Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation

Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga

Concurrency-Preserving and Sound Monitoring of Multi-Threaded Component-Based Systems

Theory, Algorithms, Implementation, and Evaluation

Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga

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

Abstract. This paper addresses the monitoring of logic-independent linear-time user-provided properties in multi-threaded component-based systems. We consider intrinsically independent components that can be executed concurrently with a centralized coordination for multiparty interactions. In this context, the problem that arises is that a global state of the system is not available to the monitor. A naive solution to this problem would be to plug in a monitor which would force the system to synchronize in order to obtain the sequence of global states at runtime. Such a solution would defeat the whole purpose of having concurrent components. Instead, we reconstruct on-the-fly the global states by accumulating the partial states traversed by the system at runtime. We define transformations of components that preserve their semantics and concurrency and, at the same time, allow to monitor global-state properties. Moreover, we present RVMT-BIP, a prototype tool implementing the transformations for monitoring multi-threaded systems described in the BIP (Behavior, Interaction, Priority) framework, an expressive framework for the formal construction of heterogeneous systems. Our experiments on several multi-threaded BIP systems show that RVMT-BIP induces a cheap runtime overhead.

1 Introduction

Component-based design is the process leading from given requirements and a set of predefined components to a system meeting the requirements. Building systems from components is essential in any engineering discipline. Components are abstract building blocks encapsulating behavior. They can be composed in order to build composite components. Their composition should be rigorously defined so that it is possible to infer the behavior of composite components from the behavior of their constituents as well as global properties from the properties of individual components.

The problem of building component-based systems (CBSs) can be defined as follows. Given a set of components $\{B_1, \ldots, B_n\}$ and a property of their product state space φ, find multiparty interactions γ (i.e., "glue" code) such that the coordinated behavior $\gamma(B_1, \ldots, B_n)$ meets the property φ. It is, however, generally not possible to ensure or verify the desired property φ using static verification techniques such as model-checking or static analysis, either because of the state-explosion problem or because φ can only be decided with information available at runtime (e.g., from the user or the environment). In this paper, we are interested in complementary verification techniques for CBSs such as runtime verification. In [FJN+15], we introduced runtime verification of sequential CBSs against properties referring to the global states of the system, which, in particular, implies that properties can not be "projected" and checked on individual components. From an input composite system $\gamma\left(B_1, \ldots, B_n\right)$ and a regular linear-time property, a component monitor M and a new set of interactions γ' are synthesized to build a new composite system $\gamma'\left(B_1, \ldots, B_n, M\right)$ where the property is checked at runtime.

The underlying model of CBSs relies on multiparty interactions which consist of actions that are jointly executed by certain components, either sequentially or concurrently. In the sequential setting, components are coordinated by a single centralized controller and joint actions are atomic. Components notify the controller of their current states. Then, the controller computes the possible interactions, selects one, and then sequentially executes the actions of each component involved in the interaction. When components finish their executions, they notify the controller of their new states, and the aforementioned steps are repeated. For performance reasons, it is desirable to parallelize the execution of components. In the multi-threaded setting, each component executes on a thread and a controller is in charge of coordination. Parallelizing the execution of $\gamma\left(B_1, \ldots, B_n\right)$ yields a bisimilar [Mil95] component ([BBBS08]) where each synchronized action a occurring on B_i is broken down into β_i and a' where β_i represents an internal computation of B_i and a' is a synchronization action. Between β_i and a', a new *busy location* is added. Consequently, the components can perform their interaction independently after synchronization, and the joint actions become non atomic. After starting an interaction, and

before this interaction completes (meaning that certain components are still performing internal computations), the controller can start another interaction between ready components.

The problem that arises in the multi-threaded setting is that a global steady state of the system (where all components are ready to perform an interaction) may never exist at runtime. Note that we do not target distributed but multi-threaded systems in which components execute with a centralized controller, there is a global clock and communication is instantaneous and atomic. We define a method to monitor CBSs against any linear-time property referring to global states. Our method preserves the concurrency and semantics of the monitored system. It transforms the system so that global states can be reconstructed by accumulating partial states at runtime. The execution trace of a multi-threaded CBS is a sequence of partial states. For an execution trace of a multi-threaded CBS, we define the notion of *witness* trace, which is intuitively the unique trace of global states corresponding to the trace of the multi-threaded CBS if this CBS was executed on a single thread. For this purpose, we define transformations allowing one to add a new component building the witness trace.

We prove that the transformed and initial systems are bisimilar: the obtained reconstructed sequence of global states from a parallel execution is as the sequence of global states obtained when the multi-threaded CBS is executed with a single thread.

We introduce RVMT-BIP, a tool integrated in the BIP tool suite.[1] BIP (Behavior, Interaction, Priority) framework is a powerful and expressive component framework for the formal construction of heterogeneous systems. BIP offers two powerful mechanisms for composing components by using *multiparty interactions* and *priorities*. The combination of interactions and priorities is expressive enough to express usual composition operators of other languages as shown in [BS07]. A system model is layered. The lowest layer contains atomic components whose behavior is described by state machines with data and functions described in the C language. As in process algebras, atomic components can communicate by using ports. The second layer contains interactions which are relations between communication ports of individual components. Priorities are used to express scheduling policies by selecting among the enabled interactions of the layer underneath. RVMT-BIP takes as input a BIP CBS and a monitor description which expresses a property φ, and outputs a new BIP system whose behavior is monitored against φ while running concurrently. Figure 1 presents an overview of our approach. Recall that according to [BBBS08], a BIP system with global-state semantics S_g (sequential model), is (weakly) bisimilar with the corresponding partial-state model S_p (concurrent model). This is formalized as $S_g \sim S_p$ ($\sim$ is formally defined in Section 2). Moreover, S_p generally runs faster than S_g because of its parallelism. Thus, if a trace of S_g, i.e., σ_g, satisfies φ, then the corresponding trace of S_p, i.e., σ_p, satisfies φ as well. The technique in [FJN$^+$15] *could* serve as a monitoring solution. In short, [FJN$^+$15] instruments the components and synthesizes additional interactions in such a way that, whenever the system performs an interaction, the monitor receives the current global state of the system. Hence, based on [FJN$^+$15], a couple of naive solutions to monitor S_p would be (i) to monitor S_g and run S_p, which would incur unpredictable delays in detecting verdicts or (ii) plug the monitor (as in [FJN$^+$15]) into S_p, which would force the (concurrent) components to synchronize for the monitor to take a snapshot of the global state of the system. Such solutions would completely defeat the purpose of using multi-threaded models. Instead, we here propose a transformation technique to build another system S_{pg} out of S_p such that (i) S_{pg} and S_p are bisimilar (hence S_g and S_{pg} are bisimilar), (ii) S_{pg} is as concurrent as S_p and preserves the performance gained from multi-threaded execution and (iii) S_{pg} produces a witness trace, that is the *unique* trace that allows to check the property φ. Our method does not introduce any delay in the detection of verdicts since it always reconstructs the maximal (information-wise) prefix of the witness trace (Theorem 1). Moreover, we show that our method is correct in the sense that it always produces the correct witness trace (Theorem 2).

Remark 1 (On the monitored properties). Note that our approach allows one to monitor *any* linear-time property. Moreover, how the property is defined is irrelevant as one can use the approaches in [BLS10,FFM12] to synthesize a monitor which emits verdicts in a 4-valued domain. Our approach directly uses the definition of a monitor as input and is thus compatible with the various approaches compatible with the ones in [BLS10,FFM12].

This paper extends a previous contribution [NFB$^+$16] that appeared in the 12$^{\text{th}}$ International Conference on integrated Formal Methods, with the following additional contributions:

- we propose detailed and rigorous proofs of the propositions and theorems related to the soundness of our monitoring approach;

[1] RVMT-BIP is available for download at [Naz].

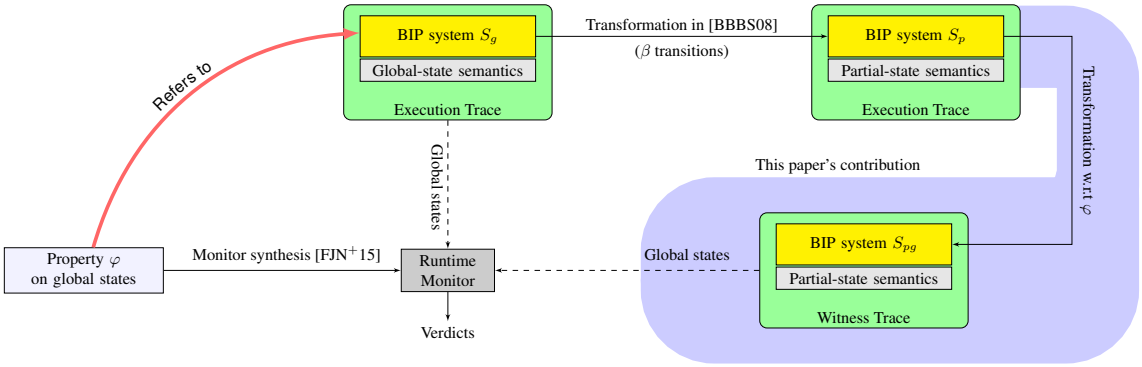


Fig. 1: Approach overview

- we improve the presentation and readability of [NFB+16] by (i) formalizing some concepts that remained informal in the conference version, (ii) providing more detailed explanations in each section, and (iii) illustrating the concepts with additional examples;
- we present the actual algorithms used in the instrumented system to reconstruct global states;
- we further validate our approach against additional case studies and report on additional experimental data;
- we propose a deeper study of related work.

Running example. We use a task system, called Task, to illustrate our approach throughout the paper. The system consists of a task generator (*Generator*) along with 3 task executors (*Workers*) that can run in parallel. Each newly generated task is processed whenever two cooperating workers are available. A desirable property of system Task is the homogeneous distribution of the tasks among the workers.

Outline. The remainder of this paper is organized as follows. Section 2 introduces some preliminary concepts. Section 3 overviews CBS design and semantics. In Section 4, we define a theoretical framework for the monitoring of multi-threaded CBSs. In Section 5, we present the transformation of a multi-threaded CBS model for introducing monitors. Section 6 describes RVMT-BIP, an implementation of the approach and its evaluation on several examples. Section 7 presents related work. Section 8 concludes and presents future work. Complete proofs related to the correctness of the approach are given in Appendix A.

2 Preliminaries and Notations

We introduce some preliminary concepts and notations.

Functions. For two domains of elements E and F, we note $E \to F$ the set of functions from E to F. For two functions $v \in X \to Y$ and $v' \in X' \to Y'$, the function obtained by overriding v images by v' images is denoted by $v \backslash v'$, where $v \backslash v' \in X \cup X' \to Y \cup Y'$, and is defined as follows:

$$v \backslash v'(x) = \begin{cases} v'(x) \text{ if } x \in X', \\ v(x) \text{ otherwise.} \end{cases}$$

Sequences. 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 $\forall i \in [1\,..\,n] \centerdot e_i \in E$. Sequences of assignments are delimited by square brackets for clarity. The empty sequence is denoted by ϵ or $[\,]$, depending on the context. The set of (finite) sequences over E is denoted by E^*. E^+ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence s is denoted by $\mathrm{length}(s)$. 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. We also denote by $\mathrm{pref}(s)$, the set of *prefixes* of s such that $\mathrm{pref}(s) = \{s(1 \cdots k) \mid k \leq \mathrm{length}(s)\}$. Operator pref is naturally extended to sets of sequences. Function $\max_{\preceq}$ (resp. $\min_{\preceq}$) returns the maximal (resp. minimal) sequence w.r.t. prefix ordering of a set of sequences. We define function $\mathrm{last} : E^+ \to E$ such that $\mathrm{last}(e_1 \cdot e_2 \cdots e_n) = e_n$.

Map operator: applying a function to a sequence. For a sequence $e = e_1 \cdot e_2 \cdots e_n$ of elements over E of some length $n \in \mathbb{N}$, and a function $f : E \to F$, $\mathrm{map}\ f\ e$ is the sequence of elements of F defined as $f(e_1) \cdot f(e_2) \cdots f(e_n)$ where $\forall i \in [1\,..\,n] \centerdot f(e_i) \in F$.

Labeled transition systems. Labeled Transition System (LTS) are used to define the semantics of component-based systems. An LTS is defined over an alphabet Σ and is a 3-tuple $(\mathrm{Sta}, \mathrm{Lab}, \mathrm{Trans})$ where Sta is a non-empty set of states, Lab is a set of labels, and $\mathrm{Trans} \subseteq \mathrm{Sta} \times \mathrm{Lab} \times \mathrm{Sta}$ is the transition relation. A transition $(q, e, q') \in \mathrm{Trans}$ means that the LTS can move from state q to state q' by consuming label e. We abbreviate $(q, e, q') \in \mathrm{Trans}$ by $q \xrightarrow{e}_{\mathrm{Trans}} q'$ or by $q \xrightarrow{e} q'$ 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 [Mil95], where θ-transitions are considered to be unobservable. Given two transition systems $S_1 = (\mathrm{Sta}_1, \mathrm{Lab} \cup \{\theta\}, \rightarrow_{\mathrm{Trans}_1})$ and $S_2 = (\mathrm{Sta}_2, \mathrm{Lab} \cup \{\theta\}, \rightarrow_{\mathrm{Trans}_2})$, system S_1 *weakly simulates* system S_2, if there exists a relation $R \subseteq \mathrm{Sta}_1 \times \mathrm{Sta}_2$ such that the two following conditions hold:

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

Equation *1.* says 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.* says 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 the 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.

3 Component-Based Systems with Multiparty Interactions

An action of a CBS is an interaction i.e., a coordinated operation between certain atomic components. Atomic components are transition systems with a set of ports labeling individual transitions. Ports are used by components to communicate. Composite components are obtained from atomic components by specifying interactions.

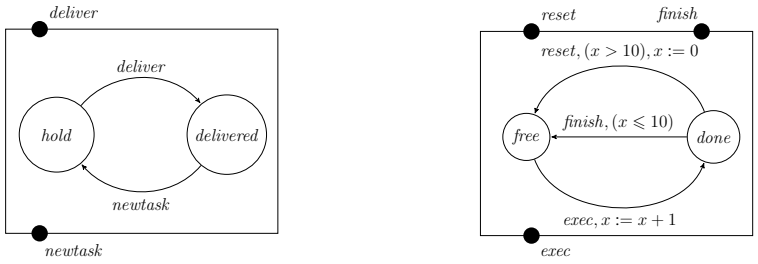
Atomic Components. An atomic component is endowed with a finite set of local variables X taking values in a set Data. Atomic components synchronize and exchange data with other components through *ports*.

Definition 1 (Port). *A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier p and some data variables in a set x_p.*

Variables attached to ports are purposed to transfer values between interacting components (see also Definition 3 for interactions). The variables attached to the port are also used to determine whether a communication through this port can take place (see below).

Definition 2 (Atomic component). *An atomic component is defined as a tuple (P, L, T, X) where P is the set of ports, L is the set of (control) locations, $T \subseteq L \times P \times \mathcal{G}(X) \times \mathcal{F}^*(X) \times L$ is the set of transitions, and X is the set of variables. $\mathcal{G}(X)$ denotes the set of Boolean expressions over X and $\mathcal{F}(X)$ the set of assignments of expressions over X to variables in X. For each transition $\tau = (l, p, g_\tau, f_\tau, l') \in T$, g_τ is a Boolean expression over X (the guard of τ), $f_\tau \in \{x := f^x(X) \mid x \in X \wedge f^x \in \mathcal{F}(X)\}^*$: the computation step of τ, a sequence of assignments to variables.*

The semantics of the atomic component is an LTS $(Q, P, \rightarrow)$ where $Q = L \times (X \rightarrow \mathrm{Data})$ is the set of states, and $\rightarrow = \{((l, v), p(v_p), (l', v')) \in Q \times P \times Q \mid \exists \tau = (l, p, g_\tau, f_\tau, l') \in T \,.\, g_\tau(v) \wedge v' = f_\tau(v \backslash v_p)\}$ is the transition relation.



(a) Component $Generator$ (b) Component $Worker$

Fig. 2: Atomic components of system Task

A state is a pair $(l, v) \in Q$, where $l \in L$, $v \in X \to \mathrm{Data}$ is a valuation of the variables in X. The evolution of states $(l, v) \xrightarrow{p(v_p)} (l', v')$, where v_p is a valuation of the variables x_p attached to port p, is possible if there exists a transition $(l, p[x_p], g_\tau, f_\tau, l')$, such that $g_\tau(v) = \texttt{true}$. As a result, the valuation v of variables is modified to $v' = f_\tau(v \backslash v_p)$.

We use the dot notation to denote the elements of atomic components. e.g., for an atomic component B, $B.P$ denotes the set of ports of the atomic component B, $B.L$ denotes its set of locations, etc.

Example 1 (Atomic component). Figure 2 shows the atomic components of system Task.

- Figure 2a depicts a model of component $Generator$[2] defined as follows:
  - $Generator.P = \{deliver[\emptyset],\ newtask[\emptyset]\}$,
  - $Generator.L = \{hold\ ,\ delivered\}$,
  - $Generator.T = \{(hold, deliver, \texttt{true}, [\,], delivered), (delivered\ , newtask, \texttt{true}, [\,], hold)\}$,
  - $Generator.X = \emptyset$.
- Figure 2b depicts a model of component $Worker$ defined as follows:
  - $Worker.P = \{exec[\emptyset], finish[\emptyset], reset[\emptyset]\ \}$,
  - $Worker.L = \{free, done\}$,
  - $Worker.T = \{(free, exec, \texttt{true}, [x := x+1], done), (done, finish, (x \leqslant 10), [\,], free), (done, reset, (x > 10), [x := 0], free)\}$,
  - $Worker.X = \{x\}$.

Definition 3 (Interaction). *An interaction a is a tuple $(\mathcal{P}_a, F_a)$, where $\mathcal{P}_a = \{p_i[x_i] \mid p_i \in B_i.P\}_{i \in I}$ is the set of ports such that $\forall i \in I\,.\,\mathcal{P}_a \cap B_i.P = \{p_i\}$ and F_a is a sequence of assignments to the variables in $\cup_{i \in I} x_i$.*

When clear from context, in the following examples, an interaction $(\{p[x_p]\}, F_a)$ consisting of only one port p is denoted by p.

