Monitoring Multi-Threaded Component-Based Systems

Hosein Nazarpour, Ylièss Falcone, Saddek Bensalem, Marius Bozga, Jacques Combaz

Monitoring Multi-Threaded Component-Based Systems

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

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

Abstract. This paper addresses the monitoring of logic-independent linear-time user-provided properties 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 formal 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.

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 [9], 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 linear-time regular property, a

component monitor M and a new set of interactions γ' are synthesized 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 relies on multiparty interactions which consist of actions that are jointly executed by certain components, either sequentially or concurrently. In the sequential setting, components are coordinated by a single centralized controller and joint actions are atomic. Components notify the controller of their current states. Then, the controller computes the possible interactions, selects one, and then sequentially executes the actions of each component involved in the interaction. When components finish their executions, they notify the controller of their new states, and the aforementioned steps are repeated. For performance reasons, it is desirable to parallelize the execution of components. In the multi-threaded setting, each component executes on a thread and a controller is in charge of coordination. Parallelizing the execution of $\gamma(B_1, \ldots, B_n)$ yields a bisimilar [10] 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 linear-time properties referring to global states. Our method preserves the concurrency and semantics of the monitored system. It transforms the system so that global states can be reconstructed by accumulating partial states at runtime. The execution trace of a multi-threaded CBS is a sequence of partial states. For an execution trace of a multi-threaded CBS, we define the notion of *witness* trace, which is intuitively the unique trace of global states corresponding to the trace of the multi-threaded CBS if this CBS was executed on a single thread. For this purpose, we define transformations allowing 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. Recall that according to [1], a BIP system with global-state semantics S_g (sequential model), is (weakly) bisimilar with the corresponding partial-state model S_p (concurrent model) noted $S_g \sim S_p$. Moreover, S_p generally

[1] RVMT-BIP is available for download at [12].

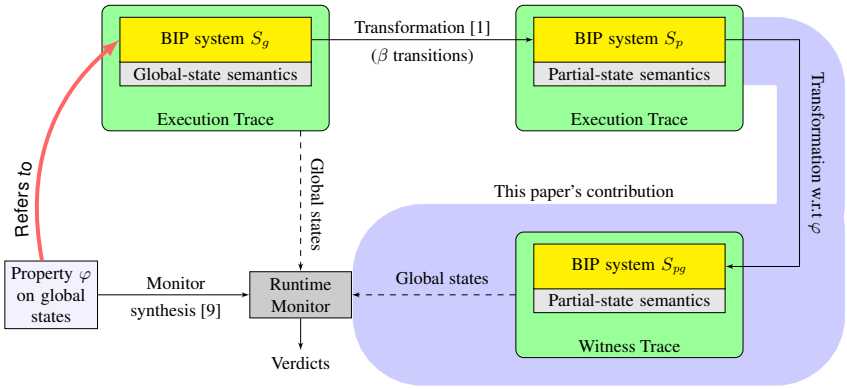


Fig. 1: Approach overview

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 [9] and run S_p, which ends up with delays in detecting verdicts or ii) to plug the monitor proposed in [9] 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 bisimilar (hence S_g and S_{pg} are bisimilar), ii) S_{pg} is as concurrent as S_p and preserves the performance gained from multi-threaded execution and iii) S_{pg} produces a witness trace, that is the 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).

An extended version of this paper with more detail and proofs is available as [13].

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.

2 Preliminaries and notations.

For two domains of elements E and F, we note $[E \to F]$ the set of functions from E to F. For two functions $v \in [X \to Y]$ and $v' \in [X' \to Y']$, the substitution function noted v/v', where $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as $v/v'(x) = v'(x)$ if $x \in X'$, and $v(x)$ otherwise. 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 the context. The set of (finite) sequences over E is noted E^*. E^+ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence s is noted $\mathrm{length}(s)$. We define $s(i)$ as the i^{th} element of s and $s(i \cdots j)$ as the factor of s from the i^{th} to the j^{th} element. We also note $\mathrm{pref}(s)$, the set of *prefixes* of s s.t. $\mathrm{pref}(s) = \{s(1 \cdots k) \mid k \leq \mathrm{length}(s)\}$. Operator pref

is naturally extended to sets of sequences. Function $\max_{\preceq}$ (resp. $\min_{\preceq}$) returns the maximal (resp. minimal) sequence w.r.t. prefix ordering of a set of sequences. We define function $\text{last} : E^+ \to E$ s.t. $\text{last}(e_1 \cdot e_2 \cdots e_n) = e_n$. For a sequence $e = e_1 \cdot e_2 \cdots e_n$ over E, and a function $f : E \to F$, $\text{map } f\ e$ is the sequence over 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.

