Monitoring Multi-Threaded Component-Based Systems

Hosein Nazarpour, Yliès Falcone, Saddek Bensalem,

Marius Bozga, Jacques Combaz

Verimag Research Report n^o TR-2015-5

January 13, 2016

Monitoring Multi-Threaded Component-Based Systems

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

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

January 13, 2016

Abstract

This paper addresses the monitoring of user-provided properties on 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 a monitor which would force the system to synchronize in order to obtain the sequence of global states at runtime. Such 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 the semantics and the 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.

Keywords: component-based design, multiparty interaction, multi-threaded, monitoring, concurrency, runtime verification

Reviewers:

Notes:
Version 1: July 16, 2015,
Version 2: November 20, 2015.

How to cite this report:

```
@techreport {TR-2015-5,
  title = {Monitoring Multi-Threaded Component-Based Systems},
   author = {Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga, Jacques
Combaz},
  institution = {{Verimag} Research Report},
  number = {TR-2015-5},
  year = {2015}
}
```

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 behaviour. They can be composed in order to build composite components. Their composition should be rigorously defined so that it is possible to infer the behaviour of composite components from the behaviour 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) s.t. the coordinated behaviour $\gamma(B_1, \ldots, B_n)$ meets the property φ. It is however generally not possible to ensure or verify the desired property φ using static verification techniques, either because of the state-explosion problem or because φ can only be decided with runtime information. In this paper, we are interested in complementary verification techniques for CBSs such as runtime verification. In [8], we introduce 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 $\gamma\,(B_1, \ldots, B_n)$ and a regular property, a component monitor M and a new set of interactions γ' are synthesised to build a new composite system $\gamma'\,(B_1, \ldots, B_n, M)$ where the property is checked at runtime.

The underlying model of CBSs rely 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 $\gamma\,(B_1, \ldots, B_n)$ yields a bisimilar component ([1]) 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 properties referring to global states that preserves the concurrency and semantics of the monitored system. Our method 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 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. 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 overviews our approach. According to [1], a BIP system

[1]RVMT-BIP is available for download at [11].

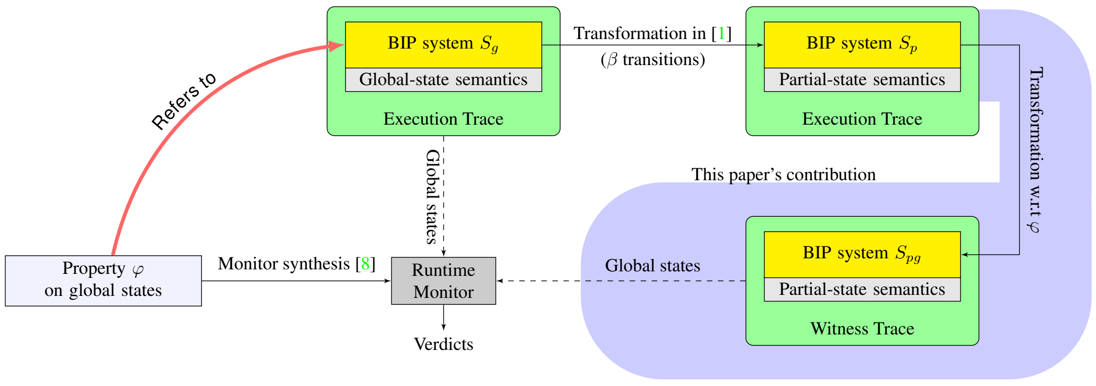


Figure 1: Approach overview

with global-state semantics S_g (sequential model), is (weakly) bisimilar [9] with the corresponding partial-state model S_p (concurrent model) noted $S_g \sim S_p$. 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. Naive solutions to monitor S_p would be i) to monitor S_g with the technique in [8] and run S_p, which ends up with delays in detecting verdicts or ii) to plug the monitor proposed in [8] in S_p, which forces the components to synchronize for the monitor to take a snapshot of the global state of the system. Such approaches would completely defeat the purpose of using multi-threaded models. Instead, we propose a transformation technique to build another system S_{pg} out of S_p such that i) S_{pg} and S_p are bisimillar (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 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 that it always produces the correct witness trace (Theorem 2).

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 handled whenever two cooperating workers are available. A desirable property of system Task is the homogeneous distribution of the tasks among the workers.

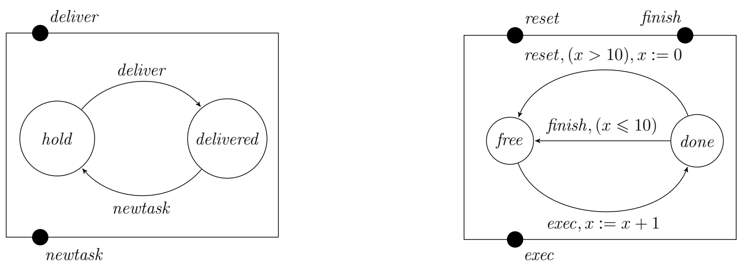
Outline. The remainder of this paper is organized as follows. Section 2 introduces some preliminary concepts. Section 3 overviews CBS design and semantics. In Section 4, we define a theoretical framework for the monitoring of multi-threaded CBSs. In Section 5, we present the transformations of multi-threaded 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 poofs related to definitions and propositions in Section 5 are given in Appendix A.

2 Preliminaries and Notations

Functions. For two domains of elements E and F, we note $[E \rightarrow F]$ the set of functions from E to F. For two functions $v \in [X \rightarrow Y]$ and $v' \in [X' \rightarrow Y']$, the substitution function noted v/v', where $v/v' \in [X \cup X' \rightarrow Y \cup Y']$, is defined as follows:

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

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 $\forall i \in [1,n] : e_i \in E$. Sequences of assignments are delimited by square brackets for clarity. The empty sequence is noted ϵ or $[\,]$, depending on context. The set of (finite) sequences over E is noted E^*. E^+ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence s is noted $\text{length}(s)$. We define $s(i)$ as the i^{th} element of s



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

Figure 2: Atomic components of system Task

and $s(i \cdots j)$ as the factor of s from the i^{th} to the j^{th} element. We also note $\text{pref}(s)$, the set of *prefixes* of s such that $\text{pref}(s) = \{s(1 \cdots k) \mid k \leq \text{length}(s)\}$. 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 $\text{last} : E^+ \to E$ such that $\text{last}(e_1 \cdot e_2 \cdots e_n) = e_n$.

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 $f : E \to F$, $\text{map } f\ e$ is the sequence of elements of F defined as $f(e_1) \cdot f(e_2) \cdots f(e_n)$ where $\forall i \in [1, n] : f(e_i) \in F$.

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 domain 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.*

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 $(Q, P, \to)$ where $Q = L \times [X \to \text{Data}]$ is the set of states, and $\to = \{((l, v), p(v_p), (l', v')) \in Q \times P \times Q \mid \exists \tau = (l, p, g_\tau, f_\tau, l') \in T : g_\tau(v) \wedge v' = f_\tau(v/v_p)\}$ is the transition relation.

A state is a pair $(l, v) \in Q$, where $l \in L$, $v \in [X \to \text{Data}]$ 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 $v = f_\tau(v'/v_p)$.