Definition 4 (Composite component). *A composite component $\gamma(B_1, \ldots, B_n)$ is defined from a set of atomic components $\{B_i\}_{i=1}^n$ and a set of interactions γ.*

A state q of a composite component $\gamma(B_1, \ldots, B_n)$ is an n-tuple $q = (q_1, \ldots, q_n)$, where $q_i = (l_i, v_i)$ is a state of atomic component B_i. The semantics of the composite component is an LTS $(Q, \gamma, \longrightarrow)$, where $Q = B_1.Q \times \ldots \times B_n.Q$ is the set of states, γ is the set of all possible interactions and $\longrightarrow$ is the least set of transitions satisfying the following rule:

$$\frac{a = (\{p_i[x_i]\}_{i \in I}, F_a) \in \gamma \qquad \forall i \in I\,.\,q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{a_i}(v(X)) \qquad \forall i \notin I\,.\,q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

X is the set of variables attached to the ports of a, v is the global valuation, and F_{a_i} is the restriction of F to the variables of p_i.

[2] For the sake of simpler notation, the variables attached to the ports are not shown.

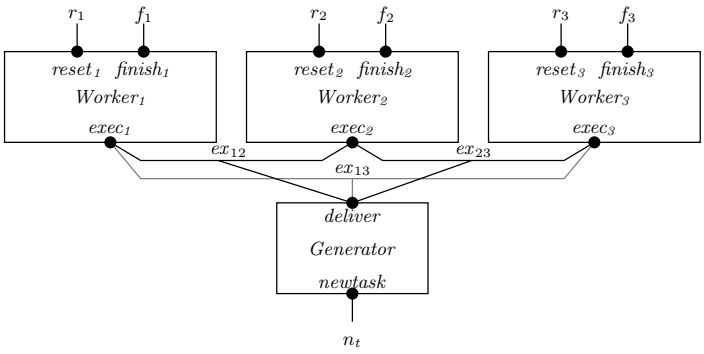


Fig. 3: Composite component of system Task

The semantic rule in Definition 4 says that a composite component moves from state $(q_1, \ldots, q_n)$ to a state $(q'_1, \ldots, q'_n)$ through some interaction a if there exists an interaction $a \in \gamma$ of the form $(\{p_i[x_i]\}_{i\in I}, F_a)$, i.e., involving component of index in a set $I \subseteq [1\,..\,n]$. The components involved in interaction a (i.e., components with index in set I) evolve according to their transition relation $\longrightarrow_i$ (as per Definition 2): they move from state q_i to state q'_i by executing port p_i with valuation v_i obtained after executing the assignments F_{a_i} related to the variables of port p_i (obtained from the sequence of assignments F_a of interaction a). The components not involved in interaction a (i.e., components with index not in set I) remain in the same state.

A trace is a sequence of states and interactions $(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s)$ such that: $q_0 = Init \wedge (\forall i \in [1\,..\,s] \,.\, q_i \in Q \wedge a_i \in \gamma \wedge q_{i-1} \xrightarrow{a_i} q_i)$, where $Init \in Q$ is the initial state. Given a trace $(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s)$, the sequence of interactions is defined as $\text{interactions}(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s) = a_1 \cdots a_s$. The set of traces of composite component B is denoted by $\text{Tr}(B)$.

Example 2 (Interaction, composite component). Figure 3 depicts the composite component $\gamma(Worker_1, Worker_2, Worker_3, Generator)$ of system Task, where each $Worker_i$ is identical to the component in Fig. 2b and $Generator$ is the component depicted in Fig. 2a. The set of interactions is $\gamma = \{ex_{12}, ex_{13}, ex_{23}, r_1, r_2, r_3, f_1, f_2, f_3, n_t\}$. We have $ex_{12} = (\{deliver, exec_1, exec_2\}, [\,])$, $ex_{23} = (\{deliver, exec_2, exec_3\}, [\,])$, $ex_{13} = (\{deliver, exec_1, exec_3\}, [\,])$, $r_1 = (\{reset_1\}, [\,])$, $r_2 = (\{reset_2\}, [\,])$, $r_3 = (\{reset_3\}, [\,])$, $f_1 = (\{finish_1\}, [\,])$, $f_2 = (\{finish_2\}, [\,])$, $f_3 = (\{finish_3\}, [\,])$, and $n_t = (\{newtask\}, [\,])$.

One of the possible traces[3] of system Task is: $(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t \cdot (done, done, free, hold)$ such that from the initial state $(free, free, free, hold)$, where workers are at location *free* and task generator is ready to deliver a task, interaction ex_{12} is fired and $Worker_1$ and $Worker_2$ move to location $done$ and $Generator$ moves to location $delivered$. Then, a new task is generated by the execution of interaction n_t so that $Generator$ moves to location $hold$.

Two composite components are bi-similar if the LTSs of their semantics are bi-similar.

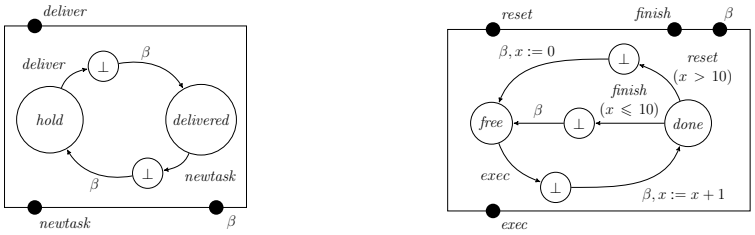
4 Monitoring Multi-Threaded CBSs with Partial-State Semantics

The general semantics defined in the previous section is referred to as the global-state semantics of CBSs because each state of the system is defined in terms of the local states of components, and, all local states are defined. In this section, we consider what we refer to as the partial-state semantics where the states of a system may contain undefined local states because of the concurrent execution of components.

4.1 Partial-State Semantics

To model concurrent behavior, we associate a partial-state model to each atomic component. In global-state semantics, one does not distinguish the beginning of an interaction (or a transition) from its completion. That is, the interactions and transitions of a system execute atomically and sequentially. Partial states and the corresponding internal transitions are needed for modeling non-atomic executions. Atomic components with partial states behave as atomic components except that each transition is decomposed into a sequence of two transitions: a visible transition followed by an internal β-labeled transitions (aka busy transition). Between these

[3] For the sake of simpler notation, we denote a state by its location (and omit the valuation of variables).



(a) $Generator^{\perp}$ (b) $Worker^{\perp}$

Fig. 4: Atomic components of system Task with partial-states

two transitions, a so-called *busy location* is added. Intuitively, busy transitions are notifications indicating the completion of internal computations. Below, we define the transformation of a component with global-state semantics to a component with partial-state semantics (extending the definition in [BBBS08] with variables, guards, and computation steps on transitions).

Definition 5 (Atomic component with partial states). *The partial-state-semantics version of atomic component $B = (P, L, T, X)$ is $B^{\perp} = (P \cup \{\beta\},\ L \cup L^{\perp},\ T^{\perp},\ X)$, where $\beta \notin P$ is a special port, $L^{\perp} = \{l_t^{\perp} \mid t \in T\}$ (resp. L) is the set of busy locations (resp. ready locations) such that $L^{\perp} \cap L = \emptyset$ and $T^{\perp} = \{(l, p, g_{\tau}, [\,], l_{\tau}^{\perp}), (l_{\tau}^{\perp}, \beta, \texttt{true}, f_{\tau}, l') \mid \exists \tau = (l, p, g_{\tau}, f_{\tau}, l') \in T\}$ is the set of transitions.*

Assuming some available atomic components with partial states $B_1^{\perp}, \ldots, B_n^{\perp}$, we construct a composite component with partial states.

Definition 6 (Composite component with partial states). *$B^{\perp} = \gamma^{\perp}(B_1^{\perp}, ..., B_n^{\perp})$ is a composite component where $\gamma^{\perp} = \gamma \cup \{\{\beta_i\}\}_{i=1}^{n}$, and $\{\{\beta_i\}\}_{i=1}^{n}$ is the set of singleton busy interactions.*

The notions and notation related to traces are lifted to components with partial states in the natural way. We extend the definition of interactions (defined in Section 3) to traces in partial-state semantics such that β interactions are filtered out.

Example 3 (Composite component with partial states). The corresponding composite component of system Task with partial-state semantics is $\gamma^{\perp}(Worker_1^{\perp}, Worker_2^{\perp}, Worker_3^{\perp}, Generator^{\perp})$, where each $Worker_i^{\perp}$ for $i \in [1\,..\,3]$ is identical to the component in Fig. 4b and $Generator^{\perp}$ is the component in Fig. 4a. To simplify the depiction of these components, we represent each busy location $l^{\perp}$ as $\perp$. The set of interactions is $\gamma^{\perp} = \{ex_{12}, ex_{13}, ex_{23}, r_1, r_2, r_3, f_1, f_2, f_3, n_t\} \cup \{\{\beta_1\}, \{\beta_2\}, \{\beta_3\}, \{\beta_4\}\}$. One possible trace of system Task with partial-state semantics is: ($free$, $free$, $free$, $hold$) $\cdot\ ex_{12} \cdot (\perp, \perp, free, \perp) \cdot \beta_4 \cdot (\perp, \perp, free, delivered) \cdot n_t \cdot (\perp, \perp, free, \perp)$.

It is possible to show that the partial-state system is a correct implementation of the global-state system, that is, the two systems are (weakly) bisimilar (cf. [BBBS08], Theorem 1). A weak bisimulation relation R is defined between the set of states of the model in global-state semantics (i.e., Q) and the set of states of its partial-state model (i.e., $Q^{\perp}$), such that $R = \{(q, r) \in Q \times Q^{\perp} \mid r \xrightarrow{\beta^*} q\}$. Any global state in partial-state semantics model is equivalent to the corresponding global state in global-state semantics model, and any partial state in partial-state semantics model is equivalent to the successor global state obtained after stabilizing the system by executing busy interactions (which take place independently).

In the sequel, we consider a CBS with global-state semantics B and its partial-states semantics version $B^{\perp}$. Intuitively, from any trace of $B^{\perp}$, we want to reconstruct on-the-fly the corresponding trace in B and evaluate a property which is defined over global states of B.

Remark 2. We note that transforming a CBS with global-state semantics into a CBS with partial-state semantics resembles a special case of splitting semantics carried out for process algebraic systems [vGV97,Hoa78], i.e., splitting each action into two atomic sub-actions. Indeed, we shall see that the method presented in this paper re-constructs a trace in the interleaving semantics from a trace in the concurrent semantics based on splitting.

4.2 Witness Relation and Witness Trace

We define the notion of *witness* relation between traces in global-state semantics and traces in partial-state semantics, based on the bisimulation between B and $B^{\perp}$. Any trace of $B^{\perp}$ is related to a trace of B, i.e., its

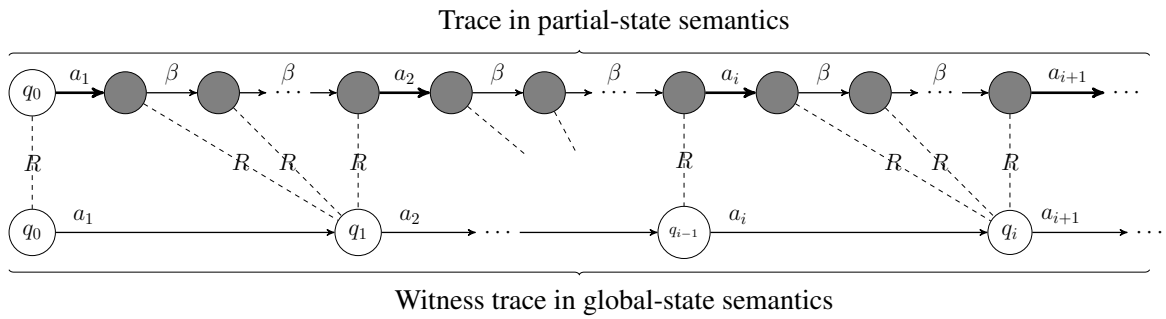


Fig. 5: Witness trace built using weak bisimulation (R)

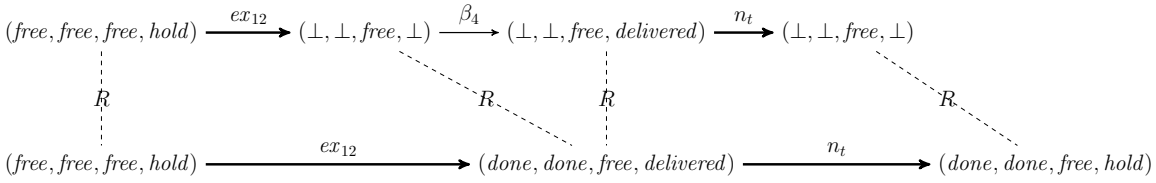


Fig. 6: An example of witness trace in system Task

witness. The witness trace allows to monitor the system in partial-state semantics (thus benefiting from the parallelism) against properties referring to the global behavior of the system.

Definition 7 (Witness relation and witness trace). *Given the bisimulation R between B and $B^{\perp}$, the witness relation* $\mathrm{W} \subseteq \mathrm{Tr}(B) \times \mathrm{Tr}(B^{\perp})$ *is the smallest set that contains* $(Init, Init)$ *and satisfies the following rules:*

- $(\sigma_1 \cdot a \cdot q_1, \sigma_2 \cdot a \cdot q_2) \in \mathrm{W}$, *if* $a \in \gamma$ *and* $(q_1, q_2) \in R$,
- $(\sigma_1, \sigma_2 \cdot \beta \cdot q_2) \in \mathrm{W}$, *if* $(\mathrm{last}(\sigma_1), q_2) \in R$;

whenever $(\sigma_1, \sigma_2) \in \mathrm{W}$.

If $(\sigma_1, \sigma_2) \in \mathrm{W}$, *we say that* σ_1 *is a* witness trace *of* σ_2.

Suppose that the witness relation relates a trace in partial-state semantics σ_2 to a trace in global-state semantics σ_1. The states obtained after executing the same interaction in the two systems are bisimilar. Moreover, any move through a busy interaction in $B^{\perp}$ preserves the bisimulation between the state of σ_2 followed by the busy interaction in $B^{\perp}$ and the last state of σ_1 in B.

Example 4 (Witness relation). Figure 5 illustrates the witness relation. State q_0 is the initial state of B and $B^{\perp}$. In the trace of $B^{\perp}$, gray circles after each interaction represent partial states which are bisimilar to the global state that comes after the corresponding trace of B.

Example 5 (Witness trace). Let us consider σ_2 as a trace of system Task with partial-state semantics depicted in Fig. 6 where $\sigma_2 = (free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot) \cdot \beta_4 \cdot (\bot, \bot, free, delivered) \cdot n_t \cdot (\bot, \bot, free, \bot)$. The witness trace corresponding to trace σ_2 is $(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t \cdot (done, done, free, hold)$.

The following property states that any trace in partial-state semantics and its witness trace have the same sequence of interactions.

Property 1. $\forall (\sigma_1, \sigma_2) \in \mathrm{W} \,.\, \mathrm{interactions}(\sigma_1) = \mathrm{interactions}(\sigma_2)$.

Proof. The proof is done by induction on the length of the sequence of interactions and follows from the definitions of the witness relation and witness trace. The proof of this property can be found in Appendix A.1 (p. 29).

The next property states that any trace in the partial-state semantics has a unique witness trace in the global-state semantics.

Property 2. $\forall \sigma_2 \in \mathrm{Tr}(B^{\perp}), \exists ! \, \sigma_1 \in \mathrm{Tr}(B) \,.\, (\sigma_1, \sigma_2) \in \mathrm{W}$.

Proof. This proof is done by contradiction. The proof of this property is given in Appendix A.2 (p. 29).

We note $\mathrm{W}(\sigma_2) = \sigma_1$ when $(\sigma_1, \sigma_2) \in \mathrm{W}$.

Note that, when running a system in partial-state semantics, the global state of the witness trace after an interaction a is not known until all the components involved in a have reached their ready locations after the execution of a. Nevertheless, even in non-deterministic systems, after a deterministic execution, this global state is uniquely defined and consequently there is always a unique witness trace (that is, non-determinism is resolved at runtime).

4.3 Construction of the Witness Trace

Given a trace in partial-state semantics, the witness trace is computed using function RGT (Reconstructor of Global Trace). The global states (of the trace in the global-state semantics) are reconstructed from partial states. We define a function to reconstruct global states from partial states.

Definition 8 (Function RGT - Reconstructor of Global States). *Function* $\mathrm{RGT} : \mathrm{Tr}(B^{\perp}) \longrightarrow \mathrm{pref}(\mathrm{Tr}(B))$ *is defined as:*

$$\mathrm{RGT}(\sigma) = \mathrm{discriminant}(\mathrm{acc}(\sigma)),$$

where:

- $\mathrm{acc} : \mathrm{Tr}(B^{\perp}) \longrightarrow Q \cdot (\gamma \cdot Q)^* \cdot (\gamma \cdot (Q^{\perp} \backslash Q))^*$ *is defined as:*
  - $\mathrm{acc}(\mathit{Init}) = \mathit{Init}$,
  - $\mathrm{acc}(\sigma \cdot a \cdot q) = \mathrm{acc}(\sigma) \cdot a \cdot q$ *for* $a \in \gamma$,
  - $\mathrm{acc}(\sigma \cdot \beta \cdot q) = \mathrm{map}\ [x \mapsto \mathrm{upd}(q, x)]\ (\mathrm{acc}(\sigma))$ *for* $\beta \in \{\{\beta_i\}\}_{i=1}^{n}$;
- $\mathrm{discriminant} : Q \cdot (\gamma \cdot Q)^* \cdot (\gamma \cdot (Q^{\perp} \backslash Q))^* \longrightarrow \mathrm{pref}(\mathrm{Tr}(B))$ *is defined as:*
  $\mathrm{discriminant}(\sigma) = \max_{\preceq}(\{\sigma' \in \mathrm{pref}(\sigma) \mid \mathrm{last}(\sigma') \in Q\})$

with $\mathrm{upd} : Q^{\perp} \times (Q^{\perp} \cup \gamma) \longrightarrow Q^{\perp} \cup \gamma$ *defined as:*

- $\mathrm{upd}((q_1, \ldots, q_n), a) = a$, *for* $a \in \gamma$,
- $\mathrm{upd}\left((q_1, \ldots, q_n), (q'_1, \ldots, q'_n)\right) = (q''_1, \ldots, q''_n)$,
  where $\forall k \in [1\,..\,n] \centerdot q''_k = \begin{cases} q_k & \text{if } (q_k \notin Q_k^{\perp}) \wedge (q'_k \in Q_k^{\perp}) \\ q'_k & \text{otherwise.} \end{cases}$

Function RGT uses helper functions acc and discriminant. First, function acc is an *accumulator* function which takes as input a trace in partial-state semantics σ, removes β interactions and the partial states after β. Function acc uses the (information in the) partial state after β interactions in order to update the partial states using function upd. Then, function discriminant returns the longest prefix of the result of acc corresponding to a trace in global-state semantics.

Note that, because of the inductive definition of function acc, the input trace can be processed step by step by function RGT and allows to generate the witness incrementally. Moreover, such definition allows to apply the function RGT to a running system by monitoring execution of interactions and partial states of components. Finally, we note that function RGT is monotonic (w.r.t. prefix ordering on sequences).

