \overline{\mathrm{inv}(\gamma)} \cap \mathrm{scp}(S_j), \\ q_i & \text{if } B_i \in \mathrm{inv}(\gamma) \cap \mathrm{scp}(S_j), \\ ? & \text{otherwise } (B_i \notin \mathrm{scp}(S_j)). \end{cases}$

We assume that the initial system state 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., $S_j \in \mathrm{mng}(a)$). 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. The state observed after an observable interaction or internal action consists of the states of components in the scope of S_j, i.e., a state $(q_1, \ldots, q_{|\mathbf{B}|})$ where q_i is the new state of component B_i if $B_i \in \mathrm{scp}(S_j)$ and ? otherwise.

4 Efficient Construction of the Computation Lattice

We define how a global observer constructs on-the-fly a computation lattice representing the possible global traces compatible with the local partial-traces observable by schedulers. Since schedulers do not interact directly, 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 s.t. a shared component $B_i \in \mathbf{B}_\mathrm{s}$ is involved in a and b, i.e., $B_i \in \mathrm{inv}(a) \cap \mathrm{inv}(b)$, 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 do not belong to the system). To find out the actual ordering and obtain the local partial-traces, one needs instrumenting the system by adding controllers to the schedulers and to the shared components. Each time a scheduler executes an interaction, the involved components are notified by the scheduler to execute their corresponding actions. Moreover, the controller of the scheduler updates its local clock and notifies the controller of the shared components involved in the interaction by sending its vector clock. Whenever a shared component executes its internal action β, schedulers with the shared component in their scope are

notified by receiving the updated state. Moreover, the vector clock stored in the controller of the shared component is sent to the controller of the associated schedulers. Consequently, schedulers with a shared component in their scope exchange their vector clocks through the shared component. Such an instrumentation is described in [29] but omitted for space reasons.

Intuitively, for scheduler S_j, the execution of an interaction (labeled by a vector clock), or notification by the internal action of a component which the execution of its latest action has been managed by scheduler S_j, is defined as an *event* of scheduler S_j. For a partial trace t, the sequence of events of scheduler S_j is denoted by $\mathrm{event}(s_j(t))$.

4.1 Computation Lattice

The computation lattice is represented implicitly using vector clocks. The construction 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 referred to as *update events*. (Recall that internal actions carry the state of the component that 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 $E_{\mathrm{a}} = Int \times VC$ with VC the set of vector clocks, 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 $E = E_{\beta} \cup E_{\mathrm{a}}$. Since the received events are not totally ordered (because of 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 adapt the notion of computation lattice to CBSs.

Definition 8 (Computation lattice). *A computation lattice $\mathcal{L}$ is a tuple $(N, Int, \rightarrowtail\!\!\!\!\!\!\rightarrow)$, where:*

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

where $\rightarrowtail\!\!\!\!\!\!\rightarrow$ is the extended happened-before relation, which is labeled by the set of multi-party interactions and $\eta.state$ refers 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 system state and a vector clock. A system state consists of the states of all components. The state of a component is either a ready state or a busy state (as per Def. 1). We represent a busy state of component B_i, 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. A computation lattice $\mathcal{L}$ initially consists of an initial node $init_{\mathcal{L}} = (init, (0, \ldots, 0))$, where $init$ is the initial state of the system and $(0, \ldots, 0)$ is a vector clock where all the clocks associated with the schedulers are zero. The set

of nodes of $\mathcal{L}$ is denoted by $\mathcal{L}.nodes$, and for a node $\eta = (q, vc) \in \mathcal{L}.nodes$, $\eta.state$ denotes q and $\eta.clock$ denotes vc. If (i) the event of node η happened before the events of node η', that is $\eta'.clock > \eta.clock$ and $\eta \rightarrowtail \eta'$, and (ii) the states of η and η' follow the global behavior of the system (Def. 4) in the sense that the execution of an interaction $a \in Int$ from the state of η brings the system to the state of η', that is $\eta.state \xrightarrow{a} \eta'.state$, 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 $\eta.clock > \eta'.clock$ nor $\eta'.clock > \eta.clock$. For two concurrent nodes η and η' if there exists a node η'' s.t. $\eta'' \rightarrowtail\!\!\!\!\twoheadrightarrow \eta$ and $\eta'' \rightarrowtail\!\!\!\!\twoheadrightarrow \eta'$, then node η'' is said to be the *meet* of η and η' denoted by $\mathrm{meet}(\eta, \eta', \mathcal{L}) = \eta''$.