An *atomic component* is endowed with a finite set of local variables X taking values in a domain Data, and it synchronizes with other components through *ports*. A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier p and some variables in a set x_p.

Definition 1 (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.*

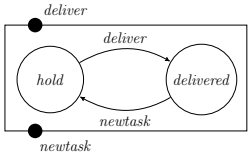
The semantics of the atomic component is an LTS $(Q, P, \to)$ where $Q = L \times [X \to \text{Data}]$ is the set of states, and $\to = \{((l, v), p(v_p), (l', v')) \in Q \times P \times Q \mid \exists \tau = (l, p, g_\tau, f_\tau, l') \in T : g_\tau(v) \wedge v' = f_\tau(v/v_p)\}$ is the transition 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 $(l, p[x_p], g_\tau, f_\tau, l')$, s.t. $g_\tau(v) = \texttt{true}$. As a result, the valuation v of variables is modified to $v' = f_\tau(v/v_p)$.

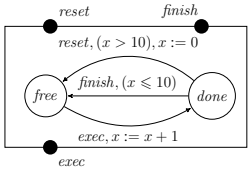
We use the dot notation to denote the elements of atomic components. e.g., for a component B, $B.P$ denotes the set of ports of the atomic component B, etc. Figure 2 depicts atomic components of system Task.

Definition 2 (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 s.t. $\forall i \in I : \mathcal{P}_a \cap B_i.P = \{p_i\}$ 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 the context, in the following examples, an interaction $(\{p[x_p]\}, F_a)$ consisting of only one port p is noted p.



(a) Component *Generator*



(b) Component *Worker*

Fig. 2: Atomic components

Definition 3 (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 $\gamma(B_1, \ldots, B_n)$ 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 $(q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s)$ s.t.: $q_0 = Init \wedge (\forall i \in [1, s] : q_i \in Q \wedge a_i \in \gamma : q_{i-1} \xrightarrow{a_i} q_i)$, where $Init \in Q$ is the initial state. The sequence of interactions is then 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 1 (Interaction, composite component). Figure 3 depicts the composite component $\gamma(Worker_1, Worker_2, Worker_3, Generator)$ of system Task. The set of interactions is $\gamma = \{ex_{12}, ex_{13}, ex_{23}, r_1, r_2, r_3, f_1, f_2, f_3, n_t\}$. For instance, we have $ex_{12} = (\{deliver, exec_1, exec_2\}, [\,])$.

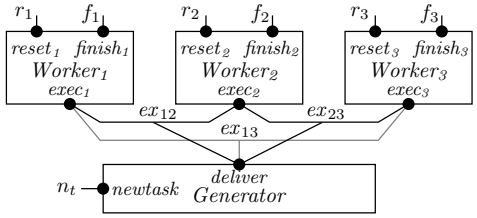


Fig. 3: Composite component of system Task

One of the possible traces[2] of system Task is: $(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t \cdot (done, done, free, hold)$ s.t. 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 semantics defined in Sec. 3 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 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

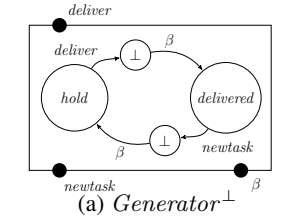
[2] For the sake of simpler notation, we represent a state by its location.

β-labeled transition (aka busy transition). Between these 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).

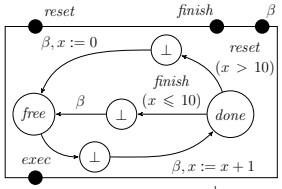
Definition 4 (Components with partial states). *The partial-state version of atomic component $B = (P, L, T, X)$ is $B^\perp = (P \cup \{\beta\},\ L \cup L^\perp,\ T^\perp,\ X)$, where $\beta \notin P$ is a special port, $L^\perp = \{l_t^\perp \mid t \in T\}$ (resp. L) is the set of busy locations (resp. ready location) s.t. $L^\perp \cap L = \emptyset$, $T^\perp = \{(l, p, g_\tau, [\,], l_\tau^\perp), (l_\tau^\perp, \beta, \texttt{true}, f_\tau, l') \mid \exists \tau = (l, p, g_\tau, f_\tau, l') \in T\}$ is a set of transitions.*

Assuming some atomic components with partial-state semantics $B_1^\perp, \ldots, B_n^\perp$, we construct a composite component $B^\perp = \gamma^\perp(B_1^\perp, ..., B_n^\perp)$ 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-state semantics in the natural way. We extend the definition of interactions to traces in partial-state semantics s.t. $\beta_{i\in[1,n]}$ are filtered out.