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.*

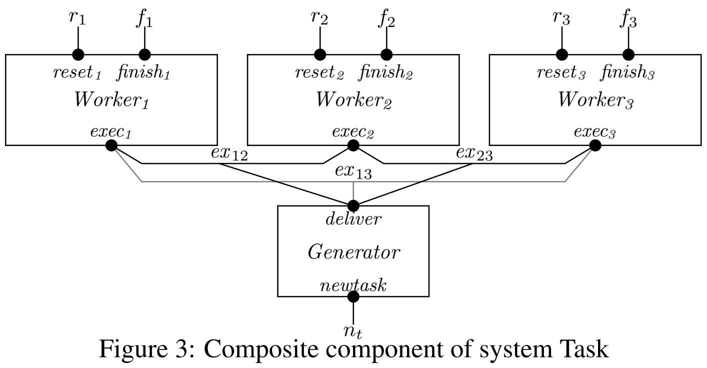


Figure 3: Composite component of system Task

- *Figure 2a depicts a model of component Generator*[2] *defined as* $Generator.P = \{deliver[\emptyset], newtask[\emptyset]\}$, $Generator.L = \{hold\ ,\ delivered\}$, $Generator.T = \{(hold, deliver, \texttt{true}, [\,], delivered), (delivered\ , newtask, \texttt{true}, [\,], hold)\}$, $Generator.X = \emptyset$.

- *Figure 2b depicts a model of worker. Component Worker is defined as* $Worker.P = \{exec[\emptyset], finish[\emptyset], reset[\emptyset]\}$, $Worker.L = \{free,\ done\}$, $Worker.T = \{(free,\ exec, \texttt{true}, [x := x+1], done), (done, finish, (x \leqslant 10), [\,], free), (done, reset, (x > 10), [x := 0], free)\}$, $Worker.X = \{x\}$.

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 assignment to the variables in $\cup_{i\in I} x_i$.*

Variables attached to ports are purposed to transfer values between interacting components. When clear from context, in the following examples, an interaction $(\{p[x_p]\}, F_a)$ consisting of only one port p is noted 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:

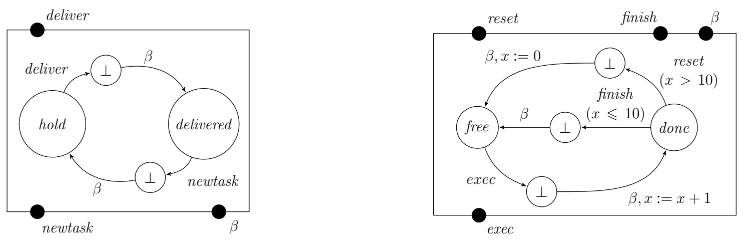
$$\frac{a = (\{p_i[x_i]\}_{i\in I}, F_a) \in \gamma \qquad \forall i \in I:\ q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{a_i}(v(X)) \qquad \forall i \notin I:\ q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

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.

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: $q_0 = Init \wedge (\forall i \in [1, s] : q_i \in Q \wedge a_i \in \gamma : q_{i-1} \xrightarrow{a_i} q_i)$, where $Init \in Q$ is the initial state. The sequence of interactions in a trace $(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s)$ is defined as $\underline{\text{interactions}}(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s) = a_1 \cdots a_s$. 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 $n_t = (\{newtask\}, [\,])$.*

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



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

Figure 4: Atomic components of system Task with partial-states

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.*

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 two transitions, a so-called *busy location* is added. Below, we define the transformation of a component with global-state semantics to a component with partial-state semantics (extending the definition in [1] with variables, guards, and computation steps on transitions).

Definition 5 (Atomic component with partial states) *The partial-state 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* $L^{\perp} \cap L = \emptyset$*,* $T^{\perp} = \{(l, p, g_\tau, [\,], l_\tau^{\perp}), (l_\tau^{\perp}, \beta, \texttt{true}, f_\tau, l') \mid \exists \tau = (l, p, g_\tau, f_\tau, l') \in T\}$ *is a 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 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 $\underline{\text{interactions}}$ 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*

[3]For the sake of simpler notation, we represent a state by its location.

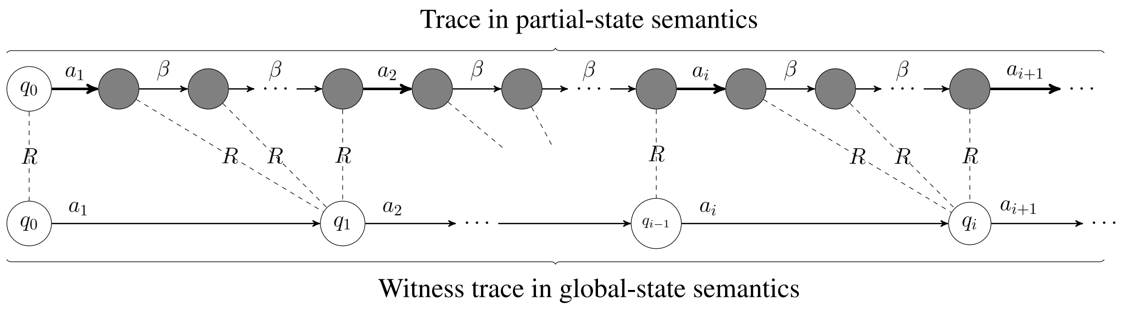

Trace in partial-state semantics
q_0
a_1
β
a_2
a_i
a_{i+1}
R
q_1
q_{i-1}
q_i
Witness trace in global-state semantics

Figure 5: Witness trace built using weak bisimulation (R)

$Worker_i^\perp$ 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. [1], Theorem 1). 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., $Q^\perp$), s.t. $R = \{(q,r) \in Q \times Q^\perp \mid r \xrightarrow{\beta^*} q\}$. 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.

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.

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 *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) *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:*

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

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

$$\begin{tikzcd}
(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold}) \arrow[r, "ex_{12}"] \arrow[d, dash, dashed, "R"] & (\perp, \perp, \mathit{free}, \perp) \arrow[r, "\beta_4"] \arrow[dr, dash, dashed, "R"] & (\perp, \perp, \mathit{free}, \mathit{delivered}) \arrow[r, "n_t"] \arrow[d, dash, dashed, "R"] & (\perp, \perp, \mathit{free}, \perp) \arrow[dr, dash, dashed, "R"] & \\
(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold}) \arrow[rr, "ex_{12}"] & & (\mathit{done}, \mathit{done}, \mathit{free}, \mathit{delivered}) \arrow[rr, "n_t"] & & (\mathit{done}, \mathit{done}, \mathit{free}, \mathit{hold})
\end{tikzcd}$$

Figure 6: An example of witness trace in system Task

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

The following property states that any trace in partial-state semantics and its witness trace have the same sequence of interactions. The proof of this property can be found in Appendix A.1.

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

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

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

We note $\mathrm{W}(\sigma_2) = \sigma_1$ when $(\sigma_1, \sigma_2) \in \mathrm{W}$. The proof of this property is given in Appendix A.2.

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, 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 out of partial states. define a function to reconstruct global states out of a partial states.