4.2 Intermediate Operations

We consider a computation lattice $\mathcal{L}$ (Def. 8). A received event either modifies $\mathcal{L}$ or is kept for later in a queue. Action events extend $\mathcal{L}$ using operator extend (Def. 9), and update events update the existing nodes of $\mathcal{L}$ by adding the missing state information into them using operator update (Def. 12). By extending the lattice with new nodes, one needs to further complete the lattice by computing the joins of created nodes (Def. 11) with existing ones to complete the set of possible global states and 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 and an action event and outputs a new node.

Definition 9 (Node extension). *Given a node $\eta = (q, vc) \in Q^l \times VC$ and an action event $e = (a, vc') \in E_{\mathrm{a}}$, function* $\mathrm{extend} : (Q^l \times VC) \times E_{\mathrm{a}} \to Q^l \times VC$ *is defined as follows:*

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

with $\forall i \in [1\,..\,|\mathbf{B}|]\,.\, q'[i] = \begin{cases} \perp_i^k & \text{if } B_i \in \mathrm{inv}(a), \text{where } k = \mathrm{mng}(a).index, \\ q[i] & \text{otherwise.} \end{cases}$

Node η is said to be *extendable* by event e if $\mathrm{extend}(\eta, e)$ is defined. 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 $S_j = \mathrm{mng}(a)$, could execute interaction a. State $\perp_i^k$ 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 $\mathcal{L}$ is extendable by action event e if there exists a node $\eta \in \mathcal{L}.nodes$ s.t. $\mathrm{extend}(\eta, e)$ 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 action 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). We define a relation between two vector clocks to distinguish the concurrent execution of two interactions s.t. both could happen from a specific global system state.

Definition 10 (Relation $\mathcal{J}_{\mathcal{L}}$). $\mathcal{J}_{\mathcal{L}} = \{(vc, vc') \in VC \times VC \mid \exists!\, k \in [1\,..\,|\mathbf{S}|]\,.\, vc[k] = vc'[k] + 1 \wedge \exists!\, l \in [1\,..\,|\mathbf{S}|]\,.\, vc'[l] = vc[l] + 1 \wedge \forall j \in [1\,..\,|\mathbf{S}|] \setminus \{k, l\}\,.\, vc[j] = vc'[j]\}$.

For two vector clocks vc and vc' to be in $\mathcal{J}_{\mathcal{L}}$, they should agree on all but two clock values related to two schedulers of indexes k and l. On one index, 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 η and η' are associated with two concurrent events (caused by the execution of two interactions managed by different schedulers) that both could happen from a unique global system state, which is the meet of η and η' (see Property 2).

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

The join of two nodes is defined as follows.

Definition 11 (Join node). *For two nodes $\eta, \eta' \in \mathcal{L}.nodes$ s.t. $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$, the join of η and η', denoted by $\mathrm{join}(\eta, \eta', \mathcal{L}) = \eta''$, is the node defined as follows:*

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

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 join 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 state different from their meet (components can not be both involved in two concurrent executions). The join 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 $\mathrm{join}(\eta, \eta', \mathcal{L}) = \mathrm{join}(\eta', \eta, \mathcal{L})$, because join is defined for nodes whose clocks are in relation $\mathcal{J}_{\mathcal{L}}$.

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

Definition 12 (Node update). *Given 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 $k \in [1\,..|\mathbf{S}|]$, function* $\mathrm{update} : (Q^l \times VC) \times E_\beta \to Q^l \times VC$ *is defined as follows:*

$$\mathrm{update}(\eta, e) = ((q_1, \ldots, q_{i-1}, q_i'', q_{i+1}, \ldots, q_{|\mathbf{B}|}), vc), \text{ with } q_i'' = \begin{cases} q_i' & \text{if } q_i = \bot_i^k, \\ q_i & \text{otherwise.} \end{cases}$$

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 partial 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 it means that a busy state resulting of the execution of an action managed by scheduler S_k can only be replaced by a ready state sent by S_k. Updating node η does not modify vc.