Example 2 (Composite component with partial states). The corresponding composite component of Task with partial-state semantics is $\gamma^\perp(Worker_1^\perp,\ Worker_2^\perp,\ Worker_3^\perp,\ Generator^\perp)$, where each $Worker_i^\perp$ for $i \in [1, 3]$ is identical to the component in Fig. 4b and $Generator^\perp$ is the component in Fig. 4a. We represent each busy location $l^\perp$ as $\perp$.



(a) $Generator^\perp$



(b) $Worker^\perp$

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

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-state 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.

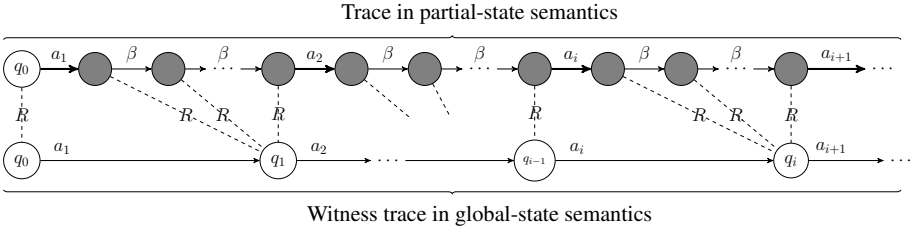


Fig. 5: Witness trace built using weak bisimulation (R)

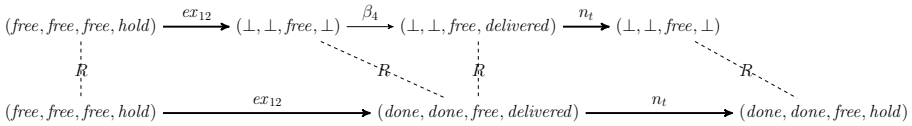


Fig. 6: An example of witness trace in system Task

Definition 5 (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 trace *of* σ_2*.*

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

Example 3 (Witness relation and trace). 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.

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

Property 1 states that any trace in partial-state semantics and its witness trace have the same sequence of interactions. Property 2 states that any trace in the partial-state semantics has a unique witness trace in the global-state semantics.

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

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

Following Property2, we note $\mathrm{W}(\sigma_2) = \sigma_1$ when $(\sigma_1, \sigma_2) \in \mathrm{W}$.

Note that, when running a system in partial-state semantics, the global state of the witness trace after an interaction a is not known until all the components involved in a

have reached their ready locations after the execution of a. Nevertheless, even in non-deterministic systems, 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).

Definition 6 (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 $\forall k \in [1, n], q''_k = \begin{cases} q_k & \text{if } (q_k \notin Q_k^{\perp}) \wedge (q'_k \in Q_k^{\perp}) \\ q'_k & \text{otherwise.} \end{cases}$

Function RGT uses sub-functions acc and $\mathrm{discriminant}$. 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 $\mathrm{discriminant}$ 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 which can incrementally generate the witness trace of a running system by monitoring interactions and partial states of components.

Example 4 (Applying function RGT*).* Table 1 illustrates Definition 6 on one trace of system Task with initial state $(\mathit{free}, \mathit{free}, \mathit{free}, \mathit{hold})$ followed by interactions ex_{12}, β_4, n_t, β_2, and β_1. At step 0, the outputs of functions acc and $\mathrm{discriminant}$ are equal to the initial state. At step 1, the execution of interaction ex_{12} adds $ex_{12} \cdot (\perp, \perp, \mathit{free}, \perp)$ to traces σ and $\mathrm{acc}(\sigma)$. At step 2, the state after β_4 has fresh information on component $\mathit{Generator}$ which is used to update the existing partial states, so that $(\perp, \perp, \mathit{free}, \perp)$ is updated to $(\perp, \perp, \mathit{free}, \mathit{delivered})$. At step 5, Worker_1 becomes ready after β_1, and the partial state $(\perp, \mathit{done}, \mathit{free}, \mathit{delivered})$ in the intermediate step is updated to the global state $(\mathit{done}, \mathit{done}, \mathit{free}, \mathit{delivered})$, therefore it appears in the output trace.

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.