Definition 8 (Function RGT) *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* $a \in \gamma$,
- $\mathrm{upd}\Big((q_1, \ldots, q_n), (q'_1, \ldots, q'_n)\Big) = (q''_1, \ldots, q''_n)$,

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

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	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp)$	$(free, free, free, hold) \cdot ex_{12}$
2	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp) \cdot \beta_4 \cdot (\perp, \perp, free, delivered)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, delivered)$	$(free, free, free, hold) \cdot ex_{12}$
3	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp) \cdot \beta_4 \cdot (\perp, \perp, free, delivered) \cdot n_t \cdot (\perp, \perp, free, \perp)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, delivered) \cdot n_t \cdot (\perp, \perp, free, \perp)$	$(free, free, free, hold) \cdot ex_{12}$
4	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp) \cdot \beta_4 \cdot (\perp, \perp, free, delivered) \cdot n_t \cdot (\perp, \perp, free, \perp) \cdot \beta_2 \cdot (\perp, done, free, \perp)$	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, done, free, delivered) \cdot n_t \cdot (\perp, done, free, \perp)$	$(free, free, free, hold) \cdot ex_{12}$
5	$(free, free, free, hold) \cdot ex_{12} \cdot (\perp, \perp, free, \perp) \cdot \beta_4 \cdot (\perp, \perp, free, delivered) \cdot n_t \cdot (\perp, \perp, free, \perp) \cdot \beta_2 \cdot (\perp, done, free, \perp) \cdot \beta_1 \cdot (done, done, free, \perp)$	$(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t \cdot (done, done, free, \perp)$	$(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t$

Function RGT uses sub-functions acc and discriminant. First, acc 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. 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 $(free, free, free, hold)$ 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 $ex_{12} \cdot (\perp, \perp, free, \perp)$ to traces σ and $\mathrm{acc}(\sigma)$. At step 2, the state after β_4 has fresh information on component Generator which is used to update the existing partial states, so that $(\perp, \perp, free, \perp)$ is updated to $(\perp, \perp, free, delivered)$. At step 5, $Worker_1$ becomes ready after β_1, and the partial state $(\perp, done, free, delivered)$ in the intermediate step is updated to the global state $(done, done, free, delivered)$, therefore it appears in the output trace.*

The following proposition states that 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 (Computation of the witness with RGT)** $\forall \sigma \in \mathrm{Tr}(B^{\perp})$:

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

The proof is given in Appendix A.4. Theorem 1 distinguishes two cases:

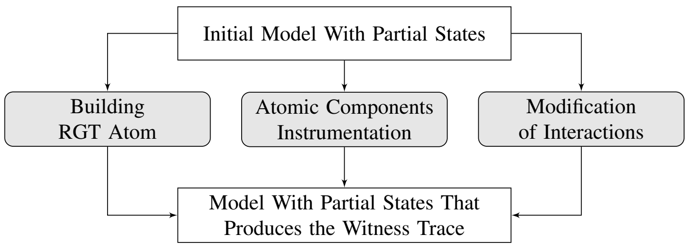

Initial Model With Partial States
Building RGT Atom
Atomic Components Instrumentation
Modification of Interactions
Model With Partial States That Produces the Witness Trace

Figure 7: Model transformation

- 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, 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.

5 Model Transformation

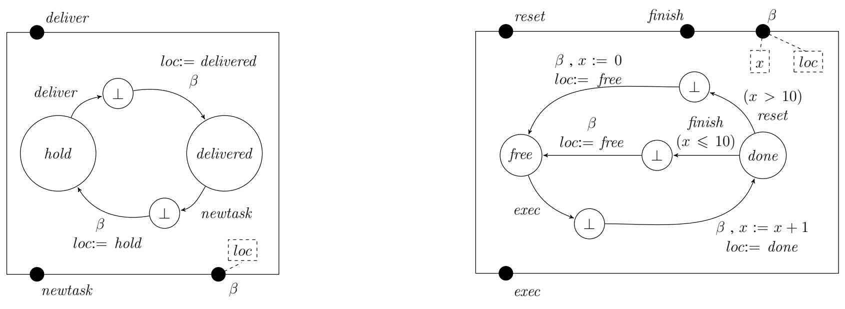
We propose a model transformation of a composite component $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ such that it can produce the witness trace on-the-fly. The transformed system can be plugged to a runtime monitor as described in [8]. Our model transformation consists of three steps: 1) instrumentation of atomic components (Sec. 5.1), 2) construction of a new component (RGT) which implements Definition 8 (Sec. 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 (Sec. 5.3).

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* $\beta^r = \beta[X^r]$;
- $T^r = \{(l, p, g_\tau, [\,], l_\tau^\perp), (l_\tau^\perp, \beta, \texttt{true}, f_\tau; [loc := l'], l') \mid (l, p, g_\tau, [\,], l_\tau^\perp),\ (l_\tau^\perp, \beta, \texttt{true}, f_\tau, l') \in T^\perp\}$.



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

Figure 8: Instrumented atomic components of system Task

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

- *Figure 8a depicts component task generator, where* $Generator^r.P^r = \{deliver[\emptyset], newtask[\emptyset], \beta[\{loc\}]\}$, $Generator^r.T^r = \{(hold, deliver, \mathtt{true}, [\,], \perp), (\perp, \beta, \mathtt{true}, [loc := delivered], delivered), (delivered, newtask, \mathtt{true}, [\,], \perp), (\perp, \beta, \mathtt{true}, [loc := hold], hold)\}$, $Generator^r.X^r = \{loc\}$.

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

5.2 Creating a New Atomic Component to Reconstruct Global States

Let us consider a composite component $B^{\perp} = \gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$ with partial-state semantics, such that:

- 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.*

- $T = T_{\text{new}} \cup T_{\text{upd}} \cup T_{\text{out}}$, *where:*
  - $T_{\text{new}} = \{(l, p_a, \texttt{true}, \texttt{new}(a), l) \mid a \in \gamma\}$,
  - $T_{\text{upd}} = \{(l, \beta_i, \bigwedge_{a\in\gamma}(\neg gs_a), \texttt{upd}(i), l) \mid i \in [1, n]\}$,
  - $T_{\text{out}} = \{(l, p'_a, gs_a, \texttt{get}, l) \mid a \in \gamma\}$.

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 `null`. 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}. The guard of these transitions always holds true and 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 which 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 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. Then, if the m^{th} tuple of V, associated to $a \in \gamma$, becomes a global state and has no `null` element, then the corresponding variable gs_a is set to `true`.
Algorithm `get` (see Algorithm 4) is called whenever component RGT has a reconstructed global state to

Algorithm 1 `new`(a)

```
1:  for $i = 1 \rightarrow n$ do
2:      if $B_i.P \cap a \neq \emptyset$ then          ▷ Checks if component $B_i$ is involved in interaction $a$.
3:          $z_i :=$ true                                ▷ In case component $B_i$ is busy, $z_i$ is true.
4:          $v_i :=$ 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 $V$.
```

Algorithm 2 `upd`(i)

```
1: $z_i :=$ false
2: for $j = 1 \rightarrow \text{length}(V)$ do
3:     if $V(j)_i ==$ null then            ▷ The $i^{\text{th}}$ element of the $j^{\text{th}}$ tuple in $V$ is represented by $V(j)_i$.
4:         $V(j)_i := B_i^r.X^r$            ▷ Updates the null states.
5:     end if
6: end for
7: check                                    ▷ Checks if the $m^{\text{th}}$ tuple of sequence $V$ is a global state.(Algorithm. 3)
```

Algorithm 3 `check`

```
1: $B_{\text{tmp}} :=$ true                                      ▷ Makes a temporary boolean variable initialized to $true$.
2: for $i = 1 \rightarrow n$ do
3:     $B_{\text{tmp}} := B_{\text{tmp}} \wedge (V(m)_i \neq \texttt{null})$    ▷ $B_{\text{tmp}}$ remains true until a null is found in the $m^{\text{th}}$ tuple of $V$.
4: end for
5: $gs_{(V(m)_{n+1})} := B_{\text{tmp}}$                          ▷ Updates the corresponding $gs_a$.
```