Such an online computation is illustrated in the following example.

Example 6 (Applying function RGT). Table 1 illustrates Definition 8 on one trace of system Task with initial state $(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold})$ followed by interactions ex_{12}, β_4, n_t, β_2, and β_1. We comment on certain steps illustrated in Table 1. At step 0, the outputs of functions acc and discriminant are equal to the initial state. At step 1, the execution of interaction ex_{12} adds two elements $ex_{12} \cdot (\perp, \perp, \mathit{free}, \perp)$ to traces σ and $\mathrm{acc}(\sigma)$. At step 2, the state after β_4 has fresh information on component $\mathit{Generator}$ which is used to update the existing partial states, so that $(\perp, \perp, \mathit{free}, \perp)$ is updated to $(\perp, \perp, \mathit{free}, \mathit{delivered})$. At step 5, Worker_1 becomes ready after β_1, and the partial state $(\perp, \mathit{done}, \mathit{free}, \mathit{delivered})$ in the intermediate step is updated to the global state $(\mathit{done}, \mathit{done}, \mathit{free}, \mathit{delivered})$, therefore it appears in the output trace.

Table 1: Values of function RGT for a sample input

Step	Input trace in partial semantics σ	Intermediate step $\mathrm{acc}(\sigma)$	Output trace in global semantics $\mathrm{RGT}(\sigma)$
0	$(free, free, free, hold)$	$(free, free, free, hold)$	$(free, free, free, hold)$
1	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot)$	$(free, free, free, hold) \cdot ex_{12}$
2	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot) \cdot \beta_4 \cdot (\bot, \bot, free, delivered)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, delivered)$	$(free, free, free, hold) \cdot ex_{12}$
3	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot) \cdot \beta_4 \cdot (\bot, \bot, free, delivered) \cdot n_t \cdot (\bot, \bot, free, \bot)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, delivered) \cdot n_t \cdot (\bot, \bot, free, \bot)$	$(free, free, free, hold) \cdot ex_{12}$
4	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot) \cdot \beta_4 \cdot (\bot, \bot, free, delivered) \cdot n_t \cdot (\bot, \bot, free, \bot) \cdot \beta_2 \cdot (\bot, done, free, \bot)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, done, free, delivered) \cdot n_t \cdot (\bot, done, free, \bot)$	$(free, free, free, hold) \cdot ex_{12}$
5	$(free, free, free, hold) \cdot ex_{12} \cdot (\bot, \bot, free, \bot) \cdot \beta_4 \cdot (\bot, \bot, free, delivered) \cdot n_t \cdot (\bot, \bot, free, \bot) \cdot \beta_2 \cdot (\bot, done, free, \bot) \cdot \beta_1 \cdot (done, done, free, \bot)$	$(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t \cdot (done, done, free, \bot)$	$(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t$

4.4 Properties of Global-trace Reconstruction

We state some properties of global-trace reconstruction based on function RGT, namely the soundness and maximality (information-wise) of the reconstructed global trace. To do so, we first start by stating some intermediate lemmas on the computation performed by function RGT.

Lemma 1. $\forall(\sigma_1, \sigma_2) \in W \,.\, |\mathrm{acc}(\sigma_2)| = |\sigma_1| = 2s + 1$, *where* $s = |\mathrm{interactions}(\sigma_1)|$, acc *is the accumulator used in the definition of function* RGT *(Definition 8), and function* interactions *(defined in Section 4.1) returns the sequence of interactions in a trace (removing* β*).*

Lemma 1 states that, for a given trace in partial-state semantics σ_2, the length of $\mathrm{acc}(\sigma_2)$ is equal to the length of the witness of σ_2 (i.e., σ_1).

Lemma 2. $\forall \sigma \in \mathrm{Tr}(B^{\bot}) \,.$ *let* $\mathrm{acc}(\sigma) = (q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s)$ *in*

$$\exists k \in [1\,..\,s] \,.\, q_k \in Q \implies \forall z \in [1\,..\,k] \,.\, q_z \in Q \wedge q_{z-1} \xrightarrow{a_z} q_z.$$

Lemma 2 states that, for a given trace in partial-state semantics σ, if there exists a global state $q_k \in Q, k \in [1\,..\,s]$ in sequence $\mathrm{acc}(\sigma)$, then all the states occurring before q_k in $\mathrm{acc}(\sigma)$ are global states.

The next proposition states that the sequence of global states produced by function RGT (which is the composition of functions discriminant and acc) follows the global-state semantics.

Proposition 1. $\forall \sigma \in \mathrm{Tr}(B^{\bot}) \,.$

$$\begin{aligned} &|\mathrm{discriminant}(\mathrm{acc}(\sigma))| \leq |\mathrm{acc}(\sigma)| \\ \wedge\, &\mathrm{discriminant}(\mathrm{acc}(\sigma)) = q_0 \cdot a_1 \cdot q_1 \cdots a_d \cdot q_d \implies \forall i \in [1\,..\,d] \,.\, q_{i-1} \xrightarrow{a_i} q_i, \end{aligned}$$

where acc *(resp.* discriminant*) is the accumulator (resp. discriminant) function used in the definition of function* RGT *(Definition 8) such that* $\mathrm{RGT}(\sigma) = \mathrm{discriminant}(\mathrm{acc}(\sigma))$.

Proposition 1 states that, for any trace in partial-state semantics σ, 1) the length of the output trace of function RGT (i.e., $\mathrm{discriminant}(\mathrm{acc}(\sigma))$) is lower than or equal to the length of the output of function acc (i.e., $\mathrm{acc}(\sigma)$), and 2) the output trace of function RGT is a trace in global-state semantics.

Moreover, the last element of a given trace in partial-state semantics σ is always the same as the last element of output of $\mathrm{acc}(\sigma)$, as stated by the following lemma.

Lemma 3. $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\,\mathrm{last}(\mathrm{acc}(\sigma)) = \mathrm{last}(\sigma)$.

Finally, any trace in partial-state semantics σ and its image through function acc have the same sequence of interactions, as stated by the following lemma.

Lemma 4. $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\,\mathrm{interactions}(\mathrm{acc}(\sigma)) = \mathrm{interactions}(\sigma)$.

Based on the above lemmas, we have the following theorem which states the *soundness* and *maximality* of the reconstructed global trace. That is, applying function RGT on a trace in partial-state semantics produces the longest possible prefix of the corresponding witness trace with respect to the current trace of the partial-state semantics model.

Theorem 1 (On the reconstructed global trace with function RGT). $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.$

$$\begin{array}{r} \mathrm{last}(\sigma) \in Q \implies \mathrm{RGT}(\sigma) = \mathrm{W}(\sigma) \\ \wedge\, \mathrm{last}(\sigma) \notin Q \implies \mathrm{RGT}(\sigma) = \mathrm{W}(\sigma') \cdot a, \textit{with} \\ \sigma' = \min_{\preceq}\{\sigma_p \in \mathrm{Tr}(B^{\perp}) \mid \exists a \in \gamma, \exists \sigma'' \in \mathrm{Tr}(B^{\perp})\,.\,\sigma = \sigma_p \cdot a \cdot \sigma'' \wedge \exists i \in [1\,..\,n]\,. \\ (B_i.P \cap a \neq \emptyset) \wedge (\forall j \in [1\,..\,\mathrm{length}(\sigma'')]\,.\,\beta_i \neq \sigma''(j))\} \end{array}$$

Theorem 1 distinguishes two cases:

- When the last state of a system is a global state ($\mathrm{last}(\sigma) \in Q$), none of the components are in a busy location. Moreover, function RGT has sufficient information to build the corresponding witness trace ($\mathrm{RGT}(\sigma) = \mathrm{W}(\sigma)$).
- When the last state of a system is a partial state, at least one component is in a busy location and function RGT can not build a complete witness trace because it lacks information on the current state of such components. It is possible to decompose the input sequence σ into two parts σ' and σ'' separated by an interaction a. The separation is made on the interaction a occurring in trace σ such that, for the interactions occurring after a (i.e., in σ''), at least one component involved in a has not executed any β transition (which means that this component is still in a busy location). Note that it may be possible to split σ in several manners with the above description. In such a case, function RGT computes the witness for the smallest sequence σ' (w.r.t. prefix ordering) as above because it is the only sequence for which it has information regarding global states. Note also that such splitting of σ is always possible as $\mathrm{last}(\sigma) \notin Q$ implies that σ is not empty, and σ' can be chosen to be ϵ.

In both cases, because of its inductive definition and monotonicity, RGT returns the maximal prefix of the corresponding witness trace that can be built with the information contained in the partial states observed so far.

The above explanation can be extended to a full proof which is given in Appendix A.4 (p. 31).

Example 7 (Illustration of Theorem 1). We illustrate the correctness of Theorem 1 based on the execution trace in Table 1. At step 0, since the last element in the trace is the initial state we can see that the output of function RGT is equal to the witness trace which is the initial state as well. At step 5, the output of function RGT is a sequence which consists of the witness of sequence $(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold}) \cdot ex_{12} \cdot (\perp, \perp, \mathit{free}, \perp) \cdot \beta_4 \cdot (\perp, \perp, \mathit{free}, \mathit{delivered})$ (i.e., $(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold}) \cdot ex_{12} \cdot (\mathit{done}, \mathit{done}, \mathit{free}, \mathit{delivered})$) followed by n_t. At this step, function RGT can not process partial states following interaction n_t, because the component involved in n_t is still busy.

5 Model Transformation

We propose a model transformation of a composite component $B^{\perp} = \gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$ such that it can produce the witness trace on-the-fly. The transformed system can be plugged to a runtime monitor as described in [FJN+15]. Our model transformation consists of three steps: 1) instrumentation of atomic components (Section 5.1), 2) construction of a new component (RGT) which implements Definition 8 (Section 5.2), 3) modification of interactions in $\gamma^{\perp}$ such that (i) component RGT can interact with the other components in the system and (ii) new interactions connect RGT to a runtime monitor (Section 5.3).

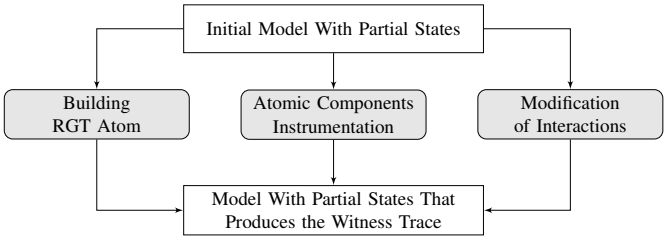
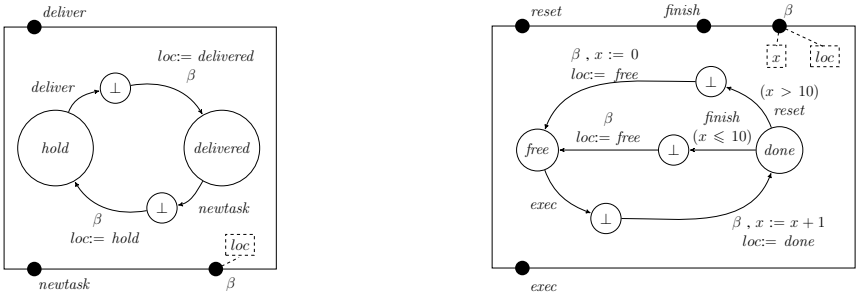


Fig. 7: Model transformation



(a) Instrumented component $Generator^r$ (b) Instrumented component $Worker^r$

Fig. 8: Instrumented atomic components of system Task

5.1 Instrumentation of Atomic Components

Given an atomic component with partial-state semantics as per Definition 5, we instrument this atomic component such that it is able to transfer its state through port β. The state of an instrumented component is delivered each time the component moves out from a busy location. In the following instrumentation, the state of a component is represented by the values of variables and the current location.

Definition 9 (Instrumenting an atomic component). *Given an atomic component in partial-state semantics* $B^\perp = (P \cup \{\beta\}, L \cup L^\perp, T^\perp, X)$ *with initial location* $l_0 \in L$*, we define a new component* $B^r = (P^r, L \cup L^\perp, T^r, X^r)$ *where:*

- $X^r = X \cup \{loc\}$*, loc is initialized to* l_0*;*
- $P^r = P \cup \{\beta^r\}$*, with* $\beta^r = \beta[X^r]$;
- $T^r = \{(l, p, g_\tau, [\,], l^\perp_\tau), (l^\perp_\tau, \beta, \texttt{true}, f_\tau; [loc := l'], l') \mid \{(l, p, g_\tau, [\,], l^\perp_\tau), (l^\perp_\tau, \beta, \texttt{true}, f_\tau, l')\} \subseteq T^\perp\}$.

In X^r, loc is a variable containing the current location. X^r is exported through port β. An assignment is added to the computation step of each transition to record the location.

Example 8 (Instrumenting an atomic component). Figure 8 shows the instrumented version of atomic components in system Task (depicted in Fig. 4).

- Figure 8a depicts component task generator, where $Generator^r.P^r = \{deliver[\emptyset], newtask[\emptyset], \beta[\{loc\}]\}$, $Generator^r.T^r = \{(hold, deliver, \texttt{true}, [\,], \perp), (\perp, \beta, \texttt{true}, [loc := delivered], delivered), (delivered, newtask, \texttt{true}, [\,], \perp), (\perp, \beta, \texttt{true}, [loc := hold], hold)\}$, $Generator^r.X^r = \{loc\}$.
- Figure 8b depicts a worker component, where $Worker^r.P^r = \{exec[\emptyset], finish[\emptyset], reset[\emptyset], \beta[\{x, loc\}]\}$, $Worker^r.T^r = \{(free, exec, \texttt{true}, [\,], \perp), (\perp, \beta, \texttt{true}, [x := x + 1; loc := done], done), (done, finish, (x \leqslant 10), [\,], \perp), (\perp, \beta, \texttt{true}, loc := free], free), (done, reset, (x > 10), [\,], \perp), (\perp, \beta, \texttt{true}, [x := 0; loc := free], free)\}$, $Worker^\perp.X^r = \{x, loc\}$.

5.2 Creating a New Atomic Component to Reconstruct Global States

Let us consider a composite component $B^\perp = \gamma^\perp(B^\perp_1, \ldots, B^\perp_n)$ with partial-state semantics, such that:

- $Init = (q^0_1, \ldots, q^0_n)$ is the initial state,

- γ is the set of interactions in the corresponding composite component with global-state semantics with $\gamma = \gamma^{\perp} \setminus \{\{\beta_i\}\}_{i=1}^{n}$, and
- the corresponding instrumented atomic components $B_1^r, \ldots, B_n^r$ have been obtained through Definition 9 such that B_i^r is the instrumented version of $B_i^{\perp}$.

We define a new atomic component, called RGT, which is in charge of accumulating the global states of the system $B^{\perp}$. Component RGT is an operational implementation, as a component of function RGT (Definition 8). At runtime, we represent a global state as a tuple consisting of the valuation of variables and the location for each atomic component. After a new interaction gets fired, component RGT builds a new tuple using the current states of components. Component RGT builds a sequence with the generated tuples. The stored tuples are updated each time the state of a component is updated. Following Definition 9, atomic components transfer their states through port β each time they move from a busy location to a ready location. RGT reconstructs global states from these received partial states and delivers them through the dedicated ports.

Definition 10 (RGT atom). *Component* RGT *is defined as* (P, L, T, X) *where:*

- $X = \bigcup_{i\in[1\,..\,n]}\{B_i^r.X^r\}\bigcup_{i\in[1\,..\,n]}\{B_i^r.X_c^r\} \cup \{gs_a \mid a \in \gamma\} \cup \{(z_1, \ldots, z_n)\} \cup \{V, v, m\}$*, where* $B_i^r.X_c^r$ *is a set containing a copy of the variables in* $B_i^r.X^r$*.*
- $P = \bigcup_{i\in[1\,..\,n]}\{\beta_i[B_i^r.X^r]\} \cup \{p_a[\emptyset] \mid a \in \gamma\} \cup \{p'_a[\bigcup_{i\in[1\,..\,n]}\{B_i^r.X_c\}] \mid a \in \gamma\}$.
- $L = \{l\}$ *is a set with one control location.*
- $T = T_{\text{new}} \cup T_{\text{upd}} \cup T_{\text{out}}$ *is the set of transitions, where:*
  - $T_{\text{new}} = \{(l, p_a, \bigwedge_{a\in\gamma}(\neg gs_a), \texttt{new}(a), l) \mid a \in \gamma\}$,
  - $T_{\text{upd}} = \{(l, \beta_i, \bigwedge_{a\in\gamma}(\neg gs_a), \texttt{upd}(i), l) \mid i \in [1\,..\,n]\}$,
  - $T_{\text{out}} = \{(l, p'_a, gs_a, \texttt{get}, l) \mid a \in \gamma\}$.

X is a set of variables that contains the following variables:

- the variables in $B_i^r.X^r$ for each instrumented atomic component B_i^r;
- a Boolean variable gs_a that holds `true` whenever a global state corresponding to interaction a is reconstructed;
- a tuple $(z_1, \ldots, z_n)$ of Boolean variables initialized to `false`;
- an $(n+1)$-tuple $v = (v_1, \ldots, v_n, v_{n+1})$.

For each $i \in [1\,..\,n]$, z_i is `true` when component i is in a busy location and `false` otherwise. For $i \in [1\,..\,n]$, v_i is a state of B_i^r and $v_{n+1} \in \gamma$. V is a sequence of $(n+1)$-tuples initialized to $(q_1^0, \ldots, q_n^0, -)$. m is an integer variable initialized to 1.

P is a set of ports.

- For each atomic component B_i^r for $i \in [1\,..\,n]$, RGT has a corresponding port β_i. States of components are exported to RGT through this port.
- For each interaction $a \in \gamma$, RGT has two corresponding ports p_a and p'_a. Port p_a is added to interaction a (later in Definition 11) in order to notify RGT when a new interaction is fired. A reconstructed global state which is related to the execution of interaction a, is exported to a runtime monitor through port p'_a.

RGT has three types of transitions:

- The transitions labeled by port p_a, for $a \in \gamma$, are in T_{new}. When no reconstructed global state can be delivered (that is, the Boolean variables in $\{gs_a \mid a \in \gamma\}$ are `false`), the transitions occur when the corresponding interaction a is fired.
- The transitions labeled by port β_i, for $i \in [1\,..\,n]$, are in T_{upd}. When no reconstructed global state can be delivered, to obtain the state of component $B_i^{\perp}$, these transitions occur at the same time transition β occurs in component $B_i^{\perp}$.
- The transition labeled by port p'_a for $a \in \gamma$ are in T_{get}. If RGT has a reconstructed global state corresponding to the global state of the system after executing interaction $a \in \gamma$, these transitions deliver the reconstructed global state to a runtime monitor.