Table 1: Values of function RGT for a sample input

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

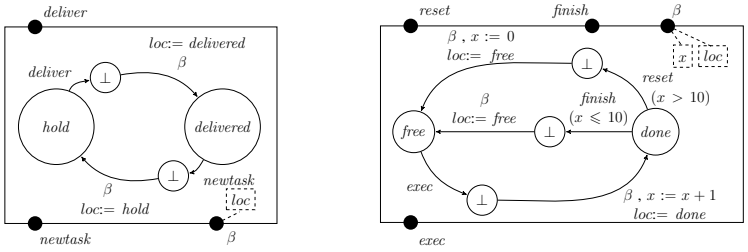
Theorem 1 (Computation of the witness with RGT). $\forall \sigma \in \mathrm{Tr}(B^{\bot})$:

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

Theorem 1 distinguishes two cases:

- When the last state of a system is a global state ($\mathrm{last}(\sigma) \in Q$), none of the components is 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 s.t., 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.



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

Fig. 7: Instrumented atomic components of system Task

5 Model Transformation

5.1 Instrumentation of Atomic Components

Given an atomic component with partial-state semantics as per Definition 4, we instrument this atomic component s.t. it is able to transfer its state through port β, each time the component moves out from a busy location.

Definition 7 (Instrumenting an atomic component). *Given an atomic component in partial-state semantics $B^\perp = (P \cup \{\beta\}, L \cup L^\perp, T^\perp, X)$ with initial location $l_0 \in L$, we define a new component $B^r = (P^r, L \cup L^\perp, T^r, X^r)$ where:*

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

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

Example 5 (Instrumenting an atomic component). Figure 7 shows the instrumented version of atomic components in system Task (depicted in Fig. 4).

5.2 Creating a New Atomic Component to Reconstruct Global States

Let us consider $B^\perp = \gamma^\perp(B^\perp_1, \ldots, B^\perp_n)$ with partial-state semantics, s.t.:

- 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 $B^r_1, \ldots, B^r_n$ have been obtained through Definition 7 s.t. B^r_i is the instrumented version of $B^\perp_i$.

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 6).

Definition 8 (RGT atom). *Component* RGT *is defined as* (P, L, T, X) *where:*

- $X = \bigcup_{i\in[1,n]}\{B^r_i.X^r\} \bigcup_{i\in[1,n]}\{B^r_i.X^r_c\} \cup \{gs_a \mid a \in \gamma\} \cup \{(z_1, \ldots, z_n)\} \cup \{V, v, m\}$*, where* $B^r_i.X^r_c$ *is a set containing a copy of the variables in* $B^r_i.X^r$.

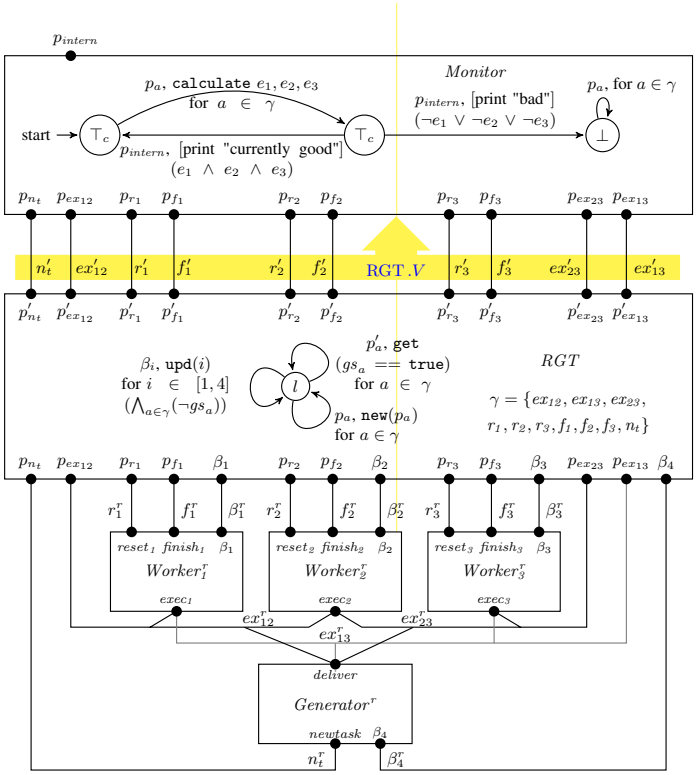


Fig. 8: Monitored version of system Task

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

For space reasons, we only overview the description of atom RGT and do not provide the internal algorithms. Full and formal details can be found in [13]. A global state is encoded 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 7, 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, stores them in variable V and delivers them through the dedicated ports.