Algorithm 4 `get`

```
1: for $i = 1 \rightarrow n$ do
2:     $B_i^r.X_c^r := V(m)_i$                 ▷ Copies the $m^{\text{th}}$ tuple of $V$.
3: end for
4: $gs_{(V(m)_{n+1})} :=$ false                ▷ Resets the corresponding $gs_a$ of the $V(m)$.
5: $m := m + 1$                                ▷ Increment $m$.
6: check                                        ▷ Checks if the $m^{\text{th}}$ tuple of sequence $V$ is a global state. (Algorithm 3)
```

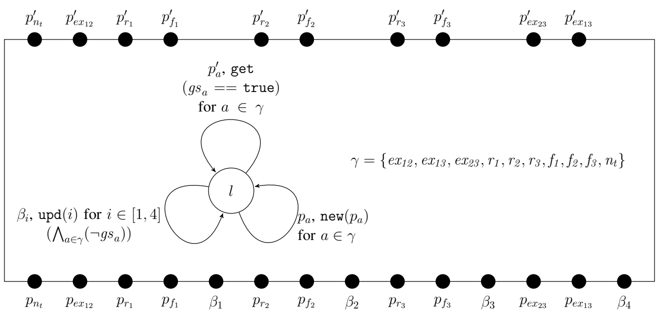


Figure 9: Component RGT for system Task

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 and also checks for the existence of any reconstructed global state and updates gs_a for $a \in \gamma$. Note, to facilitate the presentation of proofs in Section A, component RGT is defined in such a way that component RGT 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 monitor.

Example 8 *Figure 9 depicts the component* RGT *for system Task. For space reasons, only one instance of each type of transitions is shown.*

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 $B^r = B_i^r \in \{B_1^r, \cdots, B_n^r\}$, 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:*

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

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

- *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_γ^r and a_β^r. The set a^m is the set of all unary interactions a' associated to each existing interaction $a \in \gamma$ in the system.

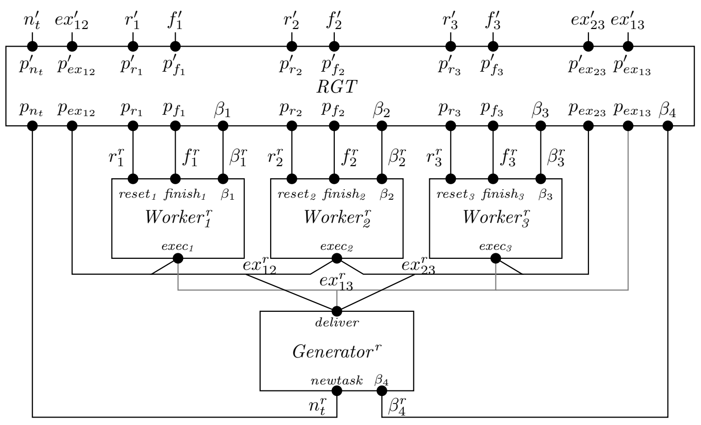

n'_t ex'_{12} r'_1 f'_1 r'_2 f'_2 r'_3 f'_3 ex'_{23} ex'_{13}
p'_{n_t} $p'_{ex_{12}}$ p'_{r_1} p'_{f_1} p'_{r_2} p'_{f_2} p'_{r_3} p'_{f_3} $p'_{ex_{23}}$ $p'_{ex_{13}}$
RGT
p_{n_t} $p_{ex_{12}}$ p_{r_1} p_{f_1} β_1 p_{r_2} p_{f_2} β_2 p_{r_3} p_{f_3} β_3 $p_{ex_{23}}$ $p_{ex_{13}}$ β_4
r^r_1 f^r_1 β^r_1 r^r_2 f^r_2 β^r_2 r^r_3 f^r_3 β^r_3
$reset_1$ $finish_1$ β_1
$Worker^r_1$
$exec_1$
$reset_2$ $finish_2$ β_2
$Worker^r_2$
$exec_2$
$reset_3$ $finish_3$ β_3
$Worker^r_3$
$exec_3$
ex^r_{12} ex^r_{23} ex^r_{13}
$deliver$
$Generator^r$
$newtask$ β_4
n^r_t β^r_4

Figure 10: Transformed composite component of system Task

Example 9 (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 and Monitoring

Combined together, the transformations preserve the semantics of the initial model as stated by the following propositions.

Intuitively, the component RGT defined in Definition 10 implements function RGT defined in Definition 8. Reconstructed global states are transferable through the ports $p'_{a\in\gamma}$. 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.

***Proposition* 1 (Correctness of component** RGT**)** *For any execution, at any time, variable* $\mathrm{RGT}.V$ *encodes the witness trace of the current execution:* $\mathrm{RGT}.V$ *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.*

The proposition is formalized in Appendix A.6.
Proof: The proof is done in Appendix A.6.

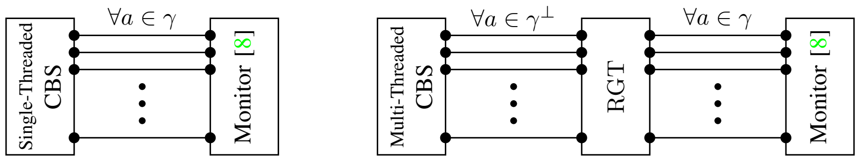
For each trace resulting from an execution with partial-state semantics, component RGT produces a trace of global states which is the witness of this trace in the initial model, as stated by the following theorem.

***Theorem* 2 (Transformation Correctness)** $\gamma^{\perp}(B^{\perp}_1, \ldots, B^{\perp}_n) \sim \gamma^r(B^r_1, \ldots, B^r_n, RGT)$.

Proof: The proof is done in Appendix A.7

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.

Connecting a monitor. As it is shown in Figure 11, one can reuse the results in [8] 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

Figure 11: Abstract presentation of runtime monitoring of centralized CBSs

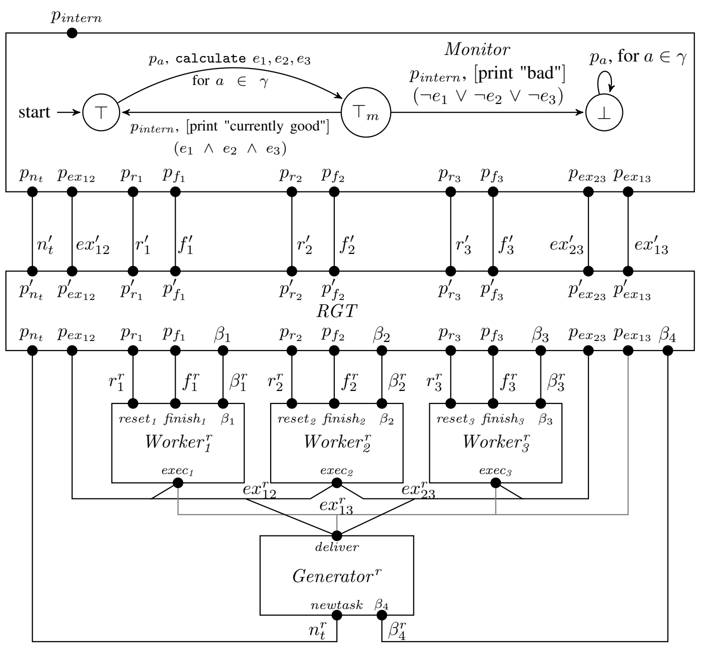

p_intern
Monitor
start
⊤
⊤_m
⊥
p_a, calculate e1, e2, e3 for a ∈ γ
p_intern, [print "currently good"] (e1 ∧ e2 ∧ e3)
p_intern, [print "bad"] (¬e1 ∨ ¬e2 ∨ ¬e3)
p_a, for a ∈ γ
RGT
Worker1r
Worker2r
Worker3r
Generator^r
deliver
newtask
exec1
exec2
exec3