RGT uses three algorithms.

Algorithm `new` (see Algorithm 1) implements the case of function acc that corresponds to the occurrence of a new interaction $a \in \gamma$ (Definition 8). It takes $a \in \gamma$ as input and then: 1) sets z_i to `true` if component i is involved in interaction a, for $i \in [1\,..\,n]$; 2) fills the elements of the $(n+1)$-tuple v with the states of components after the execution of the new interaction a in such a way that the i^{th} element of v corresponds to

Algorithm 1 $\texttt{new}(a)$

1: **for** $i = 1 \rightarrow n$ **do**
2: **if** $B_i.P \cap a \neq \emptyset$ **then** ▷ Check if component B_i is involved in interaction a.
3: $z_i := \texttt{true}$ ▷ In case component B_i is busy, z_i is `true`.
4: $v_i := \texttt{null}$ ▷ The i^{th} element of tuple v is represented by v_i.
5: **else**
6: $v_i := B_i^r.X^r$ ▷ v_i receives the state of B_i^r.
7: **end if**
8: **end for**
9: $v_{n+1} := a$ ▷ Last element of v receives interaction a.
10: $V := V \cdot v$ ▷ v is added to V.

the state of component $B_i^{\perp}$. Moreover, the state of busy components is `null`. The $(n+1)^{\text{th}}$ element of v is dedicated to interaction a, as a record specifying that tuple v is related to the execution of a; 3) appends v to V.

Algorithm `upd` (see Algorithm 2) implements the case of function acc which corresponds to the occurrence of transition β of atomic component $B_i^{\perp}$ for $i \in [1\,..\,n]$. According to Definition 9, the current state of the instrumented atomic component B_i^r for $i \in [1\,..\,n]$ is exported through port β of B_i^r. Algorithm `upd` takes the current state of B_i^r and looks into each element of V and replaces `null` values which correspond to B_i^r with the current state of B_i^r. Finally, algorithm `upd` invokes algorithm `check` to check the elements of V. If any tuple of V, associated to $a \in \gamma$, becomes a global state and has no `null` element, then the corresponding Boolean variable gs_a is set to `true`.

Algorithm 2 $\texttt{upd}(i)$

1: $z_i := \texttt{false}$
2: **for** $j = 1 \rightarrow \mathrm{length}(V)$ **do**
3: **if** $V(j)_i$ == `null` **then** ▷ The i^{th} element of the j^{th} tuple in V is represented by $V(j)_i$.
4: $V(j)_i := B_i^r.X^r$ ▷ Update the `null` states.
5: **end if**
6: **end for**
7: `check`() ▷ Check the elements of sequence V (cf. Algorithm. 3)

Algorithm 3 $\texttt{check}()$

1: **for** $i = m \rightarrow \mathrm{length}(V)$ **do** ▷ Check those tuples of V which have not been delivered to the monitor.
2: **if** $\neg gs_{(V(i)_{n+1})}$ **then**
3: $b_{\text{tmp}} := \texttt{true}$ ▷ Make a temporary boolean variable initialized to `true`.
4: **for** $j = 1 \rightarrow n$ **do**
5: $b_{\text{tmp}} := b_{\text{tmp}} \wedge (V(i)_j \neq \texttt{null})$ ▷ b_{tmp} remains `true` until a `null` is found in the i^{th} tuple of V.
6: **end for**
7: $gs_{(V(i)_{n+1})} := b_{\text{tmp}}$ ▷ Update the value of Boolean gs associated to $V(i)_{n+1}$.
8: **end if**
9: **end for**

Algorithm `get` (see Algorithm 4) is called whenever component RGT has a reconstructed global state to deliver. Algorithm `get` takes the m^{th} tuple in V and copies its values into $\{B_i^r.X_c^r\}_{i=1}^n$ and then increments m. Finally, algorithm `get` calls algorithm `check` in order to update the value of the Boolean variables gs_a for $a \in \gamma$, because there are possibly several reconstructed global states associated to an interaction $a \in \gamma$. In this case, after delivering one of those reconstructed global states and resetting gs_a to `false`, one must again set variable gs_a to `true` for the rest of the reconstructed global states associated to interaction a. Note, to facilitate the presentation of proofs in Appendix A, component RGT is defined in such a way that it does not discard the reconstructed global states of the system after delivering them to the monitor. In our actual implementation of RGT, these states are discarded because they are not useful after being delivered to the

Algorithm 4 `get()`

1: **for** $i = 1 \rightarrow n$ **do**
2: $\quad B_i^r.X_c^r := V(m)_i$ ▷ Copy the m^{th} tuple of V.
3: **end for**
4: $gs_{(V(m)_{n+1})} :=$ `false` ▷ Reset the corresponding gs_a of the $V(m)$.
5: $m := m + 1$ ▷ Increment m.
6: `check()` ▷ Check the elements of sequence V (cf. Algorithm. 3)

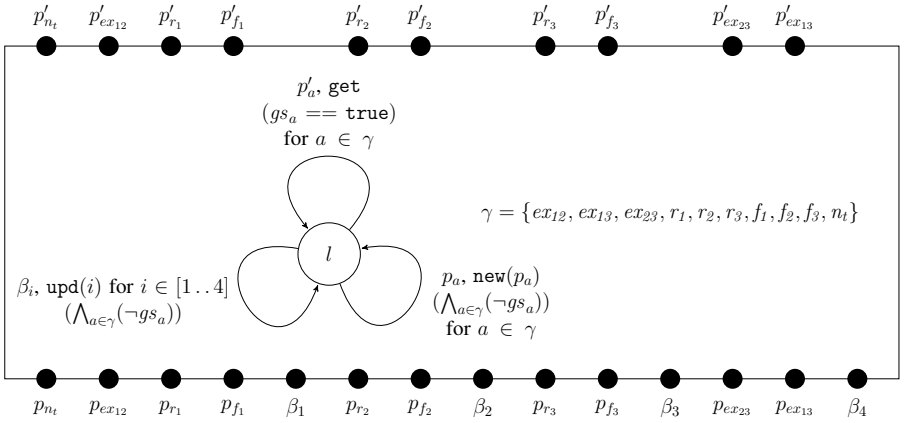


Fig. 9: Component RGT for system Task

monitor. At runtime, RGT $.V$ contains the sequence of global states associated with the witness trace (as stated later by Proposition 2).

Example 9 (Component RGT*).* Figure 9 depicts the component RGT for system Task. For readability, only one instance of each type of transitions is shown. The execution of a new interaction $a \in \{ex_{12}, ex_{13}, ex_{23}, r_1, r_2, r_3, f_1, f_2, f_3, n_t\}$ in system Task is synchronized with the execution of transition p_a of the component RGT which applies the algorithm `new`. Each busy interaction in the system Task is synchronized with the execution of transition β_i ($i \in [1..4]$ are the indexes of the four components in system Task) which applies the algorithm upd to update the reconstructed states so far and check whether or not a new global state is reconstructed. Transition β_i, $i \in [1..4]$, is guarded by $\bigwedge_{a\in\gamma}(\neg gs_a)$ which ensures the delivery of the new reconstructed global state through the ports $p_{a\in\gamma}$ as soon as they are reconstructed. At runtime, RGT produces the sequence of global states in the right-most column of Table 1.

5.3 Connections

After building component RGT (see Definition 10), and instrumenting atomic components (see Definition 9), we modify all interactions and define new interactions to build a new transformed composite component. To let RGT accumulate states of the system, first we transform all the existing interactions by adding a new port to communicate with component RGT, then we create new interactions that allow RGT to deliver the reconstructed global states of the system to a runtime monitor.

Given a composite component $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ with corresponding component RGT and instrumented components $B^r = (P \cup \{\beta^r\}, L \cup L^\perp, T^r, X^r)$ such that $B^r = B_i^r \in \{B_1^r, \ldots, B_n^r\}$, we define a new composite component.

Definition 11 (Composite component transformation). *For a composite component $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$, we introduce a corresponding transformed component $B^r = \gamma^r(B_1^r, \ldots, B_n^r, RGT)$ such that $\gamma^r = a_\gamma^r \cup a_\beta^r \cup a^m$ where:*

– *a_γ^r and a_β^r are the sets of transformed interactions such that:*

$$\forall a \in \gamma^\perp \,.\, a^r = \begin{cases} a \cup \{RGT.p_a\} & \text{if } a \in \gamma \\ a \cup \{RGT.\beta_i\} & \text{otherwise } (a \in \{\{\beta_i\}\}_{i\in[1\,..\,n]}) \end{cases}$$

$$a_\gamma^r = \{a^r \mid a \in \gamma\},\ a_\beta^r = \{a^r \mid a \in \{\{\beta_i\}\}_{i\in[1\,..\,n]}\}$$

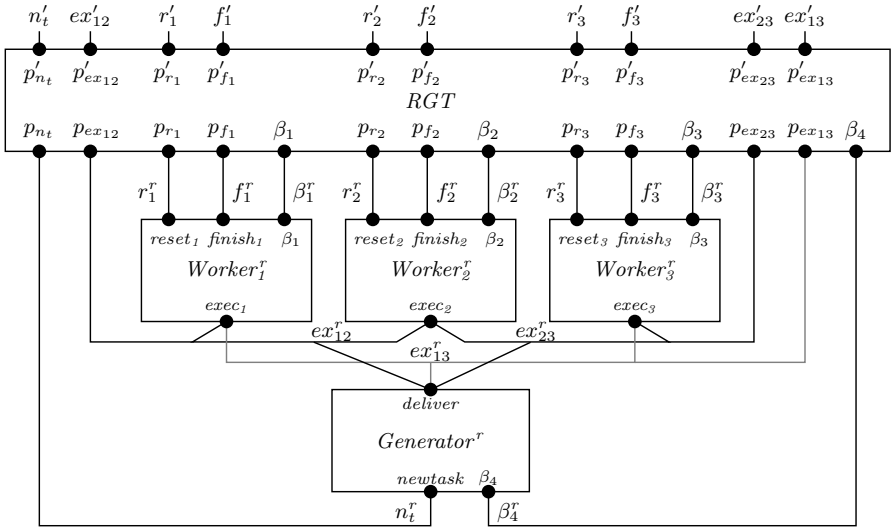


Fig. 10: Composite component of system Task obtained by applying the transformation in Definition 11

– a^m *is a set of new interactions such that:*
$a^m = \{a' \mid a \in \gamma\}$ *where* $\forall a \in \gamma \,.\, a' = \{RGT.p'_a\}$ *is a corresponding unary interaction.*

For each interaction $a \in \gamma^{\perp}$, we associate a transformed interaction a^r which is the modified version of interaction a such that a corresponding port of component RGT is added to a. Instrumenting interaction $a \in \gamma$ does not modify sequence of assignment F_a, whereas instrumenting busy interactions $a \in \{\{\beta_i\}\}_{i=1}^n$ adds assignments to transfer attached variables of port β_i to the component RGT. The transformed interactions belong to two subsets, a^r_γ and a^r_β. The set a^m is the set of all unary interactions a' associated to each existing interaction $a \in \gamma$ in the system.

The set of the states of transformed composite component B^r is $Q^r = B_1^r.Q \times \ldots \times B_n^r.Q \times RGT.Q$.

Example 10 (Transformed composite component). Figure 10 shows the transformed composite component of system Task. The goal of building a' for each interaction a is to enable RGT to connect to a runtime monitor. Upon the reconstruction of a global state corresponding to interaction $a \in \gamma$, the corresponding interaction a' delivers the reconstructed global state to a runtime monitor.

5.4 Correctness of the Transformations

Combined together, the transformations preserve the semantics of the initial model as stated in the rest of this section.

Intuitively, the component RGT defined in Definition 10 implements function RGT defined in Definition 8. Reconstructed global states can be transferred through the ports p'_a with $a \in \gamma$. If interaction a happens before interaction b, then in component RGT, port p'_a which contains the reconstructed global state after executing a will be enabled before port p'_b. In other words, the total order between executed interactions is preserved.

In the transformed composite component $\gamma^r(B_1^r, \ldots, B_n^r, RGT)$, the notion of equivalence is used to relate the tuples constructed by component RGT to the states of the initial system in partial-state semantics. Below, we define the notion of equivalence between an $(n+1)$-tuple $v = (v_1, \ldots, v_n, v_{n+1})$ and a state of the system $q = (q_1, \ldots, q_n)$ such that, for $i \in [1\,..\,n]$, v_i is a state of B_i^r and $v_{n+1} \in \gamma$.

Definition 12 (Equivalence of an $(n+1)$-tuple and a state). *An $(n+1)$-tuple $v = (v_1, \ldots, v_n, v_{n+1})$ is equivalent to a state $q = (q_1, \ldots, q_n)$ if:*

$$\forall i \in [1\,..\,n] \,.\, v_i = \begin{cases} q_i & \text{if } q_i \in Q_i, \\ \texttt{null} & \text{otherwise.} \end{cases}$$

When an $(n+1)$-tuple v is equivalent to a state q, we denote it by $v \cong q$.

A tuple $(v_1, \ldots, v_n, v_{n+1})$ and a state $(q_1, \ldots, q_n)$ are equivalent if $v_i = q_i$ for each position i where the state q_i of component B_i^r is also a state of the initial model, and $v_i = \texttt{null}$ otherwise. The notion of equivalence is extended to traces and sequences of $(n+1)$-tuples. A trace $\sigma = q'_0.a_1.q'_1 \ldots a_k.q'_k$ and a sequence of $(n+1)$-tuples $V = v(0) \cdot v(1) \ldots v(k)$ are equivalent, denoted $\sigma \cong V$, if q'_j is equivalent to $v(j)$ for all $j \in [0\,..\,k]$ and $v(j)_{n+1} = a_j$ for all $j \in [1\,..\,k]$.

Proposition 2 (Correctness of component RGT**).** $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\,\mathrm{RGT}\,.V \cong \mathrm{acc}(\sigma)$.

Proposition 2 states that, for any trace σ, at any time, variable $\mathrm{RGT}\,.V$ encodes the witness trace $\mathrm{acc}(\sigma)$ of the current trace: $\mathrm{RGT}\,.V$ is a sequence of tuples where each tuple consists of the state and the interaction that led to this state, in the same order as they appear on the witness trace.

Proof. The proof is done by induction on the length of $\sigma \in \mathrm{Tr}(B^{\perp})$, i.e., the trace of the system in partial-state semantics. The proof is given in Appendix A.5 (p. 32).

For each trace resulting from an execution with partial-state semantics, component RGT produces a trace of global states which is the witness of this trace in the initial model.

Definition 13 (State stability). *State $(l, v) \in RGT.Q$ is said to be* stable *when* $\forall x \in \{\mathrm{RGT}\,.gs_a \mid a \in \gamma\}\,.\,v(x) = \texttt{false}$.

A state q in the the semantics of atomic component RGT is said to be stable when all Boolean variables in set $\{\mathrm{RGT}\,.gs_a \mid a \in \gamma\}$ evaluate to `false` with the valuation of variables in state q. In other words, the current state of component RGT is stable when it has no reconstructed global states to deliver. We say that the composite component B^r is stable when the state of its associated component RGT is stable.

Example 11 (Stable state). We illustrate Definition 13 based on the execution trace in Table 1. By the evolution of system Task from step 4 to step 5, component RGT reconstructs the global state associated to the execution of ex_{12} and respectively sets boolean variable $gs_{ex_{12}}$ to `true`. Once $gs_{ex_{12}}$ becomes `true`, we say that the state of the component RGT is not stable. In component RGT, the execution of transition labeled by port $p'_{ex_{12}}$ delivers the reconstructed global state (i.e., $(done, done, free, delivered)$) to the monitor and sets boolean variable $gs_{ex_{12}}$ to `false`. Consequently, component RGT becomes stable. We say that component RGT is not stable whenever there exists at least one reconstructed global state which has not been delivered to the monitor. Whenever component RGT is not stable, we say that the system is not stable as well.

The following lemma states a property of the algorithms in Section 5.2 ensuring that whenever component RGT has reconstructed some global states, it transmits them to the monitor before the system can execute any new partial state can be created.

Lemma 5. *In any state of the transformed system, if there is a non-empty set $GS \subseteq \{\mathrm{RGT}\,.gs_a \mid a \in \gamma\}$ in which all variables are* `true`*, the variables in $\{\mathrm{RGT}\,.gs_a \mid a \in \gamma\} \setminus GS$ cannot be set to* `true` *until all variables in GS are reset to* `false` *first.*

The following lemma states that any state of the composite component B^r can be stabilized by executing interactions in a^m.

Lemma 6. *We shall prove that for any state $q \in Q^r$, there exists a state $q' \in Q^r$ reached after interactions in a^m (i.e., $q \xrightarrow{(a^m)^*} q'$), such that q' is a stable state (i.e.,* $\mathrm{stable}(q')$*).*

We define a notion of equivalence between states of the transformed model and states of the initial system.

Definition 14 (Equivalent states). *Let $q^r = (q_1^r, \cdots, q_n^r, q_{n+1}^r) \in Q^r$ be a state in the transformed model where q_{n+1}^r is the state of component* RGT*, function* $\mathrm{equ} : Q^r \longrightarrow Q^{\perp}$ *is defined as follows:* $\mathrm{equ}(q^r) = q$*, where* $q = (q_1, \cdots, q_n)$*,* $(\forall i \in [1\,..\,n]\,.\,q_i^r = q_i) \wedge \mathrm{stable}(q_{n+1}^r)$.

A state in the initial model is said to be equivalent to a state in the transformed model if the state of each component in the initial model is equal to the state of the corresponding component in transformed model and the state of component RGT is stable.

The following lemma is a direct consequence of Definition 14. The lemma states that, if an interaction is enabled in the transformed model, then the corresponding interaction is enabled in the initial model when the states of two models are equivalent.

Lemma 7. *For any two equivalent states $q \in Q^{\perp}$ and $q^r \in Q^r$ (i.e.,* $\mathrm{equ}(q^r) = q$*), if interaction $a \in \gamma^{\perp}$ is enabled in state q, then $a^r \in \gamma^r$ is enabled at state q^r.*

Based on the above lemmas, we can now state the correctness of our transformations.

Theorem 2 (Transformation Correctness). $\gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp}) \sim \gamma^{r}(B_1^{r}, \ldots, B_n^{r}, RGT)$.

Theorem 2 states that the initial model and the transformed model are observationally equivalent.

Proof. The proof relies on exhibiting a bi-simulation relation between the set of states of $B^r = \gamma^r(B_1^r, \ldots, B_n^r, RGT)$, that is Q^r, and the set of states of $B = \gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$, that is $Q^{\perp}$. The proof is given in Appendix A.7 (p. 33).