Buffering events The reception of an action or 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 lattice to use events in a specific order. Events

not in the desired order must be kept in a waiting queue to be used later. For example, such a situation occurs when receiving action event e s.t. function extend is not defined over e and none of the existing nodes of the lattice. 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 s.t. 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 13 (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 $e = e_z \in \{e_1, e_2, \ldots, e_r\}$. Moreover, events in the queue are picked up in the same order as they have been stored in the queue (FIFO queue).*

4.3 Algorithms for Constructing the Computation Lattice

We define an algorithm based on the above definitions to construct the computation lattice based on the received events. The algorithm consists of a main procedure (see Algorithm 1) and several sub-procedures. The algorithm defines and uses a lattice (Def. 8, global variable $\mathcal{L}$) and a queue (Def. 13, global variable κ).

For an action event $e \in E_{\mathrm{a}}$ 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.

After the reception of each event e from a controller of a scheduler, $\textsc{Make}(e) = \textsc{Make}(e, \texttt{false})$ is called. In the sequel, we describe each procedure.

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-queue)
2:     if e ∈ E_a then
3:         ACTIONEVENT(e, from-queue)
4:     else if e ∈ E_β then
5:         UPDATEEVENT(e, from-queue)
6:     end if
7: end procedure
```

MAKE (Algorithm 1) Procedure MAKE takes two parameters as input: an event e and a boolean variable $from\text{-}queue$. Parameters e and $from\text{-}queue$ vary based on the type of event e. Boolean variable $from\text{-}queue$ 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. MAKE updates the global variables.

ACTIONEVENT (Algorithm 2) Procedure ACTIONEVENT takes as input an action event e and a boolean parameter. Procedure ACTIONEVENT modifies global variables $\mathcal{L}$ and κ. ACTIONEVENT has a local boolean variable named $lattice\text{-}extend$, 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, ACTIONEVENT checks if there exists a node η in $\mathcal{L}.nodes$ s.t. function extend is defined over event e and node η (Def. 9). If such a node η is found, ACTIONEVENT creates the new node $\mathrm{extend}(\eta, e)$, 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) join nodes. To this end, in Line 15 procedure JOINS is called to evaluate the current lattice and create the join nodes. For optimization purposes, REMOVEEXTRANODES is then called to eliminate unnecessary nodes that represent past system states. After making the join nodes and (possibly) reducing the size, if the input action event is not picked from the queue, ACTIONEVENT invokes procedure CHECKQUEUE in Line 18, otherwise it terminates.

Algorithm 2 ACTIONEVENT

```
1:  procedure ACTIONEVENT(e, from-queue)
2:      lattice-extend ← false
3:      for all η ∈ L.nodes do
4:          if ∃η' ∈ Q^l × VC . η' = extend(η, e) then
5:              L.nodes ← L.nodes ∪ {η'}
6:              MODIFYQUEUE(e, from-queue, true)
7:              lattice-extend ← true
8:              break
9:          end if
10:     end for
11:     if ¬ lattice-extend then
12:         MODIFYQUEUE(e, from-queue, false)
13:         return
14:     end if
15:     JOINS( )
16:     REMOVEEXTRANODES()
17:     if ¬ from-queue then
18:         CHECKQUEUE( )
19:     end if
20: end procedure
```

UPDATEEVENT (Algorithm 3) Recall that an update event e contains the state update of some component B_i with $i \in [1, n]$ ($e.index = i$). UPDATEEVENT takes as input an update event e and a boolean value associated with parameter *from-queue*. 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 s.t. 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 concerns component B_i, UPDATEEVENT updates all the nodes of the lattice (Lines 8-10) according to Def. 12. Finally, the input update event is removed from the queue if it is picked from the queue, using MODIFYQUEUE.

MODIFYQUEUE takes as input an event e and boolean variables *from-queue* and *event-is-used*. Procedure MODIFYQUEUE adds (*resp.* removes) event e to (*resp.* from) queue κ. If event e is picked up from the queue (i.e., *from-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. 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.

JOINS extends $\mathcal{L}$ in such a way that all the possible joins have been created. First, procedure JCOMPUTE is invoked to compute relation $\mathcal{J}_{\mathcal{L}}$ (Def. 10) among the existing nodes of the lattice and then creates the join nodes and adds them to the set of the nodes. Then, after the creation of the join of two nodes η and η', $(\eta.clock, \eta'.clock)$ is removed from $\mathcal{J}_{\mathcal{L}}$. It is necessary to compute $\mathcal{J}_{\mathcal{L}}$ again after the creation of joins, because new nodes can be in $\mathcal{J}_{\mathcal{L}}$. This process terminates when $\mathcal{J}_{\mathcal{L}}$ is empty.

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 Def. 10. The pair of vector clocks satisfying the above conditions are added to $\mathcal{J}_{\mathcal{L}}$.

Algorithm 3 UPDATEEVENT