Figure 12: Monitored version of system Task

Example 10 (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 $Workers$. 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 until reaching bad state $\perp$.*

6 Implementation and Evaluation

We present RVMT-BIP [4], a prototype tool integrated in the BIP tool suite (Section 6.1). BIP (Behavior, Interaction, Priority) framework is a powerful and expressive framework for the formal construction of heterogeneous systems [2]. In Section 6.2 we present the systems of our case studies. We experiment RVMT-BIP on Demosaicing system, Reader-Writer system and system Task. Each system is monitored against several properties. In Section 6.3, we present the experimental results and discuss the performance of RVMT-BIP. In Section 6.4, we discuss the evaluation of the functional correctness of RVMT-BIP.

[4]RVMT-BIP is available for download at [11].

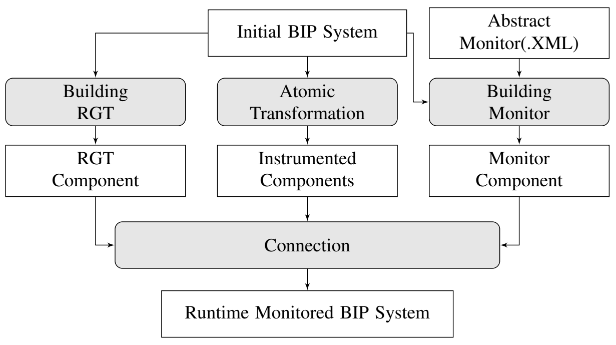


Figure 13: Overview of RVMT-BIP work-flow

6.1 RVMT-BIP: a Tool for Runtime Monitoring of Multi-Threaded Component-based Systems

RVMT-BIP (Runtime Verification of Multi-Threaded BIP) is a Java implementation ($\sim$ 2,200 LOC) of the transformation described in Section 4, and, is part of BIP distribution. RVMT-BIP takes as input a BIP system and a monitor description (an XML file), and outputs a new BIP system whose behavior is monitored while running with a multi-threaded centralized controller. RVMT-BIP uses the following modules (see Figure 13):

- *Atomic Transformation*: this module instruments the atomic components. It takes as input a BIP file containing the original BIP system and returns a transformed atomic component.
- *Building RGT*: this module takes as input the original BIP system and outputs the corresponding atomic components RGT.
- *Building Monitor:* this module takes as input a BIP file containing the original BIP system and a monitor description and then outputs the atomic component implementing the monitor (following [7, 8]).
- *Connections:* this module constructs the new composite and monitored component. The module takes as input the output from the *Atomic Transformation*, *Building RGT* and *Building Monitor* modules and then outputs a new composite component with new connections as described in Section 5.3.

6.2 Case Studies

We present some case studies on executable BIP systems conducted with RVMT-BIP. Executing these systems with a multi-threaded controller results in a faster run since the systems benefit from the parallel threads. However, these systems can also execute with a single-threaded controller which force them to run sequentially.

6.2.1 Demosaicing

Demosaicing is an algorithm[5] for digital image processing used to reconstruct a full color image from the incomplete color samples output from an image sensor. Demosaicing works on 5×5 matrices. The resulting pixels are the resulting averages of centered points of each matrix, which results to the loss of four lines and four columns of the initial image. 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.

[5]Demosaicing has been used in [16] for implementing multi-threaded timed CBSs.

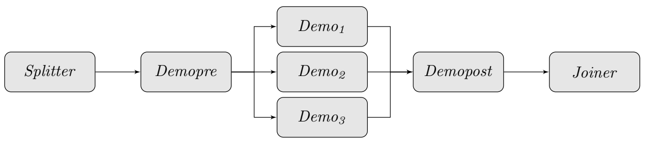
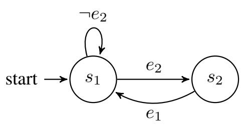
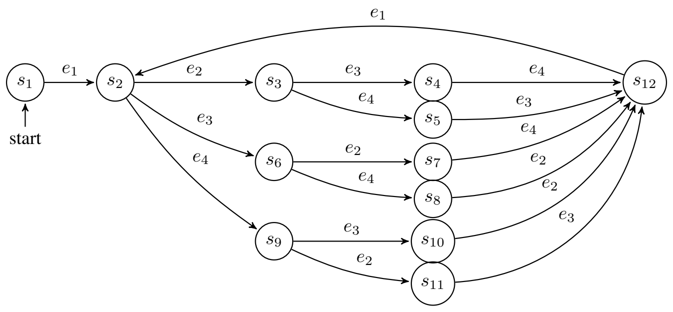


Figure 14: Processing network of system Demosaicing



(a) φ_1



(b) φ_2

Figure 15: Automata of properties of demosaicing

Specifying process completion. We consider two properties for the demosaicing system related to process completion.

1. It is necessary that all the internal demosaicing units finish their process before post-demosaicing unit start processing. 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 demosaicing process completes. This requirement is formalized as property φ_1 defined by the automaton depicted in Figure 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.

2. Moreover, internal demosaicing units ($Demo_1$, $Demo_2$, $Demo_3$) should not start demosaicing process until pre-demosaicing unit finishes its process. 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 Figure 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.

6.2.2 Reader-Writer

The Reader-Writer system consists of four components: a $Reader$, a $Writer$, a $Clock$ and a $Poster$. $Reader$ 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 Figure 16.

Specification of data freshness. 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

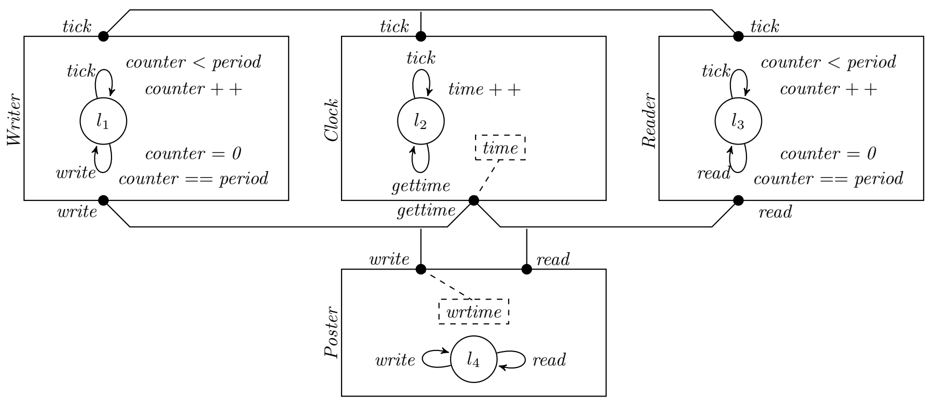


Figure 16: Model of system Reader-Writer

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 Figure 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 $Poster$.

Specification of the execution order. A more complex specification on the execution order involves several writers. 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.