Combined together, Theorem 2 and Lemma 7 imply that, for each state in the initial system, there exists an equivalent state in the transformed system in which all enabled interactions in the initial system are also enabled in the transformed system. Hence, we can conclude that the transformed system is as concurrent as the initial system.

Consequently, we can substantiate our claims stated in the introduction about the transformations: instrumenting atomic components and adding component RGT (i) preserves the semantics and concurrency of the initial model, and (ii) verdicts are sound and complete.

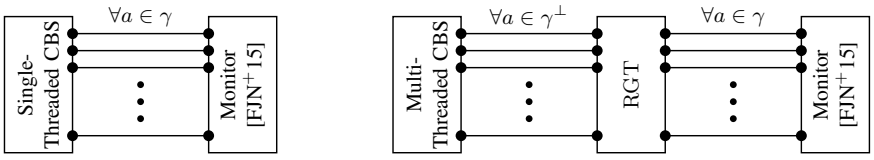
Remark 3 (Alternative RGT *atoms).* In the definition of atom RGT (Definition 10), one can observe that whenever component RGT has reconstructed global states to deliver, the system cannot proceed and must wait until all the reconstructed global states are sent (because of the guards of transitions T_{upd} and T_{new}). This gives precedence to monitoring rather than to the evolution of the system.

Three alternative definitions of RGT can be considered by changing the guards of the transitions in T_{new} and T_{upd}. For both transitions, by suppressing the guards, one gives less precedence to the transmission of reconstructed global states. By suppressing the guards in transitions in T_{new}, we let the system starting a new interaction while there may be still some reconstructed global states for RGT to deliver. By suppressing the guards in transitions in T_{upd}, we let the system execute β-transitions while there may be still some reconstructed global states for RGT to deliver.

Suppressing these guards favors the performance of the system but may delay the transmission of global states to the monitor and thus it may also delay the emission of verdicts. There is thus a tradeoff between the performance of the system and the emission of verdicts.

5.5 Monitoring

As it is shown in Fig. 11, one can reuse the results in [FJN+15] to monitor a system with partial-state semantics. One just has to transform this system with the previous transformations and plug a monitor for a property on the global-states of the system to component RGT through the dedicated ports. At runtime, such monitor will (i) receive the sequence of reconstructed global states corresponding to the witness trace, (ii) preserve the concurrency of the system, and iii) state verdicts on the witness trace.



(a) Monitoring of single-threaded CBSs (b) Monitoring of multi-threaded CBSs

Fig. 11: Abstract view of runtime monitoring of single-threaded vs. multi-threaded CBSs

Example 12 (Monitoring system Task). Figure 12 depicts the transformed system Task with a monitor (for the homogeneous distribution of the tasks among the workers) where e_1, e_2, and e_3 are events related to the pairwise comparison of the number of executed tasks by *Worker*s. For $i \in [1\,..\,3]$, event e_i evaluates to true whenever $|x_{(i \bmod 3)+1} - x_i|$ is lower than 3 (for this example). Component *Monitor* evaluates $(e_1 \wedge e_2 \wedge e_3)$ upon the reception of a new global state from RGT and emits the associated verdict till reaching bad state $\perp$. The global trace $(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold}) \cdot ex_{12} \cdot (\mathit{done}, \mathit{done}, \mathit{free}, \mathit{delivered}) \cdot n_t$ (see Table 1) is sent by component RGT to the monitor which in turn produces the sequence of verdicts $\top_c \cdot \top_c$ (where $\top_c$ is verdict "currently good", see [BLS10,FFM12]).

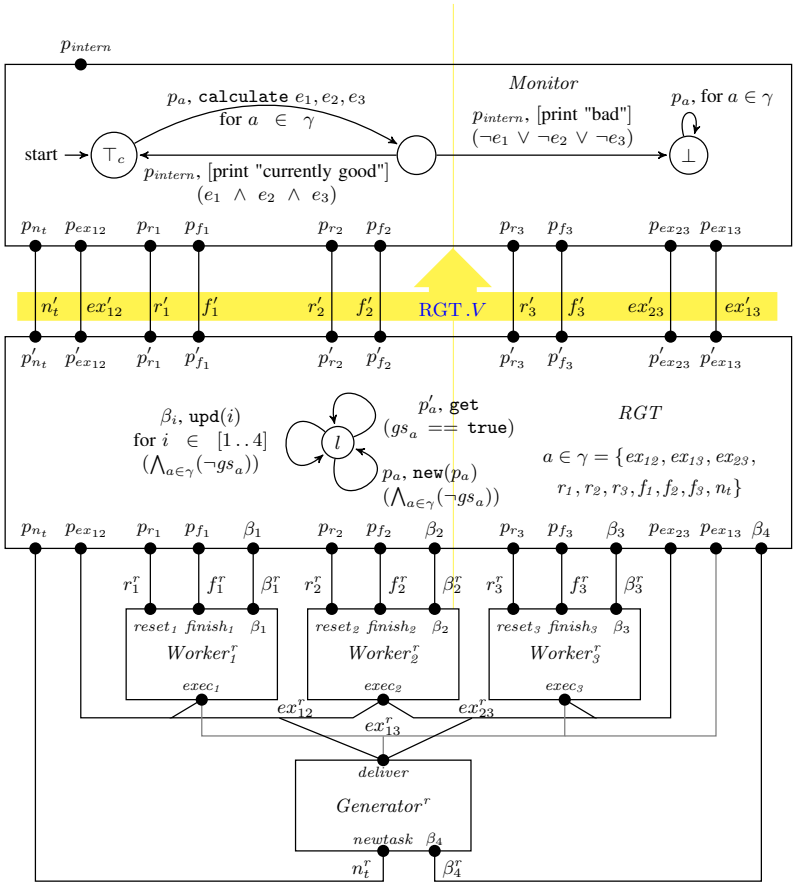


Fig. 12: Monitored version of system Task

6 Implementation and Performance Evaluation

We present an evaluation of our monitoring approach implemented in a tool called RVMT-BIP. RVMT-BIP is a prototype tool implementing the algorithms presented in Section 5.

This section is organized as follows. In Section 6.1, we present the architecture of RVMT-BIP. In Section 6.2, we present the systems and properties used in our case studies. We experiment with RVMT-BIP on four systems where each system is monitored against dedicated properties. In Section 6.3, we present the evaluation principles. In Section 6.4, we present the experimental results and discuss the performance of RVMT-BIP.

6.1 Architecture of RVMT-BIP

RVMT-BIP (Runtime Verification of Multi-Threaded BIP) is a Java implementation of ca. 2,200 LOC. RVMT-BIP is integrated in the BIP tool suite [BBS06]. The BIP (Behavior, Interaction, Priority) framework is a powerful and expressive framework for the formal construction of heterogeneous systems. RVMT-BIP takes as input a BIP CBS and a monitor description for a property, and outputs a new BIP system whose behavior is monitored against the property while running concurrently. RVMT-BIP uses the following modules:

– Module *Atomic Transformation* takes as input the initial BIP system and a monitor description. From the input abstract monitor description, it extracts the list of components, and the set of their states and variables that influence the truth-value of the property and are used by the monitor. Then, this module instruments the atomic components in the extracted list so as to observe their states and the values of the variables. Finally, the transformed components and the original version of the components that do not influence the property are returned as output.

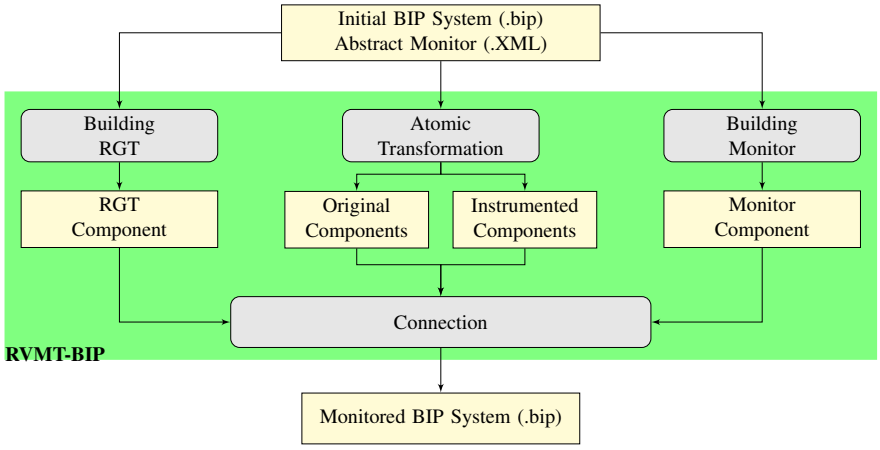


Fig. 13: Overview of RVMT-BIP work-flow

- Module *Building* RGT takes as input the initial BIP system and a monitor description and produces component RGT (Reconstructor of Global Trace) which reconstructs and accumulates global states at runtime to produce "on-the-fly" the global trace.
- Module *Building Monitor* takes as input the initial BIP system and a monitor description and then outputs the atomic component implementing the monitor (following [FJN+15]). Component *Monitor* receives and consumes the reconstructed global trace generated by component RGT at runtime and emits verdicts.
- Module *Connections* constructs the new composite and monitored component. The module takes as input the output of the *Atomic Transformation*, *Building* RGT and *Building Monitor* modules and then outputs a new composite component with new connections. The new connections are purposed to synchronize instrumented components and component RGT in order to transfer updated states of the components to RGT. Instrumented components interact with RGT independently and concurrently.

6.2 Case Studies

We present some case studies on executable BIP systems conducted with RVMT-BIP.

Process Completion of System Demosaicing Demosaicing is an algorithm for digital image processing used to reconstruct a full color image from the incomplete color samples output from an image sensor. Figure 14 shows a simplified version of the the processing network of Demosaicing. Demosaicing contains a *Splitter* and a *Joiner* process, a pre-demosaicing (*Demopre*) and a post-demosaicing (*Demopost*) process and three internal demosaicing *Demo* processes that run in parallel. The real model contains ca. 1,000 lines of code, consists of 26 atomic components interacting through 35 interactions. We consider two specifications related to process completion:

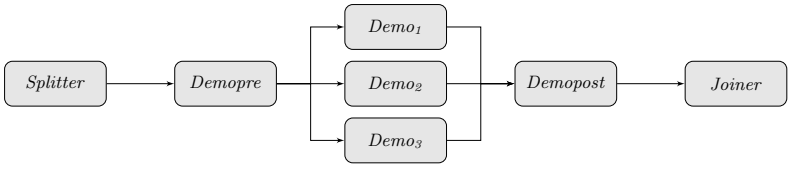
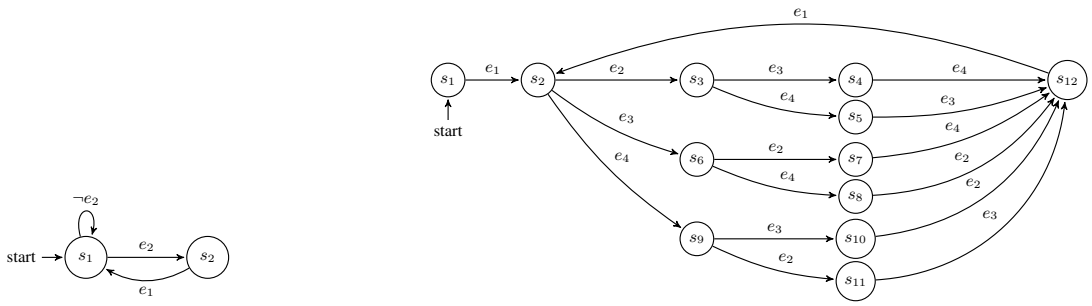


Fig. 14: Processing network of system Demosaicing

φ_1: It is necessary that all the internal demosaicing units finish their process before the post-demosaicing unit starts processing. The post-demosaicing unit receives the output results of internal demosaicing units through port *getimg*. We add variable *port* to record the last executed port. Each demosaicing unit has a boolean variable *done* which is set to `true` whenever the demosaicing process completes. This requirement is formalized as property φ_1 defined by the automaton depicted in Fig. 15a where the events are e_1 :

$Demopost.port == getimg$ and $e_2 : (Demo_1.done \wedge Demo_2.done \wedge Demo_3.done)$. From the initial state s_1, the automaton moves to state s_2 when all the internal demosaicing units finish their process. Receiving the processed images by post-demosaicing causes a move from state s_2 to s_1.

φ_2: Moreover, internal demosaicing units ($Demo_1$, $Demo_2$, $Demo_3$) should not start the demosaicing process until the pre-demosaicing unit finishes its process. The pre-demosaicing unit sends its output to the internal demosaicing units through port $transmit$ and each internal demosaicing unit starts the demosaicing process by executing a transition labeled by port $start$. This requirement is formalized as property φ_2 which is defined by the automaton depicted in Fig. 15b where $e_1 : Demopre.port == transmit$, $e_2 : Demo_1.port == start$, $e_3 : Demo_2.port == start$ and $e_4 : Demo_3.port == start$. From the initial state s_1, whenever the pre-demosaicing unit transmits its processed output to the internal demosaicing units, the automaton moves to state s_2. Internal demosaicing units can start in different order. Moreover, all demosaicing units must eventually start their internal process and the automaton reaches state s_{12}. From state s_{12}, the automaton moves back to state s_2 whenever the pre-demosaicing unit sends the next processed data to the internal demosaicing units.



(a) φ_1 (b) φ_2

Fig. 15: Automata of properties of demosaicing

Data-freshness of System Reader-WriterV1 System Reader-WriterV1 (ca. 130 LOC) consists of a set of independent composite components. Each composite component consists of four components: a $Reader$, a $Writer$, a $Clock$ and a $Poster$. (in total, 12 components and 9 interactions). $Reader$ and $Writer$ communicate with each other through the $Poster$. The data generated by $Writer$ is written in a $Poster$ that can be accessed by $Reader$. The Reader-Writer model is presented in Fig. 16. We consider a specifications related to data freshness:

φ_3: It is necessary that the data is up-to-date: the data read by component $Reader$ must be fresh enough compared to the moment it has been written by $Writer$. If t_1 and t_2 are the moments of reading and writing actions respectively, then the difference between t_2 and t_1 must be less than a specific duration δ, i.e., $(t_2 - t_1) \leq \delta$. In the model, the time counter is implemented by a component $Clock$, and the $tick$ transition occurs every second. This requirement is formalized as property φ_3 which is defined by the automaton depicted in Fig. 17a, where $\delta = 2$, $e_1 : Writer.port == write$, $e_2 : Clock.port == tick$ and $e_3 : Reader.port == read$. Whenever $Writer$ writes into $Poster$, the automaton moves from the initial state s_1 to s_2. When $Reader$ reads $Poster$, the automaton moves from s_2 to s_1. $Reader$ is allowed to read $Poster$ after one $tick$ transition. In this case, the automaton moves from s_2 to s_3 after the $tick$, and then moves from s_3 to s_1 after reading $Poster$. φ_3 also allows to read $Poster$ after two $tick$ transitions. In this case, the automaton moves from s_2 to s_4 after the first $tick$, then moves from s_4 to s_3 on the second $tick$, and finally moves from s_3 to s_1 after reading $Poster$.

Execution Order of System Reader-WriterV2 System Reader-WriterV2 (ca. 150 LOC) is a more complex version of Reader-WriterV1 and involves several writers. This system has six components: $Reader$, $Writer_1$, $Writer_2$, $Writer_3$, $Clock$ and $Poster$. The Writers are synchronized together. $Reader$ and Writers communicate with each other through $Poster$. The data generated by each writer is written to $Poster$ and can then

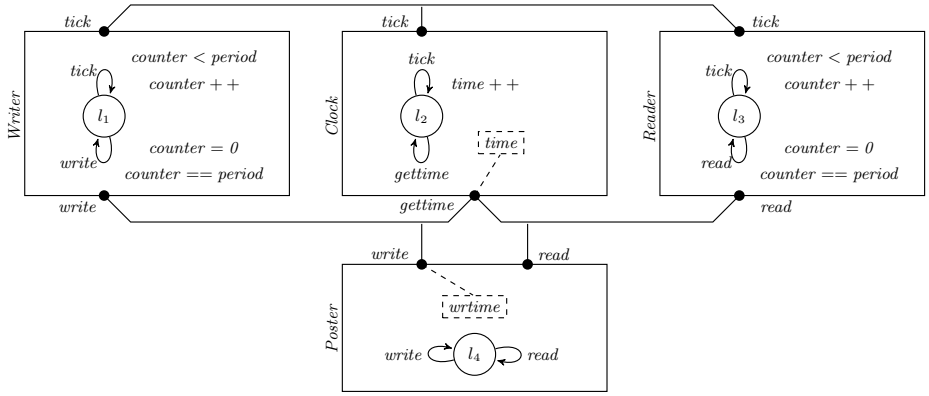
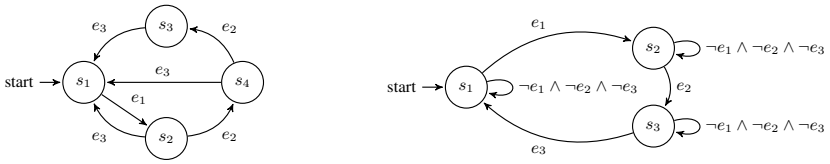


Fig. 16: Model of system Reader-Writer



(a) φ_3 (b) φ_4

Fig. 17: Automata of the properties of system Reader-Writer

be accessed by $Reader$. Having several writers, a more complex specification on the execution order can be defined. We consider a specifications related to execution order:

φ_4: The writers should periodically write data to a poster in a specific order. The specification concerns 3 writers: $Writer_1$, $Writer_2$ and $Writer_3$. During each period , the writing order must be as follows: $Writer_1$ writes to the poster first, then $Writer_2$ can write only when $Writer_1$ finishes writing to the poster, $Writer_3$ can write only when $Writer_2$ finishes writing to the poster, and the same goes on for the next periods. To do so, each writer is assigned a unique id that is passed to the poster when it starts using the poster. This id is then used to determine the last writer that used the poster. For example, when $Writer_2$ wants to access the poster, it has to check whether the id stored in the poster corresponds to $Writer_1$ or not.
This requirement is formalized as property φ_4 which is defined by the automaton depicted in Fig. 17b where:

- $e_1 : (Writer_1.port == write \wedge Poster.port == write \wedge Clock.port == getTime)$,
- $e_2 : (Writer_2.port == write \wedge Poster.port == write \wedge Clock.port == getTime)$,
- $e_3 : (Writer_3.port == write \wedge Poster.port == write \wedge Clock.port == getTime)$.

When $Writer_1$ writes to the poster, the automaton moves from initial state s_1 to state s_2. From state s_2, the automaton moves to state s_3 when $Writer_2$ writes to the poster. From state s_3, the automaton moves to the initial state s_1 when $Writer_3$ writes to the poster. This writing order must always be followed.