```
1: procedure UPDATEEVENT(e, from-queue)
2:     for all e' ∈ κ do
3:         if e' ∈ E_a ∧ e.index ∈ inv(e'.action) then
4:             MODIFYQUEUE(e, from-queue, false)
5:             return
6:         end if
7:     end for
8:     for all η ∈ L.nodes do
9:         η ← update(η, e)
10:    end for
11:    MODIFYQUEUE(e, from-queue, true)
12: end procedure
```

CHECKQUEUE recalls the events stored in the queue $e \in \kappa$ and then executes MAKE$(e, \texttt{true})$, to check whether the conditions for taking them into account to update the lattice hold. 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, 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. Note, when the algorithm is iterating over the events in the queue, i.e., when the value of variable *from-queue* is true, it is not necessary to iterate over the queue again (Algorithm 2, Line 17).

REMOVEEXTRANODES removes the extra nodes of the lattice. Since our online algorithm is used for runtime monitoring purposes, each node n represents the evaluation of system execution up to node n. Hence, the nodes which reflect the state of the system in the past are not valuable for the runtime monitor. For this, 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' s.t. 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 the occurrence of an event which has greater clock stamps than $\eta.clock$.

5 Properties of the Constructed Lattice

We give the properties of the lattice constructed in the previous section.

5.1 Insensitivity to Communication Delay

Algorithm MAKE can be defined over a sequence of events received by the observer $\zeta = e_1 \cdot e_2 \cdot e_3 \cdots e_z \in E^*$ by applying it sequentially from e_1 to e_z with the initial lattice $init_{\mathcal{L}}$ and an empty queue.

Proposition 1 (Insensitivity to the reception order). $\forall \zeta, \zeta' \in E^*, \forall S_j \in \mathbf{S} \,.\, \zeta \downarrow_{S_j} = \zeta' \downarrow_{S_j} \implies \textsc{Make}(\zeta) = \textsc{Make}(\zeta')$*, where* $\zeta \downarrow_{S_j}$ *is the projection of* ζ *on scheduler* S_j *which results the sequence of events generated by* S_j*.*

Proposition 1 states that different ordering of the events does not affect the output result of Algorithm MAKE. Note, Proposition 1 assumes that all events in ζ and ζ' can be distinguished. For a sequence of events $\zeta \in E^*$, $\textsc{Make}(\zeta).lattice$ denotes the constructed computation lattice $\mathcal{L}$ by algorithm MAKE.

5.2 Correctness of Lattice Construction

Computation lattice $\mathcal{L}$ has a *frontier* node, which is the node with the greatest vector clock. A path of the constructed computation lattice $\mathcal{L}$ is a sequence of causally related nodes of the lattice, starting from the initial node and ending up in the frontier node.

Definition 14 (Set of the paths of a lattice). *The set of the paths of a constructed computation lattice* $\mathcal{L}$ *is* $\Pi(\mathcal{L}) = \Big\{\eta_0 \cdot \alpha_1 \cdot \eta_1 \cdot \alpha_2 \cdot \eta_2 \cdots \alpha_z \cdot \eta_z \mid \eta_0 = init_{\mathcal{L}} \wedge \forall r \in [1\,..\,z] \,.\, \Big(\eta_{r-1} \overset{\alpha_r}{\rightarrowtail\!\!\!\!\!\!\rightarrow} \eta_r \vee (\exists N \subseteq \mathcal{L}.nodes \,.\, \eta_{r-1} = \text{meet}(N, \mathcal{L}) \wedge \eta_r = \text{join}(N, \mathcal{L}) \wedge \forall \eta \in N \,.\, \eta_{r-1} \overset{a_\eta}{\rightarrowtail\!\!\!\!\!\!\rightarrow} \eta \wedge \alpha_r = \bigcup_{\eta \in N} a_\eta)\Big)\Big\}$*, where the notions of* meet *and* join *are naturally extended to a set of nodes.*

A path is a sequence of nodes s.t. for each pair of adjacent nodes either (i) the prior node and the next node are related according to $\rightarrowtail\!\!\!\!\!\!\rightarrow$ or (ii) the prior and the next node are the meet and the join of a set of existing nodes respectively. A path from a meet node to the associated join node represents an execution of a set of concurrent interactions.

At runtime, the execution of such a system produces a partial 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 partial states and global actions (Def. 6). Due to the occurrence of concurrent interactions and internal actions, each partial trace can be represented as a set of compatible and possible partial traces.