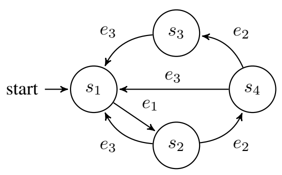
This requirement is formalized as property φ_4 which is defined by the automaton depicted in Figure 17b where:

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

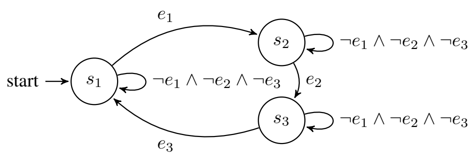
When $Writer_1$ writes to the poster, the automaton moves from initial state s_1 to state s_2. From state s_2, the automaton moves to state s_3 when $Writer_2$ writes to the poster. From state s_3, the automaton moves to the initial state s_1 when $Writer_3$ writes to the poster. This writing order must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

6.2.3 Task Management

We consider our running example system Task. We consider a specification that states the homogeneous distribution of the generated tasks among the workers. 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.



(a) φ_3



(b) φ_4

Figure 17: Automata of the properties of system Reader-Writer

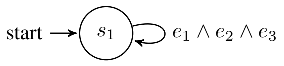


Figure 18: Automaton of the property of system Task

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 Figure 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.

6.3 Evaluating the Performance

6.3.1 Evaluation Principles

Following the work-flow depicted presented in Section 6.1 and depicted in Figure 13, for each system, and all its properties, we synthesize a BIP monitor following [8] and combine 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 number 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 force 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.

6.3.2 Results (*cf.* Table 2)

Table 2 reports results on checking *complete process* property of Demosaicing system, *data freshness* and *execution ordering* property of the Reader-Writer system and *task distribution* property of system Task. Each time measurement is an average value obtained over 100 executions of these systems. In Table 2, the columns have the following meanings:

- Column *# interactions* shows the number of functional steps of system.
- Columns *no monitor* report the execution time of the systems without monitors when varying the number of threads.
- Columns *with monitor* report the execution time of the systems with monitors when varying the number of threads, the number of additional interactions and overhead induced by monitoring. Column *events* indicates the number of reconstructed global states (events sent to the associated monitor).

Table 2: Results of monitoring with RVMT-BIP

system	# interactions	no monitor		with monitor						
		thread	time (s)	specification		# extra interactions	events	thread	time (s)	overhead (%)
Demosaicing	8,400	1	67.97	Process completion	φ_1	6,800	4,399	1	68.706	1.07
								3	41.245	1.53
		3	40.62					10	29.521	1.24
					φ_2	2,200	1,599	1	69.116	1.67
		10	29.15					3	41.235	1.51
								10	29.251	0.31
Reader-Writer	120,000	1	613.78	Data freshness	φ_3	120,000	39,999	1	617.59	0.62
		3	204.96					3	207.79	1.37
Reader-Writer (3 writer)	40,000	1	154.07	Execution ordering	φ_4	130,000	39,999	1	157.22	2.04
		2	102.9					2	105.64	2.66
		3	57.05					3	57.99	1.63
Task (100,000 tasks)	399,999	1	117.96	Task distribution	φ_5	200,198	100,197	1	121.12	2.67
		2	72.32					2	72.85	0.73

Table 3: Results of monitoring with RV-BIP

system	# interactions	no monitor		with monitor					
		thread	time (s)	specification		# extra interactions	thread	time (s)	overhead (%)
Demosaicing	8,400	1	67.97	Process completion	φ_2	3,202	1	175.83	158.6
		10	29.15				10	172.31	491.1
Task (100,000 tasks)	399,999	1	117.96	Task distribution	φ_5	177,611	1	126.11	6.91
		2	72.32				2	105.66	46.1

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.

RV-BIP vs. RVMT-BIP. To illustrate the advantages of monitoring multi-threaded systems with RVMT-BIP , we made a comparison between our new monitoring technique and RV-BIP proposed in [8]. Table 3 shows the results of a performance evaluation of monitoring Demosaicing and Task with RV-BIP. RV-BIP induces a cheap overhead of 6.91% with one thread and a huge overhead of 46.1% (which is mainly caused by globally-synchronous extra interactions introduced by RV-BIP) with two threads, whereas according to Table 2, the overhead induced by RVMT-BIP with two threads is 0.73%. The induced overhead is even better than the overhead induced when monitoring the single-threaded version of the system which is 2.67%. As can be seen in Table 3, RVMT-BIP outperforms RV-BIP when monitoring Demosaicing. The latter does not take any advantage of the parallel execution. This clearly demonstrates the advantages of our monitoring approach over [8].

6.4 Evaluating Functional Correctness

In this section we check the functional correctness of our tools, that is whether the verdicts of the monitors inserted by RVMT-BIP are sound and complete. Each of the system initially presented in this section is correct by design, and we run monitored versions of these for several hours without any error reported. To assess the error detection of the monitor, we built mutated versions of the systems eventually leading to a violation of the properties[6]. We built one mutant per pair of system and properties. Our monitors were able to detect and kill all the mutant. We also evaluated RVMT-BIP on several systems in the BIP distribution, and in particular non-deterministic models such as the dining philosopher model against deadlock-freedom.

[6]We do not report on the performance on monitoring mutated versions of the systems as, even if the systems produced an error eventually, the occurrence time of the error was non-deterministic

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

In [7, 8], we proposed a first approach for the runtime verification of CBSs. The approach in [7, 8] 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 [7, 8] 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 [7, 8] when dealing with multi-threaded CBSs: a monitor as synthesized in [7, 8] can be plugged to the component reconstructing the global states of the system proposed in this paper.

7.2 Decentralized Runtime Verification

The approaches in [3, 6] 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 trace are only partial w.r.t. the global behavior of the system.

7.3 Monitoring Safety Properties in Concurrent Systems

The approach in [14] 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 [14] 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 [14] 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.

7.4 Parallel Runtime Verification of Monolithic Sequential Programs

Berkovich et al. [4] introduce parallel algorithms for the runtime verification of sequential programs against LTL formulas using a graphics processing unit (GPU). Monitoring threads are added to the program and directly execute on the GPU. Our approach differs from [4] in that we do not target monolithic sequential programs but concurrent and multi-threaded CBSs. Moreover, as shown by our experiments, our approach mostly preserves the performance of the monitored system, while [4] adds significant computing power to the system to handle the monitoring overhead. Finally, as explained in the previous subsection, our approach is not bound to any particular logic, and allows for Turing-complete monitors.

8 Conclusions and 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.

Note, to simplify the presentation of the approach, we assumed that the initial system is deterministic. Considering non-deterministic systems as input does not fundamentally modify the results of this paper. It involves to adapt the notion of witness trace and its computation with component RGT. If the initial system is non-deterministic, by observing a trace in partial-state semantics, one should compute *the set of witnesses traces compatible with the observed trace*. Such computation can be done using the model of the system described in global-state semantics.

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 [5]. Many work has been done in order to monitor properties on a distributed (monolithic) systems; e.g., [13] for online monitoring of CTL properties, [10] for online monitoring of LTL properties, [12] for offline monitoring of properties expressed in a variant of CTL, and [15] 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 for timed components and timed specifications as presented in [2].