Distribution of Tasks in System Task We consider our running example system Task and a specification of the homogeneous distribution of the tasks among the workers:

φ_5: The satisfaction of this specification depends on the execution time of each worker. Different tasks may have different execution times for different workers. Obviously, the faster a worker completes each task, the higher is the number of its accomplished tasks. After executing a task, the value of the variable x of a worker is increased by one. Moreover, the absolute difference between the values of variable x of any two workers must always be less than a specific integer value (which is 3 for this case study). This requirement is formalized as property φ_5 which is defined by the automaton depicted in Fig. 18 where $e_1 : |worker_1.x - worker_2.x| < 3$, $e_2 : |worker_2.x - worker_3.x| < 3$ and $e_3 : |worker_1.x - worker_3.x| < 3$. The property holds as long as e_1, e_2 and e_3 hold.

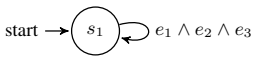


Fig. 18: Automaton of the property of system Task

Table 2: Results of monitoring Demosaicing, Reader-WriterV1, Reader-WriterV2, and Task with RVMT-BIP

system	# executed interactions	execution time and overhead according to the number of threads										# events	# extra executed interactions
		1	2	3	4	5	6	7	8	9	10		
Demosaicing (26,35)	1,300	18.98	10.24	7.75	6.85	6.58	6.09	6.33	6.45	6.29	6.27	n/a	n/a
Demosaicing (27,69)	3,051	19.02	11.53	8.17	7.43	6.68	6.50	6.27	6.05	6.03	6.18	1,300	1,751
φ_1 (11)		0.1%	12.6%	5.4%	8.5%	4.3%	6.6%	< 0.1%	< 0.1%	< 0.1%	< 0.1%		
Demosaicing (27,46)	1,850	18.68	11.05	7.65	7.80	6.77	6.38	6.22	6.45	6.17	6.35	400	550
φ_2 (4)		< 0.1%	7.9%	< 0.1%	13.8%	2.8%	4.8%	< 0.1%	< 0.1%	< 0.1%	< 0.1%		
Reader-WriterV1 (12,9)	120,000	61.48	29.67	20.03	20.00	20.05	20.21	20.60	21.54	21.92	22.13	n/a	n/a
ReaderWriterV1 (13,12)	200,000	62.53	38.29	21.96	22.28	22.62	22.71	22.88	23.48	24.15	24.47	40,000	80,000
φ_3 (3)		1.6%	27.7%	9.6%	11.4%	12.8%	12.4%	11.0%	9.0%	10.1%	10.5%		
Reader-WriterV2 (6,7)	20,000	32.06	21.45	12.04	11.37	11.33	11.37	11.44	11.49	11.53	11.58	n/a	n/a
ReaderWriterV2 (7,12)	85,000	33.92	22.72	13.90	13.77	14.09	14.36	14.83	15.18	15.41	15.57	20,000	65,000
φ_4 (5)		5.8%	5.9%	15.4%	21.1%	24.3%	26.2%	29.6%	32.1%	33.5%	34.4%		
Task (4,10)	399,999	117.28	70.18	60.91	60.06	58.98	60.01	60.93	61.77	63.13	65.45	n/a	n/a
Task (5,16)	600,197	123.98	71.73	62.28	63.26	62.79	62.78	63.35	64.57	65.61	66.27	100,198	200,198
φ_5 (3)		5.7%	2.2%	2.2%	5.3%	6.4%	4.4%	3.9%	4.5%	3.9%	1.2%		

6.3 Evaluation Principles

For each system and all its properties, we synthesized a BIP monitor following [FJN+11,FJN+15] and combined it with the CBS output from RVMT-BIP. We obtain a new CBS with corresponding RGT and monitor components. We run each system by using various numbers of threads and observe the execution time. Executing these systems with a multi-threaded controller results in a faster run because the systems benefit from the parallel threads. Additional steps are introduced in the concurrent transitions of the system. Note, these are asynchronous with the existing interactions and can be executed in parallel. These systems can also execute with a single-threaded controller which forces them to run sequentially. Varying the number of threads allows us to assess the performance of the (monitored) system under different degrees of parallelism. In particular, we expected the induced overhead to be insensitive to the degree of parallelism. For instance, an undesirable behavior would have been to observe a performance degradation (and an overhead increase) which would mean either that the monitor sequentializes the execution or that the monitoring infrastructure is not suitable for multi-threaded systems. We also extensively tested the functional correctness of RVMT-BIP, that is whether the verdicts of the monitors are sound and complete.

6.4 Results and Conclusions

Performance evaluation. Tables 2 and 3 report the timings obtained when checking the following specifications: *complete process* property on Demosaicing, *data freshness* and *execution ordering* property on Reader-Writer systems, and *task distribution* property on Task, with RVMT-BIP and RV-BIP respectively. Each measurement is an average value obtained over 100 executions of these systems. In Tables 2 and 3, the columns have the following meanings:

- Column *system* indicates the systems. System in *italic* format represents the monitored version of the initial system. Moreover, (x, y) in front of the system name means that x (resp. y) is the number of components (resp. interactions) of the system. The monitored property is written below each monitored system name with a value (z) which indicates that z components have variables influencing the truth-value of the property (and were thus instrumented by RVMT-BIP or RV-BIP).
- Column *# executed interactions* indicates the number of interactions executed by the engine which also represents the number of functional steps of the system.
- Columns *execution time and overhead according to the number of threads* report (i) the execution time of the systems when varying the number of threads and (ii) the overhead induced by monitoring (for monitored systems).

Table 3: Results of monitoring Demosaicing, Reader-WriterV1, Reader-WriterV2, and Task with RV-BIP

system	# executed interactions	execution time and overhead w.r.t. different number of threads										# events	# extra executed interactions
		1	2	3	4	5	6	7	8	9	10		
Demosaicing (26,35)	1,300	18.98	10.24	7.75	6.85	6.58	6.09	6.33	6.45	6.29	6.27	n/a	n/a
Demosaicing (27,37)	2,450	19.66	27.34	32.28	32.61	33.03	32.23	31.17	31.24	31.22	31.81	1,300	1,300
φ_1 *(11)*		3.5%	167%	316%	376%	402%	429%	392%	384%	369%	407%		
Demosaicing (27,37)	1,700	19.50	14.79	13.87	13.11	13.13	12.75	11.18	11.34	11.19	11.16	400	400
φ_2 *(11)*		2.7%	44.4%	78.8%	91.4%	99.7%	109%	76.5%	75.7%	78.0%	78.0%		
Reader-WriterV1 (12,9)	120,000	61.48	29.67	20.03	20.00	20.05	20.21	20.60	21.54	21.92	22.13	n/a	n/a
Reader-WriterV1 (13,11)	1600,000	61.97	37.77	21.94	22.13	22.62	23.14	25.09	26.21	26.73	27.18	40,000	40,000
φ_3 *(3)*		0.8%	26.0%	9.5%	10.6%	12.8%	14.5%	21.8%	21.7%	21.9%	22.7%		
Reader-WriterV2 (6,7)	20,000	32.06	21.45	12.04	11.37	11.33	11.37	11.44	11.49	11.53	11.58	n/a	n/a
Reader-WriterV2 (7,9)	40,000	33.11	23.80	13.31	13.32	13.37	13.82	14.28	14.35	14.79	14.96	20,000	20,000
φ_4 *(5)*		3.2%	10.9%	10.5%	17.1%	18.0%	21.5%	24.8%	24.8%	28.2%	29.2%		
Task (4,10)	399,999	117.28	70.18	60.91	60.06	58.98	60.01	60.93	61.77	63.13	65.45	n/a	n/a
Task (5,12)	500,197	121.61	70.12	72.25	75.11	75.66	80.54	81.62	84.58	89.65	90.21	100,198	100,198
φ_5 *(3)*		3.6%	$< 0.1\%$	18.6%	25.0%	28.2%	34.0%	33.9%	36.9%	42.01%	37.8%		

– Column *events* indicates the number of reconstructed global states (events sent to the associated monitor).
– Column *extra executed interactions* reports the number of additional interactions (i.e., execution of interactions which are added into the initial system for monitoring purposes).

As shown in Table 2, using more threads reduces significantly the execution time in both the initial and transformed systems. Comparing the overheads according to the number of threads shows that the proposed monitoring technique (i) does not restrict the performance of parallel execution and (ii) scales up well with the number of threads.

Performance comparison of RV-BIP and RVMT-BIP. To illustrate the advantages of monitoring multi-threaded systems with RVMT-BIP, we compared the performance of RVMT-BIP and RV-BIP ([FJN+15]); see Tables 2 and 3 for the results. Monitoring with RV-BIP amounts to use a standard runtime verification technique, i.e., not tailored to multi-threaded systems. At runtime, the RV-BIP monitor consumes the global trace (i.e., sequence of global states) of the system (where global snapshots are obtained by synchronization among the components) and yields verdicts regarding property satisfaction. It has been shown in [FJN+15] that RV-BIP efficiently handles CBSs with sequential executions.

In the following, we highlight some of the main observations and draw conclusions:

1. Fixing a system and a property, the number of events received by the monitors of RV-BIP and RVMT-BIP are similar, because both techniques produce monitored systems that are observationally equivalent to the initial ones [FJN+15,NFB+16]. Moreover, increasing the number of threads does not change the global behavior of the system, therefore the number of events is not affected by the number of threads.
2. Fixing a system and a property, the number of extra interactions imposed by RVMT-BIP is greater than the one imposed by RV-BIP. In the monitored system obtained with RVMT-BIP, after the execution of an interaction, the components that are involved in the interaction and influencing the truth-value of the property independently send their updated state to component RGT (whenever their internal computation is finished). In the monitored system obtained with RV-BIP, after the execution of an interaction influencing the truth value of the property, all the updated states will be sent at once (synchronously) to the component monitor. Hence, the evaluation of an event in RV-BIP is done in one step and the number of extra interactions imposed by RV-BIP is the same as the number of monitored events (see Table 3).
3. In spite of the higher number of extra interactions imposed by RVMT-BIP, during a multi-threaded execution, the fewer synchronous interactions of monitored components imposed by RV-BIP induces a significant overhead. This phenomenon is especially visible for the two most concurrent systems: Demosaicing and Task.
4. *On the independence of components*: Consider systems Demosaicing and Task, which consist of independent components with low-level synchronization and high degree of parallelism, and for which the monitored property requires the states of these independent components. On the one hand, at runtime, RV-BIP imposes synchronization among the components whose execution influences the truth value of the property and the component monitor. It results in a loss of the performance when executing with multiple

threads. On the other hand, RVMT-BIP collects updated states of the components independently right after their state update. Consequently, with RVMT-BIP, the system performance in the multi-threaded setting is preserved (systems Demosaicing and Task) as a negligible overhead is observed. This is a usual and complex problem which depends on many factors such as platform, model, external codes, compiler, etc. This renders the computation of the number of threads leading to peak performance complex.

5. *Synchronization of independent components*: In RV-BIP, the thread synchronizations and the synchronization of components with the monitor induce a huge overhead especially when concurrent component are concerned with the desired property (system Demosaicing and property φ_1).
6. *Synchronized components*: We observe that, for system ReaderWriterV2, the overhead obtained with RVMT-BIP monitor is slightly higher than the one obtained with RV-BIP monitors. Indeed, system Reader-WriterV2 consists of 3 writers synchronized by a clock component. Moreover, property φ_4 is defined over the states of all the writers. As a matter of fact, if one of the writers needs to communicate with component RGT, then all the other writers need to wait until the communication ends. That is, when the concurrency of the monitored system is limited by internal synchronizations, the global-state reconstruction performed by RVMT-BIP is less effective than the technique used by RV-BIP from a performance point of view.
7. *Synchronized components in independent composite components*: If the initial system (i) consists of independent composite components working concurrently, (ii) the components in each composite are highly synchronized (low degree of parallelism in each composition) and (iii) the desired property is defined over the states of the components of a specific composite component, then RVMT-BIP performs similarly to RV-BIP. Indeed, in the monitored system, the independent entities (i.e., composite component) are able to run as concurrently as in the initial system and the overhead is caused by the synchronized components. However, by increasing the number of threads, RVMT-BIP monitors offer better performance (system Reader-WriterV1).

7 Related Work

Several approaches are related to the one in this paper, as they either target CBSs or address the problem of concurrently runtime verifying systems.

7.1 Runtime Verification of Single-threaded CBSs

Dormoy et al. proposed an approach to runtime check the correct reconfiguration of components at runtime [DKL10]. They propose to check configurations over a variant of RV-LTL where the usual notion of state is replaced by the notion of component configuration. RV-LTL is a 4-valued variant of LTL dedicated to runtime verification introduced in [BLS10] and used in [FFM09]. Our approach offers several advantages compared to the approach in [DKL10]. First, our approach is not bound to temporal logic since it only requires a monitor written as a finite-state machine. This state-machine can be then generated by several already existing tools (e.g., Java-MOP) since it uses a generic format to express monitors. Thus, existing monitor synthesis algorithms from various specification formalisms can be re-used, up to a syntactic adaptation layer. Second, the instrumentation of the initial system and the addition of the monitor is formally defined, contrarily to [DKL10] where the process is only overviewed. Moreover, the whole approach leverages the formal semantics of BIP allowing us to provide a formal proof of the correctness of the proposed approach. All these features confers to our approach a higher-level of confidence.

In [FJN$^+$11,FJN$^+$15], we proposed a first approach for the runtime verification of CBSs. The approach in [FJN$^+$11,FJN$^+$15] takes a CBS and a regular property as input and generates a monitor implemented as a component. Then, the monitor component is integrated within an existing CBS. At runtime, the monitor consumes the global trace (i.e., sequence of global states) of the system and yields verdicts regarding property satisfaction. The technique in [FJN$^+$11,FJN$^+$15] only efficiently handles CBSs with sequential executions: if applied to a multi-threaded CBS, the monitor would sequentialize completely the execution. Hence, the approach proposed in this paper can be used in conjunction with the approach in [FJN$^+$11,FJN$^+$15] when dealing with multi-threaded CBSs: (only) the monitor-synthesis algorithm in [FJN$^+$11,FJN$^+$15] can be used to obtained a monitor that can be plugged to the RGT component (defined in this paper) reconstructing the global states of the system.

7.2 Synthesizing Correct Concurrent Runtime Monitors

In [FS15], the authors investigate the synthesis of correct monitors in a concurrent setting, whereby (i) the system being verified executes concurrently with the synthesized monitor (ii) the system and the monitor themselves consist of concurrent sub-components. Authors have constructed a formally-specified tool that automatically synthesizes monitors from sHML (adaptation of SafeHML (SHML) a sub-logic of the Hennessy-Milner Logic) formulas so as to asynchronously detect property violation by Erlang programs at runtime. SHML syntactically limits specifications to safety properties which can be monitored at runtime. Our approach is not bounded to any particular logic. Moreover, properties in our approach are not restricted to safety properties but can encompass co-safety, and properties that are neither safety nor co-safety properties. Moreover, the monitored properties can express the desired behavior not only on the internal states of components but they also on the states of external interactions.

7.3 Decentralized Runtime Verification

The approaches in [BF12,FCF14,BF16] decentralize monitors for linear-time specifications on a system made of synchronous black-box components that cannot be executed concurrently. Moreover, monitors only observe the outside visible behavior of components to evaluate the formulas at hand. The decentralized monitor evaluates the global trace by considering the locally-observed traces obtained by local monitors. To locally detect global violations and satisfactions, local monitors need to communicate, because their traces are only partial w.r.t. the global behavior of the system. In [BF12,FCF14,BF16], multiple components in a system each observe a subset of some global event trace. Given an LTL property φ, the objective is to create sound formula derived from φ that can be monitored on each local trace, while minimizing inter-component communication. However, they assume that 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.

Inspired by the decentralized monitoring approach to LTL properties in [BF12], Kouchnarenko and Weber [KW15] defines a progressive FTPL semantics allowing a decentralized evaluation of FTPL formula over component-based systems. Complementarily, Kouchnarenko and Weber [KW14] propose the use of temporal logics to integrate temporal requirements to adaptation policies in the context of Fractal components [BCL$^+$04]. The policies are used for specifying reflection or enforcement mechanisms, which refers respectively to corrective reconfiguration triggered by unwanted behaviors, and avoidance of reconfiguration leading to unwanted states. However, the approaches in [KW14,KW15] fundamentally differs from ours because (i) they target architectural invariants and (ii) our approach is specific to CBSs that can be executed in a multi-threaded fashion. The components in [KW14,KW15] are seen as black boxes and the interaction model considers only unidirectional connections. On the contrary, our approach leverages the internal behavior of components and their interactions for the instrumentation and global-state reconstruction.

7.4 Monitoring Safety Properties in Concurrent Systems

The approach in [SVAR06] addresses the monitoring of asynchronous multi-threaded systems against temporal logic formulas expressed in MTTL. MTTL augments LTL with modalities related to the distributed/multi-threaded nature of the system. The monitoring procedure in [SVAR06] takes as input a safety formula and a partially-ordered execution of a parallel asynchronous system, and then predicts a potential property violation on one of the causally-consistent interleavings of the observed execution. Our approach mainly differs from [SVAR06] in that we target CBSs. Moreover, we assume a central scheduler and we only need to monitor the unique causally-consistent global trace with the observed partial trace. Also, we do not place any expressiveness restriction on the formalism used to express properties.

7.5 Parallel Runtime Verification of Sequential Programs

Berkovich et al. [BBF15] introduce parallel algorithms to speed up the runtime verification of sequential programs against complex LTL formulas using a graphics processing unit (GPU). Berkovich et al. consider two levels of parallelism: the monitor (i) works along with the program in parallel, and (ii) evaluates a set of properties in a parallel fashion. Monitoring threads are added to the program and directly execute on the GPU. The approach in [BBF15] is not tailored to CBSs and is a complementary technique that adds significant computing power to the system to handle the monitoring overhead. Note that, as shown by our experiments, our approach preserves the performance of the monitored system. Finally, our approach is not bound to any particular logic, and allows for Turing-complete monitors.

8 Conclusions and Future Work

We draw conclusions and outline avenues for future work.

8.1 Conclusions

This paper introduces runtime verification for component-based systems that execute concurrently on several threads. Our approach considers an input system with partial-state semantics and transforms it to integrate a global-state reconstructor, i.e., a component that produces the witness trace at runtime. The witness trace is the sequence of global states that could be observed if the system was not multi-threaded and which contain the global information gathered from the partial-states actually traversed by the system at runtime. A runtime monitor can be then plugged to the global state reconstructor to monitor the system against properties referring to the global state of the system, while preserving the performance and benefits from concurrency. We implemented the model transformation in a prototype tool RVMT-BIP. We evaluated the performance and functional correctness of RVMT-BIP against three case studies and our running examples. Our experimental results show the effectiveness of our approach and that monitoring with RVMT-BIP induces a cheap overhead at runtime.