Definition 15 (Compatible partial-traces of a partial trace). *The set of all compatible partial-traces of partial trace* t *is* $\mathcal{P}(t) = \{t' \in Q \cdot (GAct \cdot Q)^* \mid \forall j \in [1\,..|\mathbf{S}|], t' \downarrow_{S_j} = t \downarrow_{S_j} = s_j(t)\}$.

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 partial 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 partial-traces can be encoded from a partial trace.

Note that two compatible traces with only difference in the ordering of their internal actions are considered as a unique compatible trace. Two compatible traces of a partial trace differ if they have different ordering of interactions.

For monitoring purposes we need to represent the run of the system by a sequence of global states (recall that we consider properties over global states). For this, we extend the technique in [28], to define a function which takes as input a partial trace of the distributed system (i.e., a sequence of partial 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 partial trace.

Definition 16 (Function refine $\mathcal{R}$). *Function* $\mathcal{R} : Q \cdot (GAct \cdot Q)^* \longrightarrow Q \cdot (Int \cdot Q)^*$ *is defined as* $\mathcal{R}(init) = init$ *and:*

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

with $\text{upd} : Q \times (Q \cup 2^{Int}) \longrightarrow Q \cup 2^{Int}$ *defined as:* $\text{upd}((q_1, \ldots, q_{|\mathbf{B}|}), \alpha) = \alpha$*, and* $\text{upd}\Big((q_1, \ldots, q_{|\mathbf{B}|}), (q'_1, \ldots, q'_{|\mathbf{B}|})\Big) = (q''_1, \ldots, q''_{|\mathbf{B}|})$*, where* $\forall k \in [1\,..\,|\mathbf{B}|], q''_k = q_k$ *if* $(q_k \notin Q^{\text{b}}_k) \wedge (q'_k \in Q^{\text{b}}_k)$ *and* q'_k *otherwise.*

Function $\mathcal{R}$ uses the state after internal actions in order to update the partial states using function upd.

By applying function $\mathcal{R}$ to the set of compatible partial-traces $\mathcal{P}(t)$, we obtain a new set of global traces, which is (i) equivalent to $\mathcal{P}(t)$ (according to [28], Def. 7), (ii) internal actions are discarded in the presentation of each global trace and (iii) contains maximal global states that can be built with the information contained in the partial states observed so far. In Sec. 3.2 (Def. 7) we define $\big\{s_1(t), \ldots, s_{|\mathbf{S}|}(t)\big\}$, the set of observable local partial-traces of the schedulers obtained from partial trace t. From each local partial-trace we can obtain the sequences of events generated by the controller of each scheduler, s.t. the set of all the sequences of the events is $\{\text{event}(s_1(t)), \ldots, \text{event}(s_{|\mathbf{S}|}(t))\}$ 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 17 (Events ordering). *Considering partial 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 compatible with the local events of schedulers.

Proposition 2 (Soundness). *Given a partial trace* t *as per Def. 6, we have:*
$\forall \zeta \in \Theta(t), \forall \pi \in \Pi(\textsc{Make}\,(\zeta)\,.lattice), \forall j \in [1\,..|\mathbf{S}|] \,.\, \pi \downarrow_{S_j} = \mathcal{R}(s_j(t))$.

Proposition 2 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 partial-trace of scheduler S_j. The following proposition states the correctness of the construction in the sense that applying Algorithm MAKE to 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, s.t. each sequence represents a global trace of the system.

Proposition 3 (Completeness). *Given a partial trace* t *as per Def. 6, we have:*
$\forall \zeta \in \Theta(t), \forall t' \in \mathcal{P}(t), \exists! \, \pi \in \Pi\Big(\textsc{Make}\,(\zeta)\,.lattice\Big) \,.\, \pi = \mathcal{R}(t')$.

π is said to be the associated path of the compatible partial-trace t'. Applying algorithm MAKE to any sequence of events constructs a computation lattice whose set of paths consists on all the compatible global traces.

Figure 1 depicts an overview of our approach.

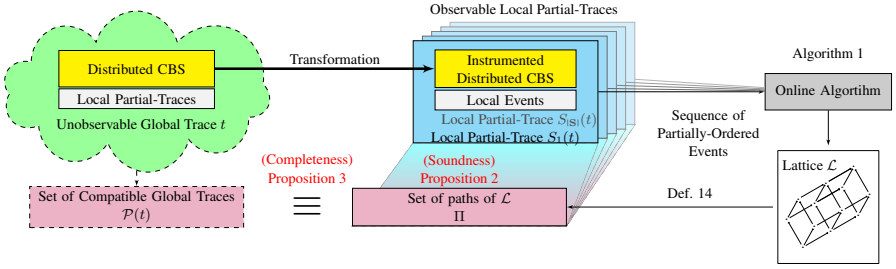

Fig. 1: Overview of the construction of the computation lattice.

6 Related Work

The problem of reconstructing a global behavior from local observations/behaviors has been investigated in several settings. In the setting of choreographies, the approach in [25] reconstructs global graphs from (local) communicating finite-state machines. More recently, in the setting of program replay, the approach in [24] introduces causal-consistent replay to record the execution of a concurrent program and reproduce a misbehavior as well as its causes inside a debugger.