References

[1] Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5048, pp. 116–133. Springer (2008) 1, **??**, 4.1, 4.1

[2] Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India. pp. 3–12. IEEE Computer Society (2006) 6, 8.2, A.2

[3] Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 85–100. Springer (2012) 7.2

[4] Berkovich, S., Bonakdarpour, B., Fischmeister, S.: GPU-based runtime verification. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013. pp. 1025–1036 (2013) 7.4

[5] Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing 25(5), 383–409 (2012) 8.2

[6] Falcone, Y., Cornebize, T., Fernandez, J.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part

of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8461, pp. 66–83. Springer (2014) 7.2

[7] Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: SEFM 2011. pp. 204–220 (2011) 6.1, 7.1

[8] Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling 14(1), 173–199 (2015) 1, **??**, 5, 5.4, **??**, **??**, 6.1, 6.3.1, 6.3.2, 7.1

[9] Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989) 1

[10] Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015. pp. 494–503. IEEE Computer Society (2015) 8.2

[11] Nazarpour, H.: Runtime Verification of Multi-Threaded BIP. `http://www-verimag.imag.fr/~nazarpou/rvmt.html` 1, 4

[12] Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: Papatriantafilou, M., Hunel, P. (eds.) Principles of Distributed Systems, 7th International Conference, OPODIS 2003 La Martinique, French West Indies, December 10-13, 2003 Revised Selected Papers. Lecture Notes in Computer Science, vol. 3144, pp. 171–183. Springer (2003) 8.2

[13] Sen, A., Garg, V.K.: Formal verification of simulation traces using computation slicing. IEEE Trans. Computers 56(4), 511–527 (2007) 8.2

[14] Sen, K., Vardhan, A., Agha, G., Rosu, G.: 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) 7.3

[15] Tomlinson, A.I., Garg, V.K.: Monitoring functions on global states of distributed programs. Journal of Parallel and Distributed Computing 41(2), 173–189 (1997) 8.2

[16] Triki, A., Bonakdarpour, B., Combaz, J., Bensalem, S.: Automated conflict-free concurrent implementation of timed component-based models. In: NASA Formal Methods, pp. 359–374. Springer (2015) 5

A Correctness Proof of the Approach

Before tacking 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 one concerns the witness trace. Given a CBS B which 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 above mentioned relation between a system in global-state semantics and the corresponding system in partial-state semantics. Property 2 that from any execution in partial-state semantics, the witness exists and is unique.

The second one states specifies a function for building the witness trace from a trace in partial-state semantics in an online fashion. Theorem 1 states the correctness of this function.

The third one states that the transformed components, the synthesized components, and their connection are correct. That is, the obtained system i) computes the witness and implements the above function (Proposition 1), 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 inSection A.1. The proof of Property 2 is in Section A.2. Some intermediate lemmas with their proofs are introduced in Section A.3 in order to prove Theorem 1 in Section A.4. Some intermediate definitions and lemmas with their proofs are given in Section A.5 in order to prove Theorem 2 in Section A.7.

A.1 Proof of Property 1

We shall prove that:

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

where W is the witness relation defined in Definition 7, and $\underline{\mathrm{interactions}}(\sigma)$ is the sequence of interactions of trace σ.

Proof: The proof is done by induction on the length of the sequence of interactions.

- Base case. By definition of W, $(\mathit{Init}, \mathit{Init}) \in W$ and $\underline{\mathrm{interactions}}(\mathit{Init}) = \epsilon$. The property holds vacuously.
- Induction case. Let us consider $(\sigma_1, \sigma_2) \in W$ and suppose that $\underline{\mathrm{interactions}}(\sigma_1) = \underline{\mathrm{interactions}}(\sigma_2)$. According to the definition of the witness relation, from $(\sigma_1, \sigma_2) \in W$, the only possible way to add an interaction to σ_1 and σ_2 and obtain sequences related by the witness relation is to apply the first rule. Moreover, the first rule adds the same interaction to both sequences.

A.2 Proof of Property 2

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: From the weak bi-simulation of a global-state semantics model with its corresponding partial-state semantics model [2], 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, the sequences of interactions of σ_1 and σ_1' are both equal to the sequence of interactions of σ_2. Therefore, the sequences of interactions of σ_1 and σ_1' are equal. From the sequence of interaction, using the determinism of the system, we can relate the sequence of interactions to a unique trace, from a unique initial state.

A.3 Intermediate Lemmas

We give some intermediate lemmas that are needed to prove Theorem 1.

Lemma 1 *$\forall(\sigma_1, \sigma_2) \in W : |\mathrm{acc}(\sigma_2)| = |\sigma_1| = 2s + 1$, where $s = |\underline{\mathrm{interactions}}(\sigma_1)$, where acc is the accumulator used in the definition of function RGT (Definition 8), and function $\underline{\mathrm{interactions}}$ (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).

Proof: The proof is done by induction on the length of the trace in partial-state semantics, i.e., σ_2.

- Base case: According to the definition of function acc (see Definition 8), Lemma 1 holds for the initial state of the systems. In this case, $\sigma_1 = \sigma_2 = \mathrm{acc}(\sigma_2) = \mathit{Init}$ and $|\mathrm{acc}(\sigma_2)| = |\sigma_1| = 1$.
- Induction case: Let us consider $(\sigma_1, \sigma_2) \in W$ and let us suppose that Lemma 1 holds for some trace σ_2 such that $\underline{\mathrm{interactions}}(\sigma_2) = s$. According to the definition of function acc, only the execution of a new interaction $a \in \gamma$ can increase in the length of $\mathrm{acc}(\sigma_2)$. After a, two elements are added to $\mathrm{acc}(\sigma_2)$: the first is the interaction a and the second is the partial state obtained after a. Since σ_1 is

the witness of σ_2, executing a in the partial-state model implies that a is executed in the global-state model. In the global-state model, after the execution of a, the length of σ_1 is also increased by two, such that interaction a along with a global state obtained after a are added to σ_1.

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

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

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. Moreover, the sequence of global states produced by function acc follows the global-state semantics.

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.

Lemma 3 *$\forall \sigma \in \mathrm{Tr}(B^{\perp})$,*

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

where acc *is the accumulator function and* 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: According to the definition of function 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 $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ is always lesser than or equal to the length of $\mathrm{acc}(\sigma)$. Moreover, according to Lemma 2, all the states of $\mathrm{discriminant}(\mathrm{acc}(\sigma))$ are global states and follow the global-state semantics.

Lemma 4 *$\forall \sigma \in \mathrm{Tr}(B^{\perp}) : \mathrm{last}(\mathrm{acc}(\sigma)) = \mathrm{last}(\sigma)$.*

Proof: The proof is done by induction on the length of the trace in partial-state semantics, i.e., σ.

- 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 which 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, we can conclude that $\mathrm{upd}(q_{m+1}, q_m) = q_{m+1}$.

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

Lemma 5 $\forall \sigma \in \mathrm{Tr}(B^{\perp}) : \underline{\mathrm{interactions}}(\mathrm{acc}(\sigma)) = \underline{\mathrm{interactions}}(\sigma)$.