8.2 Future Work

Several research perspectives can be considered.

A first direction is to consider monitoring for fully decentralized and completely distributed models where a central controller does not exist. For this purpose, we intend to make controllers collaborating in order to resolve conflicts in a distributed fashion. This setting should rely on the distributed semantics of CBSs as presented in [BBJ+12] and study the influence of the organization of decentralized monitors [BF16] as done for black box components with a global clock in [CF16].

Moreover, much work has been done in order to monitor properties on a distributed (monolithic) systems; such as [SG07] for online monitoring of CTL properties, [MB15] for online monitoring of LTL properties, [SG03] for offline monitoring of properties expressed in a variant of CTL, and [TG97] for online monitoring of global-state predicates. In the future, we plan to adapt these approaches to the context of CBSs.

Another possible direction is to extend the proposed framework to runtime verify [BLS11] and enforce [FJMP16] timed specifications on timed components [BBS06].

References

BBBS08. Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5048 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2008.

BBF15. Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. Runtime verification with minimal intrusion through parallelism. *Formal Methods in System Design*, 46(3):317–348, 2015.

BBJ+12. Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

BBS06. Ananda Basu, Marius Bozga, and Joseph Sifakis. 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*, pages 3–12. IEEE Computer Society, 2006.

BCL+04. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in Java. In *International Symposium on Component-based Software Engineering*, pages 7–22. Springer, 2004.

BF12. Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

BF16. Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1-2):46–93, 2016.

BLS10. Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

BLS11. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

BS07. Simon Bliudze and Joseph Sifakis. The algebra of connectors: structuring interaction in bip. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 11–20. ACM, 2007.

CF16. Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.

DKL10. Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. Using temporal logic for dynamic reconfigurations of components. In Luís Soares Barbosa and Markus Lumpe, editors, *Proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010)*, volume 6921 of *LNCS*, pages 200–217. Springer, 2010.

FCF14. Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In Erika Ábrahám and Catuscia Palamidessi, editors, *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*, volume 8461 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2014.

FFM09. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron Peled, editors, *Proceedings of the 9th International Workshop on Runtime Verification (RV 2009), Selected Papers*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.

FFM12. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

FJMP16. Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Systems & Control Letters*, 123:2–41, 2016.

FJN+11. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems. In *SEFM 2011*, pages 204–220, 2011.

FJN+15. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. 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.

FS15. Adrian Francalanza and Aldrin Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.

Hoa78. Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

KW14. Olga Kouchnarenko and Jean-François Weber. *Adapting Component-Based Systems at Runtime via Policies with Temporal Patterns*, volume 8348 of *Lecture Notes in Computer Science*, pages 234–253. Springer International Publishing, Cham, 2014.

KW15. Olga Kouchnarenko and Jean-François Weber. *Decentralised Evaluation of Temporal Patterns over Component-Based Systems at Runtime*, volume 8997 of *Lecture Notes in Computer Science*, pages 108–126. Springer International Publishing, Cham, 2015.

MB15. Menna Mostafa and Borzoo Bonakdarpour. 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*, pages 494–503. IEEE Computer Society, 2015.

Mil95. R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

Naz. Hosein Nazarpour. Website of RVMT-BIP, a tool for the Runtime Verification of Multi-Threaded BIP systems. `http://www-verimag.imag.fr/~nazarpou/rvmt.html`.

NFB+16. Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Monitoring multi-threaded component-based systems. In Erika Abraham and Marieke Huisman, editors, *Proceedings of the 12th International Conference on integrated Formal Methods*, LNCS, 2016.

SG03. Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In Marina Papatriantafilou and Philippe Hunel, editors, *Principles of Distributed Systems, 7th International Conference, OPODIS 2003 La Martinique, French West Indies, December 10-13, 2003 Revised Selected Papers*, volume 3144 of *Lecture Notes in Computer Science*, pages 171–183. Springer, 2003.

SG07. Alper Sen and Vijay K. Garg. Formal verification of simulation traces using computation slicing. *IEEE Trans. Computers*, 56(4):511–527, 2007.

SVAR06. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Decentralized runtime analysis of multithreaded applications. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

TG97. Alexander I Tomlinson and Vijay K Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997.

vGV97. Rob van Glabbeek and Frits Vaandrager. The difference between splitting innandn+ 1. *Information and Computation*, 136(2):109–142, 1997.

A Correctness Proof of the Approach

Before tackling the proof of correctness of our approach, we provide an intuitive description of the proof content. The correctness of our approach relies on three results.

The first result concerns the witness trace. Given a CBS B whose semantics is described as per Section 3, that is the general semantics of CBS. One can build $B^{\perp}$, a transformed version of B that can execute concurrently and which is bi-similar to B. $B^{\perp}$ executes following the partial-state semantics described in Section 4.1. Any trace of an execution of $B^{\perp}$ can be related to the trace of a unique execution of B, i.e., its witness. Property 1 states that any witness trace corresponds to the execution in global-state semantics that has the same sequence of interaction executions, i.e., that the witness relation captures the abovementioned relation between a system in global-state semantics and the corresponding system in partial-state semantics. Property 2 states that from any execution in partial-state semantics, the witness exists and is unique.

The second result states that function RGT builds the witness trace from a trace in partial-state semantics in an online fashion. Theorem 1 states the correctness of this function.

The third result states that the transformed components, the synthesized components, and their connection are correct. That is, the obtained system (i) computes the witness and implements function RGT (Proposition 2), and (ii) is bisimilar to the initial system (Theorem 2).

Proof outline. The following proofs are organized as follows. The proof of Property 1 is in Appendix A.1. The proof of Property 2 is in Appendix A.2. Some intermediate lemmas with their proofs are introduced in Appendix A.3 in order to prove Theorem 1 in Appendix A.4. The proofs of Propositions 1 and 2 are respectively given in Appendices A.4 and A.5. Some intermediate definitions and lemmas with their proofs are given in Appendix A.6 in order to prove Theorem 2 in Appendix A.7.

A.1 Proof of Property 1 (p. 8)

We shall prove that:

$$\forall(\sigma_1,\sigma_2) \in \mathrm{W} \,.\, \mathrm{interactions}(\sigma_1) = \mathrm{interactions}(\sigma_2),$$

where W is the witness relation defined in Definition 7 (using a bi-simulation relation R), and $\mathrm{interactions}(\sigma)$ is the sequence of interactions of trace σ.

Proof. The proof is done by structural induction on W.

- Base case. By definition of W, $(Init, Init) \in W$ and $\mathrm{interactions}(Init) = \epsilon$.
- Induction case. Let us consider $(\sigma_1,\sigma_2) \in W$ and suppose that $\mathrm{interactions}(\sigma_1) = \mathrm{interactions}(\sigma_2)$. According to the definition of W, there are two rules for constructing a new element in W.
  - Consider $(\sigma_1 \cdot a \cdot q_1, \sigma_2 \cdot a \cdot q_2) \in \mathrm{W}$ such that $a \in \gamma$ and $(q_1,q_2) \in R$. We have $\mathrm{interactions}(\sigma_1 \cdot a \cdot q_1) = \mathrm{interactions}(\sigma_1) \cdot a$ and $\mathrm{interactions}(\sigma_2 \cdot a \cdot q_2) = \mathrm{interactions}(\sigma_2) \cdot a$, and thus the expected result using the induction hypothesis.
  - Consider $(\sigma_1, \sigma_2 \cdot \beta \cdot q_2) \in \mathrm{W}$ such that $(\mathrm{last}(\sigma_1), q_2) \in R$. We have $\mathrm{interactions}(\sigma_2 \cdot \beta \cdot q_2) = \mathrm{interactions}(\sigma_2)$ and thus the expected result using the induction hypothesis.

A.2 Proof of Property 2 (p. 8)

We shall prove that:

$$\forall \sigma_2 \in \mathrm{Tr}(B^{\perp}), \exists! \, \sigma_1 \in \mathrm{Tr}(B) \,.(\sigma_1,\sigma_2) \in \mathrm{W},$$

where B is a component-based system (with set of traces $\mathrm{Tr}(B)$) and $B^{\perp}$ is the corresponding component-based system with partial-state semantics (with set of traces $\mathrm{Tr}(B^{\perp})$).

Proof. First, let us note that from the weak bi-simulation of a global-state semantics model with its corresponding partial-state semantics model [BBS06], we can conclude that, for any trace in the partial-state semantics model, there exists a corresponding trace in the global-state semantics model. We prove that the witness trace is unique by contradiction.

Let us assume that for a trace in partial-state semantics $\sigma_2 \in \mathrm{Tr}(B^{\perp})$, there exist two witness traces $\sigma_1', \sigma_1 \in \mathrm{Tr}(B)$ such that $(\sigma_1,\sigma_2), (\sigma_1',\sigma_2) \in W$ and $\sigma_1 \neq \sigma_1'$. From Property 1, $\mathrm{interactions}(\sigma_1) = \mathrm{interactions}(\sigma_2)$ and $\mathrm{interactions}(\sigma_1') = \mathrm{interactions}(\sigma_2)$, therefore $\mathrm{interactions}(\sigma_1) = \mathrm{interactions}(\sigma_1')$. Moreover, σ_1 and σ_1' have the same initial state because of the definition of W and (σ_1,σ_2), $(\sigma_1',\sigma_2) \in W$. From the semantics of composite components, a sequence of interactions is associated to a unique trace (from a unique initial state). This is thus a contradiction.

A.3 Intermediate Lemmas

We prove the intermediate lemmas that are needed to prove Theorem 1.

Proof of Lemma 1 (p. 10). We shall prove $\forall(\sigma_1, \sigma_2) \in W \,.|\mathrm{acc}(\sigma_2)|= |\sigma_1|= 2 \times s + 1$, where $s = |\mathrm{interactions}(\sigma_1)|$, where acc is the accumulator used in the definition of function RGT (Definition 8), and function $\mathrm{interactions}$ (defined in Section 4.1) returns the sequence of interactions in a trace (removing β).

Proof. The proof is done by structural induction on W.

- Base case. By definition of W, $(Init, Init) \in W$ and we have $\mathrm{acc}(Init) = Init$, $|Init|= 1$ and $|\mathrm{interactions}(Init)|= |\epsilon|= 0$.
- Induction case. Let us consider $(\sigma_1, \sigma_2) \in W$ such that $\mathrm{interactions}(\sigma_2) = s$ and suppose that Lemma 1 holds for (σ_1, σ_2). According to the definition of W, there are two rules for constructing a new element in W.
  - Consider $(\sigma_1 \cdot a \cdot q_1, \sigma_2 \cdot a \cdot q_2) \in \mathrm{W}$ such that $a \in \gamma$ and $(q_1, q_2) \in R$. According to Definition 8, $\mathrm{acc}(\sigma_2 \cdot a \cdot q_2) = \mathrm{acc}(\sigma_2) \cdot a \cdot q_2$. Using the induction hypothesis, $|\mathrm{acc}(\sigma_2)|= |\sigma_1|$. Hence $|\mathrm{acc}(\sigma_2 \cdot a \cdot q_2)|= |\sigma_2|+2 = |\sigma_1|+2 = |\sigma_1 \cdot a \cdot q_1|$.
  - Consider $(\sigma_1, \sigma_2 \cdot \beta \cdot q_2) \in \mathrm{W}$ such that $(\mathrm{last}(\sigma_1), q_2) \in R$. According to Definition 8 and using the definition of operator map, we have $|\mathrm{acc}(\sigma_2 \cdot \beta \cdot q_2)|= |\mathrm{map}\ [x \mapsto \mathrm{upd}(q, x)]\ (\mathrm{acc}(\sigma_2))|= |\mathrm{acc}(\sigma_2)|$, and thus we obtain the expected result using the induction hypothesis.

Proof of Lemma 2 (p. 10). We shall prove that: $\forall \sigma \in \mathrm{Tr}(B^{\perp}), \exists k \in [1\,..\,s] \,.\, q_k \in Q \implies \forall z \in [1\,..\,k] \,.\, q_z \in Q\ , q_{z-1} \xrightarrow{a_z} q_z$ where $(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s) = \mathrm{acc}(\sigma)$.

Proof. According to Lemma 1 and the definition of function acc (see Definition 8), a state is generated and added to sequence $\mathrm{acc}(\sigma)$ just after the execution of an interaction $a \in \gamma$. This state is obtained from the last state in $\mathrm{acc}(\sigma)$, say q, such that the new state has state information about less components than q because the states of all components involved in a are undetermined and the states of all other components are identical. Since after any busy transition, function upd (see Definition 8) updates all the generated partial states that do not have the state information regarding the components that performed a busy transition, the completion of each partial state guarantees the completion of previously generated states. Therefore, if there exists a global state (possibly completed through function upd) in trace $\mathrm{acc}(\sigma)$, then all the previously generated states are global states.

Moreover, the sequence of reconstructed global states follow the global-state semantics. This results stems from two facts. First, according to the definition of function upd, whenever function upd completes a partial state in the trace by adding the state of a component for which the last state in the trace is undetermined, it uses the next state reached by this component according to partial-state semantics. Second, according to Definition 5, the transformation of a component to make it compatible with partial-state semantics is such that an intermediate busy state, say $\perp$, is added between the starting state q and arriving state q' of any transition (q, p, q'). Moreover, the transitions $(q, p, \perp)$ and $(\perp, \beta, q')$ in the partial-state semantics replace the previous transition (q, p, q') in the global-state semantics. Hence, whenever a component in partial-state semantics is in a busy state $\perp$, the next state that it reaches is necessarily the same state as the one it would have reached in the global-state semantics.

Proof of Proposition 1 (p. 10). We shall prove that $\forall \sigma \in \mathrm{Tr}(B^{\perp}) \,.$

$$\begin{aligned} &|\mathrm{discriminant}(\mathrm{acc}(\sigma))|\leq |\mathrm{acc}(\sigma)| \\ \wedge\, &\mathrm{discriminant}(\mathrm{acc}(\sigma)) = q_0 \cdot a_1 \cdot q_1 \cdots a_d \cdot q_d \implies \forall i \in [1\,..\,d] \,.\, q_{i-1} \xrightarrow{a_i} q_i, \end{aligned}$$

where acc is the accumulator function and $\mathrm{discriminant}$ is the discriminant function used in the definition of function RGT (Definition 8) such that $\mathrm{RGT}(\sigma) = \mathrm{discriminant}(\mathrm{acc}(\sigma))$.

Proof. The proof directly follows from the definitions of functions acc and $\mathrm{discriminant}$, and Lemma 2. Let us consider $\sigma \in \mathrm{Tr}(B^{\perp})$.

Regarding the first conjunct, according to the definition of function $\mathrm{discriminant}$, $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ is the longest prefix of $\mathrm{acc}(\sigma)$ such that the last state of $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ is a global state. Thus, the length of sequence $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ is always lesser than or equal to the length of sequence $\mathrm{acc}(\sigma)$.

Regarding the second conjunct, according to Lemma 2, all the states of $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ are global states and follow the global-state semantics. Moreover, one can note that function $\mathrm{discriminant}$ removes the longest suffix made of partial states output by function acc and function acc only updates partial states.

Proof of Lemma 3 (p. 11). We shall prove that: $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\,\mathrm{last}(\mathrm{acc}(\sigma)) = \mathrm{last}(\sigma)$.

Proof. The proof is done by induction on the length of the trace in partial-state semantics, i.e., $\sigma \in \mathrm{Tr}(B^{\perp})$.

- Base case: The property holds for the initial state. Indeed, in this case $\sigma = \mathit{Init}$ and according to the definition of function acc (see Definition 8) $\mathrm{last}(\mathrm{acc}(\mathit{Init})) = \mathit{Init}$.
- Induction case: Let us assume that $\sigma = q_0 \cdot a_1 \cdot q_1 \cdots a_m \cdot q_m$ is a trace in partial-state semantics and $\mathrm{acc}(\sigma) = q'_0 \cdot a'_1 \cdot q'_1 \cdots a'_s \cdot q'_s$ such that $q_m = q'_s$. We have two cases according to whether the next move of the partial-state semantics model is an interaction or a busy transition:
  - If $a_{m+1} \in \gamma$, then according to the definition of the function acc, we have: $\mathrm{last}(\mathrm{acc}(\sigma \cdot a_{m+1} \cdot q_{m+1})) = q_{m+1}$.
  - If $a_{m+1} \in \{\beta_i\}_{i=1}^{n}$, then according to the definition of function acc, we have: $\mathrm{last}(\mathrm{acc}(\sigma \cdot a_{m+1} \cdot q_{m+1})) = \mathrm{upd}(q_{m+1}, q'_s)$. From the induction hypothesis: $\mathrm{upd}(q_{m+1}, q'_s) = \mathrm{upd}(q_{m+1}, q_m)$ and from the fact that the only difference between state q_m and state q_{m+1} is that in state q_m the state of the component that executed a_{m+1} is a busy state, while in state q_{m+1} it is not a busy state. From the definition of function upd (Definition 8), we can conclude that $\mathrm{upd}(q_{m+1}, q_m) = q_{m+1}$.

  In both cases, $\mathrm{last}(\mathrm{acc}(\sigma)) = \mathrm{last}(\sigma)$.

Proof of Lemma 4 (p. 11). We shall prove that $\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\,\mathrm{interactions}(\mathrm{acc}(\sigma)) = \mathrm{interactions}(\sigma)$.

Proof. By an easy induction on the length of σ and case analysis on the definition of function acc (Definition 8).

A.4 Proof of Theorem 1 (p. 11)

We shall prove that, for a given CBS $B = \gamma(B_1, \ldots, B_n)$ with set of traces $\mathrm{Tr}(B)$ and $B^{\perp}$, the following holds on the set of traces $\mathrm{Tr}(B^{\perp})$ of the corresponding CBS with partial-state semantics:

$$
\begin{aligned}
&\forall \sigma \in \mathrm{Tr}(B^{\perp})\,.\\
&\qquad \mathrm{last}(\sigma) \in Q \implies \mathrm{RGT}(\sigma) = W(\sigma)\\
&\qquad \wedge\, \mathrm{last}(\sigma) \notin Q \implies \mathrm{RGT}(\sigma) = W(\sigma') \cdot a, \text{with}\\
&\sigma' = \min_{\preceq}\{\sigma_p \in \mathrm{Tr}(B^{\perp}) \mid \exists a \in \gamma, \exists \sigma'' \in \mathrm{Tr}(B^{\perp})\,.\,\sigma = \sigma_p \cdot a \cdot \sigma''\\
&\qquad\qquad \wedge \exists i \in [1\,..\,n]\,.(B_i.P \cap a \neq \emptyset) \wedge (\forall j \in [1\,..\,\mathrm{length}(\sigma'')]\,.\,\beta_i \neq \sigma''(j))\}
\end{aligned}
$$

where function RGT is defined in Definition 8 and W is the witness relation defined in Definition 7.