In the following, we focus our comparison on the research efforts that contributed to the distribution of the monitoring process. The approaches in [4,15,12,14] define algorithms for decentralized monitoring for distributed systems with a global clock. In comparison, we target asynchronous distributed CBSs with a partial-state semantics, where global states are not available at runtime. Hence, instead of having a global trace at runtime, we deal with a set of compatible partial traces which could have happened at runtime. The approach in [5] detects and analyzes synchronous distributed systems faults in a centralized manner using local LTL properties evaluated with local traces. In our setting, global properties can not be projected and checked on individual components nor on individual schedulers. Thus, local traces can not be directly used for verifying properties. In [31], the authors designed a method for monitoring safety properties in distributed systems relying on the existing communication among processes. Compared to [31], our algorithm is sound, in the sense that we reconstruct the behavior of the distributed system based on all possible partial-traces of the distributed system. In our work, each trace could have happened as the actual trace of the system, and could have generated the same events. The approach in [26] monitors LTL properties on finite executions of a distributed system. In comparison, our approach is tailored to and leverages the structure of CBSs; traces are defined over partial states and are obtained from a generic semantic model of CBSs.

There are several approaches dedicated to the monitoring of CBSs: [13,21] for the correctness of reconfigurations of Fractal [11] components, [17] and [28] for the runtime verification of functional properties on respectively sequential and multi-threaded BIP [2] CBSs, [8] for the runtime checking of local behaviors specified as quantitative properties. However, these approaches do not handle fully concurrent components and they either assume a global clock or a shared memory. This is for instance the case with our previous work on monitoring multithreaded CBSs [28] where we assume a global clock shared by the threads used to execute the components.

7 Conclusions

Conclusions. We present a technique that enables the monitoring on distributed CBSs, where the interactions are partitioned among a set of distributed schedulers. Each scheduler is in charge of execution of a subset of interactions. The execution of each interaction triggers the actions of the components involved in the interaction. Our technique consists in (i) transforming the system to generate events associated to partial trace local to each scheduler, (ii) synthesizing a centralized observer which collects the local events of all schedulers (iii) reconstructing on-the-fly the possible orderings of the received events which forms a computation lattice. Our technique leverages the nature of distributed CBSs in that it uses components shared by several interactions to infer causality relations between events. The constructed lattice encodes exactly the compatible global traces: each could have occurred as the actual execution. We implemented our monitoring approach in a tool which executes in parallel with the distributed system and takes as input the events generated from each scheduler and outputs the evaluated computation lattice. Our experimental results, omitted for space reasons, show that even for traces with thousands of events, the lattice size remains reasonable.

Future Work. The first extension of this work is to define how to use the computation lattice for efficiently evaluating properties at runtime. Moreover, we plan to decentralize the runtime monitors so that the satisfaction or violation of specifications can be detected by local monitors alone using decentralized monitoring techniques from [6,15,14] and decentralization/projection techniques for CBSs in [22,8]. 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 [33]. Finally, we plan to go beyond simple monitoring to allow components to react to errors by defining runtime enforcement [19] approaches for concurrent CBSs. For this, we plan extending our runtime enforcement approach [16] defined in the sequential setting to the multithreaded and distributed settings.

Acknowledgment. The authors thank the reviewers for their helpful comments.