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

A.4 Proof of Theorem 1

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:

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

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:

- If $\mathrm{last}(\sigma) \in Q$, according to Lemma 4, $\mathrm{last}(\mathrm{acc}(\sigma)) \in Q$ 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 5, $\underline{\mathrm{interactions}}(\mathrm{acc}(\sigma)) = \underline{\mathrm{interactions}}(\sigma)$. 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 $q > 0$. 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 involved in interaction $a'_1 \in \gamma$ is still busy. Let i be the identifier of this component and a'_1 be s^{th} interaction in trace σ such that $a'_1 = \underline{\mathrm{interactions}}(\sigma)(s)$. 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 $a''_m = \underline{\mathrm{interactions}}(\sigma)(s+m)$ 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., $a''_p = \underline{\mathrm{interactions}}(\sigma)(s+p)$ 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 Intermediate Definitions and 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 $B^\perp = \gamma^\perp(B^\perp_1, \ldots, B^\perp_n)$ of behavior $(Q^\perp, \gamma^\perp, \longrightarrow)$;
- the transformed composite component $B^r = \gamma^r(B^r_1, \ldots, B^r_n, RGT)$ of behavior $(Q^r, \gamma^r, \longrightarrow_r)$. B^r is obtained from $B^\perp$ by following the transformations described in Section 5.

Definition 12 (State stability) *State $q \in RGT.Q$ is said to be* stable *when all Boolean variables in set $\{RGT.gs_a \mid a \in \gamma\}$ evaluate to* `false` *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 its associated component RGT is *stable*.

The following lemma states that any state of component RGT can be stabilized by executing busy interactions in $\beta \in a^m$.

Lemma 6 $\forall q, q' \in Q^r : q \xrightarrow{\alpha^*} q' \land \alpha \in a^m \implies \mathrm{stable}(q')$.

Proof: Let us consider a non-stable state $q \in RGT.Q$. 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, one of the variables in $\{gs_a \mid a \in \gamma\}$ evaluates to true in q. Such transitions entail to execute algorithm `get` (Algorithm 4). Algorithm `get` sets the value of the corresponding variable to false. It then call algorithm `check` which may set another variable $gs_{a'}$ associated to another interaction $a' \in a^m$ to false. After executing `get`, component RGT returns to a situation where again algorithm `get` can execute if another variable has been set to true by algorithm `check`. The above process executes until all variables $\{gs_a \mid a \in \gamma\}$ are set to false by `get` and algorithm `check` does not set any of these variables to true. Finally, component RGT reaches a stable state.

Definition 13 (Equivalent states) *Let $q^r = (q^r_1, \cdots, q^r_n, q^r_{n+1}) \in Q^r$ be a state in transformed model where q^r_{n+1} is the state of component RGT, function* equ $: Q^r \longrightarrow Q^\perp$ *is defined as follows:* $\mathrm{equ}(q^r) = q$, *where $q = (q_1, \cdots, q_n)$, $(\forall i \in [1, n] : q^r_i = q_i) \land q^r_{n+1}$is stable.*

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 13.

Lemma 7 *Let us consider two states: q of the initial model and q^r its corresponding state in the transformed model such that $\text{equ}(q^r) = q$. There exists an enabled interaction in the initial model ($a \in \gamma^\perp$) at 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 $\text{RGT}.p_a$, for $a \in \gamma$, are always enabled. Since for a given interaction a, a^r and a differ only by port $RGT.p_a$, we can conclude that $a^r \in a^r_\gamma$ is enabled if and only if $a \in \gamma^\perp$ is enabled.

Below we define the notion of equivalence between a $(n+1)$-tuple and a state of the system. The notion of equivalence is used to relate the tuples constructed by component RGT to the states of the system.

Definition 14 (Equivalence of a state and a $(n+1)$-tuple) *A state $q = (q_1, \ldots, q_n)$ is equivalent to a $(n+1)$-tuple $v = (v_1, \ldots, v_n, v_{n+1})$ if:*

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

We note $v \cong q$.

A state $(q_1, \ldots, q_n)$ and a tuple $(v_1, \ldots, v_n, v_{n+1})$ are equivalent if $v_i = q_i$ for each position i where the state q_i of component i 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, noted $\sigma \cong V$, if q_j is equivalent to $v(j)$, for all $j \in [0, k]$.

A.6 Proof of Proposition 1

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 $\sigma \in \text{Tr}(B^\perp)$, component RGT (Definition 10) implements function RGT (Definition 8), that is $\forall \sigma \in \text{Tr}(B^\perp), \text{RGT}.V \cong \text{acc}(\sigma)$.

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

- Base case. By definition of function RGT, at the initial state $\text{acc}(\mathit{Init}) = \mathit{Init}$. By definition of component RGT, V is initialized as a tuple representing the initial state of the system. Therefore, $\text{RGT}.V \cong \text{acc}(\mathit{Init})$.
- Induction case. Let us suppose that the proposition holds for a trace $\sigma \in \text{Tr}(B^\perp)$, that is $\text{RGT}.V \cong \text{acc}(\sigma)$. According to the definition of function RGT, $\text{RGT}(\sigma) = \text{discriminant}(\text{acc}(\sigma))$. Consequently, there exists $\sigma' \in \text{Tr}(B^\perp)$ of the form $\sigma' = q'_0 \cdot a'_1 \cdot q'_1 \cdots q'_k$, with $k \geqslant 0$, such that $\text{acc}(\sigma) = \text{RGT}(\sigma) \cdot \sigma'$. 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 $\text{acc}(\sigma \cdot a'_{k+1} \cdot q'_{k+1}) = \text{acc}(\sigma) \cdot a'_{k+1} \cdot q'_{k+1}$. On the other hand, in component RGT, according to Algorithm 1 (line 6), the corresponding transition $\tau \in T_{\text{new}}$ 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 $\text{RGT}.V \cong \text{acc}(\sigma)$ 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), in component RGT, the corresponding transition $\tau \in T_{\text{upd}}$ updates the sequence of tuples V such that $\text{RGT}.V \cong \text{acc}(\sigma)$ 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.7 Proof of Theorem 2

Proof: We shall prove the existence of a bi-simulation between initial and transformed model, that is relation $R = \{(q, q^r) \mid q^r \xrightarrow{\beta^*}_r z^r : \mathrm{equ}(z^r) = q\}$ satisfies the following properties for any $q \in Q$ and $q^r \in Q^r$:

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

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

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

Proof of (i):
Let us consider $q^r = (q^r_1, \cdots, q^r_n, q^r_{n+1})$, $z^r = (z^r_1, \cdots, z^r_n, z^r_{n+1})$, and $q = (q_1, \cdots, q_n)$ such that $(q, q^r) \in R$. After executing interaction α, none of the local states q^r_i, for $i \in [1, n]$, changes. Therefore, according to Lemma 6, we can conclude that $(q, z^r) \in R$.

Proof of (ii):
Let us consider $q^r = (q^r_1, \cdots, q^r_n, q^r_{n+1})$, $z^r = (z^r_1, \cdots, z^r_n, z^r_{n+1})$, $q = (q_1, \cdots, q_n)$ and $z = (z_1, \cdots, z_n)$. When $a^r \in (a^r_\gamma \cup a^r_\beta)$ is enabled, from the definition of semantics of transformed composite component, we can deduce that corresponding interaction $a \in \gamma^\perp$ is enabled. 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 $\exists \alpha \in a^m : z^r_{n+1} \xrightarrow{\alpha^*} \mathrm{stable}(z^r_{n+1})$. Therefore, $(z, z^r) \in R$.

Proof of (iii):
Let $q^r = (q^r_1, \cdots, q^r_n, q^r_{n+1})$ and $z^r = (z^r_1, \cdots, z^r_n, z^r_{n+1})$. 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, then according to Lemma 6 *RGT* will be stable after some $\beta \in a^m$ so $(z, z^r) \in R$.
    * If RGT has no global state, 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. Consequently, a^r is necessarily enabled after the execution of some interaction $\alpha \in a^m$.