Proof. For any trace in partial-state semantics σ, we consider two cases depending on whether the last element of σ belongs to Q of not:

- If $\mathrm{last}(\sigma) \in Q$, according to Lemma 3, $\mathrm{last}(\mathrm{acc}(\sigma)) \in Q$ and thus $\mathrm{RGT}(\sigma) = \mathrm{discriminant}(\mathrm{acc}(\sigma)) = \mathrm{acc}(\sigma)$. Let us assume that $\mathrm{acc}(\sigma) = q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s$, with $q_0 = \mathit{Init}$. According to Lemma 2, $\forall k \in [1\,..\,s]\,.\,q_{k-1} \xrightarrow{a_k} q_k \implies \mathrm{acc}(\sigma) \in \mathrm{Tr}(B)$. Moreover, according to Lemma 4, $\mathrm{interactions}(\mathrm{acc}(\sigma)) = \mathrm{interactions}(\sigma)$. Furthermore, according to definition of the witness relation (Definition 7), from the unique initial state, since $\mathrm{acc}(\sigma)$ and σ have the same sequence of interactions, $(\mathrm{acc}(\sigma), \sigma) \in W$. Therefore, $\mathrm{acc}(\sigma) = \mathrm{RGT}(\sigma) = W(\sigma)$.
- If $\mathrm{last}(\sigma) \notin Q$, we treat this case by induction on the length of σ. Let us assume that the proposition holds for some $\sigma \in \mathrm{Tr}(B^{\perp})$ (induction hypothesis). Let us consider $\sigma = \sigma' \cdot a'_1 \cdot q'_1 \cdot a'_2 \cdot q'_2 \cdots a'_k \cdot q'_k$, with $k > 0$. Let us assume that the splitting of σ is $\sigma' \cdot a'_1 \cdot \sigma''$, where σ' is the minimal sequence such that there exists at least one component that is involved in interaction $a'_1 \in \gamma$ and that is still busy. (We note that in this case σ' do exist because $\mathrm{last}(\sigma) \notin Q$ implies that the system has made at least one move.) Let i be the identifier of this component and a'_1 be s^{th} interaction in trace σ such that $a'_1 = \mathrm{interactions}(\sigma)(s)$. Let us consider $\sigma \cdot a'_{k+1} \cdot q'_{k+1}$, the trace extending σ by one interaction a'_{k+1}. We distinguish again two subcases depending on whether $a'_{k+1} \in \gamma$ or not.
  - Case $a'_{k+1} \in \gamma$. We have $\mathrm{last}(\sigma) \notin Q$ and then $\mathrm{last}(\sigma \cdot a'_{k+1} \cdot q'_{k+1}) \notin Q$ (because $a'_{k+1} \in \gamma$, i.e., the system performs an interaction, and the state following an interaction is necessarily a partial state). Moreover, $\mathrm{RGT}(\sigma) = \mathrm{RGT}(\sigma \cdot a'_{k+1} \cdot q'_{k+1})$, i.e., the reconstructed global state does not change. Hence, the components which are busy after a'_1 are still busy. Consequently, the splitting of σ and $\sigma \cdot a'_{k+1} \cdot q'_{k+1}$ are the same. Following the induction hypothesis, $\sigma \cdot a'_{k+1} \cdot q'_{k+1}$ has the expected property.

- Case $a'_{k+1} = \beta_j$, for some $j \in [1\,..\,n]$. We distinguish again two subcases.
  * If $i = j$, that is the busy interaction β_j concerns the component(s) for which information was missing in σ'' (component i). If component i is the only component involved in interaction a'_1 for which information is missing in $q'_1 \cdots q'_k$, the reconstruction of the global state corresponding to the execution of a'_1 can be done just after receiving the state information of component i. After receiving q'_{k+1}, which contains the state information of component i, the partial states of $\mathrm{acc}(\sigma)$ are updated with function upd. That is, $\mathrm{RGT}(\sigma \cdot a'_{k+1} \cdot q'_{k+1}) = \mathrm{RGT}(\sigma) \cdot q''_0 \cdot a''_1 \cdot q''_1 \cdots q''_{m-1} \cdot a''_m$, where $m > 0$, q''_0 is the reconstructed global state associated with interaction a'_1, and $a''_m = \mathrm{interactions}(\sigma)(s+m)$ is the first interaction executed after σ for which there exists at least one involved component which is still busy. Indeed, some interactions after a'_1 in trace σ (i.e., $a''_p = \mathrm{interactions}(\sigma)(s+p)$ for $m > p > 0$) may exist and be such that component i is the only component involved in them for which information is missing to reconstruct the associated global states. In this case, updating the partial states of $\mathrm{acc}(\sigma)$ with the state information of component i yields several global states i.e., $q''_1 \cdots q''_{m-1}$. Then, the splitting of σ changes as follows: $\sigma = \sigma'' \cdot a''_m \cdots a'_{k+1} \cdot q'_{k+1}$, where $\sigma'' = \sigma' \cdot a'_1 \cdot q'_1 \cdot a'_2 \cdot q'_2 \cdots q'_t$ and q'_t is the system state before interaction a''_m. Therefore, $\mathrm{RGT}(\sigma \cdot a'_{k+1} \cdot q'_{k+1}) = W(\sigma'') \cdot a''_m$ and the property holds again.
  * If $i \neq j$, we have $\mathrm{RGT}(\sigma) = \mathrm{RGT}(\sigma \cdot a'_{k+1} \cdot q'_{k+1})$. Hence, the splitting of σ and $\sigma \cdot a'_{k+1} \cdot q'_{k+1}$ are the same. Following the induction hypothesis, $\sigma \cdot a'_{k+1} \cdot q'_{k+1}$ has the expected property.

A.5 Proof of Proposition 2 (p. 17)

Given a CBS $B = \gamma(B_1, \ldots, B_n)$ with corresponding partial-state semantics model $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ and the transformed composite component $B^r = \gamma^r(B_1^r, \ldots, B_n^r, RGT)$ obtained as per Definition 11, we shall prove that for any execution of the system with partial-state semantics with trace $\sigma \in \mathrm{Tr}(B^\perp)$, component RGT (Definition 10) implements function RGT (Definition 8), that is $\forall \sigma \in \mathrm{Tr}(B^\perp)\,.\,\mathrm{RGT}\,.V \cong \mathrm{acc}(\sigma)$.

Proof. The proof is done by induction on the length of $\sigma \in \mathrm{Tr}(B^\perp)$, i.e., the trace of the system in partial-state semantics.

- Base case. By definition of function RGT, at the initial state $\mathrm{acc}(Init) = Init$. By definition of component RGT, V is initialized as a tuple representing the initial state of the system. Therefore, $\mathrm{RGT}\,.V \cong \mathrm{acc}(Init)$.
- Induction case. Let us suppose that the proposition holds for a trace $\sigma \in \mathrm{Tr}(B^\perp)$, that is $\mathrm{RGT}\,.V \cong \mathrm{acc}(\sigma)$. According to the definition of function RGT, $\mathrm{RGT}(\sigma) = \mathrm{discriminant}(\mathrm{acc}(\sigma))$. Consequently, there exists $\sigma' \in \mathrm{Tr}(B^\perp)$ of the form $\sigma' = q'_0 \cdot a'_1 \cdot q'_1 \cdots q'_k$, with $k \geqslant 0$, such that $\mathrm{acc}(\sigma) = \mathrm{RGT}(\sigma) \cdot \sigma'$. We distinguish two cases depending on the action of the system executed after σ:
  - The first case occurs when the action is the execution of an interaction a'_{k+1}, followed by a partial state q'_{k+1}. On the one hand, we have $\mathrm{acc}(\sigma \cdot a'_{k+1} \cdot q'_{k+1}) = \mathrm{acc}(\sigma) \cdot a'_{k+1} \cdot q'_{k+1}$. On the other hand, in component RGT, according to Algorithm 1 (line 6), the corresponding transition $\tau \in T_{\mathrm{new}}$ extends the sequence of tuples V by a new $(n+1)$-tuple v which consists of the current partial state of the system such that $V = V \cdot v$ and $v \cong q'_{k+1}$. Therefore, we have $\mathrm{RGT}\,.V \cong \mathrm{acc}(\sigma)$ as expected.
  - The second case occurs when the next action is the execution of a busy transition. On the one hand, function RGT updates all the partial states $q'_0, \ldots, q'_k$. On the other hand, according to Algorithm 2 (lines 2-6), in component RGT, the corresponding transition $\tau \in T_{\mathrm{upd}}$ updates the sequence of tuples V such that $\mathrm{RGT}\,.V \cong \mathrm{acc}(\sigma)$ hold.
    Moreover, function RGT and component RGT similarly create new global states from the partial states whenever new global states are computed. On the one hand, after any update of partial states, through function $\mathrm{discriminant}$, function RGT outputs the longest prefix of the generated trace which corresponds to the witness trace. On the other hand, after any update of the sequence of tuples V, component RGT checks for the existence of fully completed tuples in V to deliver them to through the dedicated ports to the runtime monitor.

A.6 Proofs of Intermediate Lemmas

In the following proofs, we will consider several mathematical objects in order to prove the correctness of our framework:

- a composite component with partial-state semantics $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ of behavior $(Q^\perp, \gamma^\perp, \longrightarrow)$;
- the transformed composite component $B^r = \gamma^r(B_1^r, \ldots, B_n^r, RGT)$ of behavior $(Q^r, \gamma^r, \longrightarrow_r)$. B^r is obtained from $B^\perp$ by following the transformations described in Section 5.

Proof of Lemma 5 (p. 17). We shall prove that in any state of the transformed system, if there is a non-empty set $GS \subseteq \{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ in which all variables are `true`, the variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\} \setminus GS$ cannot be set to `true` until all variables in GS are reset to `false` first.

Proof. According to the definition of component RGT (Definition 10), on the one hand only the transitions in set T_{upd} are able to set the value of the variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ to `true`; on the other hand the transitions in set T_{upd} are guarded by $\bigwedge_{a\in\gamma}(\neg gs_a)$ which means that all of the Boolean variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ must be `false` for one of these transitions to execute. Therefore, in any state $q \in Q^r$ such that such a set GS exists, the transitions in T_{upd} are not possible. Moreover, the only possible transitions in state q are the transitions in set T_{out} which effect is to reset the value of the variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ to `false` using algorithm `get`.

Proof of Lemma 6 (p. 17). We shall prove that for any state $q \in Q^r$, there exists a state $q' \in Q^r$ reached after interactions in a^m (i.e., $q \xrightarrow{(a^m)^*} q'$), such that q' is a stable state (i.e., $\mathrm{stable}(q')$).

Proof. Let us consider a non-stable state $q \in \mathrm{RGT}.Q$. The interactions in a^m involve to execute ports in $\{p'_a \mid a \in \gamma\}$ and transitions in T_{out}. Since q is a non-stable state, at least one of the variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ evaluates to `true` in q (see Definition 13, p. 17). Such transitions entail to execute algorithm `get` (Algorithm 4) which resets the Boolean variable to `false` by delivering the associated reconstructed global state(s) to the monitor. After executing algorithm `get`, if there exists another Boolean variable in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ that evaluates to `true`, according to Lemma 5, component RGT returns to a situation where only again algorithm `get` can execute (through the interactions in set a^m). The above process executes until the system eventually reaches a state q' where no interaction in a^m is enabled. Therefore, in q' all Boolean variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ evaluate to `false`, because interactions in a^m are unary interactions, each involving port $\mathrm{RGT}.p'_a$ (Definition 11) guarded by $\bigwedge_{a\in\gamma}(\neg gs_a)$ (Definition 10). According to Definition 13, a state is stable when all Boolean variables in $\{\mathrm{RGT}.gs_a \mid a \in \gamma\}$ evaluate to `false`. Thus, q' is stable.

Proof of Lemma 7 (p. 17). Let us consider two states: q of the initial model and q^r its corresponding state in the transformed model such that $\mathrm{equ}(q^r) = q$. There exists an enabled interaction in the initial model ($a \in \gamma^\perp$) in state $q \in Q^\perp$, if and only if the corresponding interaction in the transformed model ($a^r \in \gamma^r$) is enabled at state q^r.

Proof. According to the definitions of interaction transformation and atom RGT (Definition 10), ports $\mathrm{RGT}.p_a$, for $a \in \gamma$, are always enabled. Since for a given interaction a, a^r and a differ only by port $\mathrm{RGT}.p_a$, we can conclude that $a^r \in a^r_\gamma$ is enabled if and only if $a \in \gamma^\perp$ is enabled.

A.7 Proof of Theorem 2 (p. 18)

Before tackling the proof of Theorem 2, we convey a remark preparing the definition of the weak bi-simulation relation defined in the proof.

Following Definition 11, the set of interactions γ^r of the instrumented system is partitioned as $\gamma^r = a^r_\gamma \cup a^r_\beta \cup a^m$, where a^r_γ is the set of interactions of the initial system augmented by RGT port, a^r_β is a set containing the busy interactions of the initial system (one for each component) augmented by RGT port, and a^m is a new set of interactions used for monitoring purposes. First, we note that the set of interactions in the instrumented system a^r_γ and a^r_β are isomorphic to the sets of interactions γ and $\{\{\beta_i\}\}_{i=1}^n$ of the initial system because they contain only an additional port to notify component RGT. We can thus identify these sets of interactions. Moreover, as usual in monitoring, the actions used for monitoring purposes (i.e., interactions in a^m) are considered to be unobservable. These interactions do not influence the state of the system and execute independently of the interactions in $a^r_\gamma \cup a^r_\beta$; these are interactions occurring between RGT and the monitor which are components introduced in the instrumentation. See also [FJN+15], for more arguments along these lines related to the instrumentation of single-threaded CBSs.

Proof. We exhibit a relation $R \subseteq Q^\perp \times Q^r$ between the set of states of the initial model with partial-state semantics and the set of states of the transformed model. We define $R = \{(q, q^r) \mid \exists z^r \in Q^r \centerdot q^r \xrightarrow{(a^m)^*}_r z^r \wedge \mathrm{equ}(z^r) = q\}$, and we shall prove that relation R satisfies the following properties to establish that R is a weak bi-simulation:

(i) $\Big((q, q^r) \in R \wedge \exists z^r \in Q^r \centerdot q^r \xrightarrow{a^m}_r z^r\Big) \implies (q, z^r) \in R$;

(ii) $\Big((q,q^r) \in R \wedge \exists z^r \in Q^r \centerdot q^r \xrightarrow{a^r_\gamma + a^r_\beta}_r z^r\Big) \implies \exists z \in Q \centerdot \Big(q \xrightarrow{a} z \wedge (z,z^r) \in R\Big)$;

(iii) $\Big((q,q^r) \in R \wedge q \xrightarrow{\gamma^\perp} z\Big) \implies \exists z^r \in Q^r \centerdot \Big(q^r \xrightarrow{(a^m)^*.a^r}_r z^r \wedge (z,z^r) \in R\Big)$.

Let us consider $q = (q_1, \cdots, q_n)$ and $q^r = (q^r_1, \cdots, q^r_n, q^r_{n+1})$ such that $(q, q^r) \in R$.

Proof of (i):

Since $(q, q^r) \in R$, there exists a stable state $q^{r\prime} \in Q^r$ which is reached after unobservable interactions in a^m. After the execution of some unary interaction $\alpha \in a^m$, the corresponding Boolean variable $\mathrm{RGT}.gs_\alpha$ is set to `false` (Algorithm 4). Let us consider that the next state after the execution of some interaction $\alpha \in a^m$ is $z^r = (z^r_1, \cdots, z^r_n, z^r_{n+1})$. If z^r_{n+1} is a stable state then $\mathrm{equ}(z^r) = q$ thus $(q, z^r) \in R$, and if z^r_{n+1} is not a stable state according to Lemma 6, after interaction $\alpha \in a^m$, the state of RGT (that is z^r_{n+1}) will be stable, therefore we conclude that $(q, z^r) \in R$.

Proof of (ii):

Let us consider $z^r = (z^r_1, \cdots, z^r_n, z^r_{n+1})$ and $z = (z_1, \cdots, z_n)$. When some $a^r \in (a^r_\gamma \cup a^r_\beta)$ is enabled, from the definition of the semantics of transformed composite component and Lemma 7, we can deduce that the corresponding interaction $a \in \gamma^\perp$ is enabled (recall, that for each interaction $a \in \gamma^\perp$ in the initial model with partial-state semantics there exists a corresponding interaction a^r in the transformed model, as per Definition 11). Executing the corresponding interactions a and a^r changes the local states q^r_i and q_i, for $i \in [1\,..\,n]$, to z^r_i and z_i for $i \in [1\,..\,n]$ respectively, in such a way that $z^r_i = z_i$, for $i \in [1\,..\,n]$, because the transformations do not modify the transitions of the components of the initial model. After a^r, we have two cases depending on whether z^r_{n+1} is stable or not.

- If z^r_{n+1} is stable, from the definition of relation R, we have $(z, z^r) \in R$.
- If z^r_{n+1} is not stable, then according to Lemma 6, z^r_{n+1} will be stable after some interactions $\alpha \in a^m$ (that is $z^r_{n+1} \xrightarrow{\alpha^*} \mathrm{stable}(z^r_{n+1})$). Therefore, $(z, z^r) \in R$.

Proof of (iii):

Let us consider $z^r = (z^r_1, \ldots, z^r_n, z^r_{n+1})$. When $a \in \gamma^\perp$ is enabled in the initial model, we can consider two cases depending on whether the corresponding interaction a^r in the transformed model is enabled or not.

- If a^r is enabled, we have two cases for the next state of component RGT:
  - if $a^r \in a^r_\gamma$, according to the definition of atom RGT, z^r_{n+1} is stable and $(z, z^r) \in R$.
  - if $a^r \in a^r_\beta$, we have two cases:
    - If RGT has some global states to deliver (that is z^r_{n+1} is not stable), then, according to Lemma 6, RGT will be stable after some interactions in a^m. Hence, $(z, z^r) \in R$.
    - If RGT has no global state, then atom RGT is stable and $(z, z^r) \in R$.
- If a^r is not enabled, according to the definition of atom RGT, we can conclude that RGT has some global states to deliver, thus q^r is not stable. According to Lemma 6, a not stable system becomes stable after executing some interactions in a^m. Therefore, according to Lemma 7, a^r is necessarily enabled when the system is stable. Consequently, the same reasoning followed for the previous case can be conducted in which a^r is initially enabled. Henceforth, $(z, z^r) \in R$.