The authors acknowledge the support from the H2020-ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), the European Union's Horizon 2020 research and innovation programme - Grant Agreement number 956123 (FOCETA), from the French ANR project ANR-20- CE39-0009 (SEVERITAS), the Auvergne-Rhône-Alpes research project MOAP, and LabEx PERSYVAL- Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018)
2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. 28(3), 41–48 (2011)
3. 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. pp. 116–133 (2008)
4. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods Syst. Des. 48(1-2), 46–93 (2016), https://doi.org/10.1007/s10703-016-0253-8
5. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: Proceedings of the Australian Software Engineering Conference (ASWEC'06). p. 243–252. IEEE (2006)
6. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: FM 2012: Formal Methods - 18th International Symposium. pp. 85–100 (2012)
7. Bensalem, S., Bozga, M., Quilbeuf, J., Sifakis, J.: Optimized distributed implementation of multiparty interactions with restriction. Sci. Comput. Program. 98, 293–316 (2015)
8. Bistarelli, S., Martinelli, F., Matteucci, I., Santini, F.: A formal and run-time framework for the adaptation of local behaviours to match a global property. In: Kouchnarenko, O., Khosravi, R. (eds.) Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10231, pp. 134–152 (2016)
9. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: International Conference on Concurrency Theory. pp. 508–522 (2008)
10. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Automated distributed implementation of component-based models with priorities. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.) Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011. pp. 59–68. ACM (2011)
11. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL component model and its support in java. Softw. Pract. Exp. 36(11-12), 1257–1284 (2006)
12. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods Syst. Des. 49(1-2), 109–158 (2016)
13. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) Proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010). LNCS, vol. 6921, pp. 200–217. Springer (2010)
14. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. 29(1), 1:1–1:57 (2020)
15. Falcone, Y., Cornebize, T., Fernandez, J.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8461, pp. 66–83. Springer (2014)
16. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. Int. J. Softw. Tools Technol. Transf. 19(3), 341–365 (2017)

17. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling 14(1), 173–199 (2015)
18. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. 23(2), 255–284 (2021)
19. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 103–134. Springer (2018)
20. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering (1987)
21. Kouchnarenko, O., Weber, J.: Adapting component-based systems at runtime via policies with temporal patterns. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8348, pp. 234–253. Springer (2013)
22. Kouchnarenko, O., Weber, J.: Decentralised evaluation of temporal patterns over component-based systems at runtime. In: Lanese, I., Madelaine, E. (eds.) Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014. Lecture Notes in Computer Science, vol. 8997, pp. 108–126. Springer (2014)
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
24. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. Fundam. Informaticae 178(3), 229–266 (2021)
25. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 221–232. ACM (2015)
26. Massart, T., Meuter, C.: Efficient online monitoring of LTL properties for asynchronous distributed systems. Université Libre de Bruxelles, Tech. Rep (2006)
27. Mattern, F.: Virtual time and global states of distributed systems. Parallel and Distributed Algorithms 1(23), 215–226 (1989)
28. Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M.: Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. Formal Aspects of Computing pp. 1–36 (2017)
29. Nazarpour, H., Falcone, Y., Jaber, M., Bensalem, S., Bozga, M.: Monitoring distributed component-based systems. ArXiv e-prints (2017)
30. Runtime Verification: http://www.runtime-verification.org (2001-2021)
31. 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)
32. 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)
33. 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)