Example 6 (RGT atom). Figure 8 depicts the component RGT for system Task. For space reasons, only one instance of each type of transitions is shown. At runtime, RGT produces the sequence of global states in the right-most column of Table 1.

5.3 Connections

After building component RGT (see Definition 8), and instrumenting atomic components (see Definition 7), 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 $B^r = (P \cup \{\beta^r\}, L \cup L^\perp, T^r, X^r)$ s.t. $B^r = B_i^r \in \{B_1^r, \cdots, B_n^r\}$, we define a new composite component.

Definition 9 (Composite component transformation). *For a composite component $B^\perp = \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$, we introduce a corresponding transformed component $B^r = \gamma^r(B_1^r, \ldots, B_n^r, RGT)$ s.t. $\gamma^r = a^r_{\gamma^\perp} \cup a^m$ where:*

- *$a^r_{\gamma^\perp} = \{a^r \mid a \in \gamma^\perp\}$ is the set of transformed interactions with:*

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

- *a^m is a set of new interactions s.t. $a^m = \{a' \mid a \in \gamma\}$, where $\forall a \in \gamma, a' = \{RGT.p'_a\}$ is the 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 s.t. 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 set a^m is the set of all unary interactions a' associated to each existing interaction $a \in \gamma$ in the system.

Example 7 (Transformed composite component). Figure 8 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, component RGT (cf. Definition 8) implements function RGT (cf. Definition 6). 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: the total order between executed interactions is preserved.

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

For each trace in partial-state semantics, component RGT produces the witness trace in the initial model, as stated by the following theorem.

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

Connecting a monitor. Using [9], one can monitor a system with partial-state semantics with the previous transformations by plugging a monitor 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.

Example 8 (Monitoring system Task). Figure 8 depicts the transformed system Task with a monitor (for the homogeneous distribution of the tasks among the workers) where e_1, e_2, and e_3 are events related to the pairwise comparison of the number of executed tasks by $Worker$s. For $i \in [1,3]$, event e_i evaluates to true whenever $|x_{i \bmod 3+1} - x_{i \bmod 3}|$ is lower than 3 (for this example). Component $Monitor$ evaluates $(e_1 \wedge e_2 \wedge e_3)$ upon the reception of a new global state from RGT and emits the associated verdict till reaching bad state $\perp$. The global trace $(free, free, free, hold) \cdot ex_{12} \cdot (done, done, free, delivered) \cdot n_t$ (see Table 1) is sent by component RGT to the monitor which in turn produces the sequence of verdicts $\top_c \cdot \top_c$ (where $\top_c$ is verdict currently good, see [3,8]).

6 Implementation and Performance Evaluation

We present some case studies on executable BIP systems conducted with RVMT-BIP, a tool integrated in the BIP tool suite [2].

Case Study 1: Demosaicing Demosaicing is an algorithm for digital image processing used to reconstruct a full color image from the incomplete color samples output from an image sensor. The model contains *ca.* 1,000 lines of code, consists of 26 atomic components interacting through 35 interactions. We consider two specifications related to process completion: i) Internal demosaicing units should finish their process before post-demosaicing starts processing (φ_1). ii) Internal demosaicing units should not start demosaicing process before pre-demosaicing finishes its process (φ_2).

Case Study 2: Task Management We consider our running example system Task and a specification of the homogeneous distribution of the tasks among the workers (φ_3).

Evaluation Principles For each system, and all its properties, we synthesize a BIP monitor following [9] 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 forces them to run sequentially. Varying the number of threads allows us to assess the performance of the (monitored) system under different degrees of parallelism. In particular, we expected the induced overhead to be insensitive to the degree of parallelism. For instance, an undesirable behavior would have been to observe a performance degradation (and an overhead increase) which would mean either that the monitor sequentializes the execution or that the monitoring infrastructure is not suitable for multi-

Table 2: Results of monitoring Demosaicing and Task 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
		10	29.15		φ_2	2,200	1,599	1	69.116	1.67
								3	41.235	1.51
								10	29.251	0.31
Task (100,000 tasks)	399,999	1	117.96	Task distribution	φ_3	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_3	177,611	1	126.11	6.91
		2	72.32				2	105.66	46.1

threaded systems. We also extensively tested the functional correctness of RVMT-BIP, that is whether the verdicts of the monitors are sound and complete.

Results (*cf.* Table 2) Each time measurement is an average value obtained after 100 executions. Column *# interactions* shows the number of functional steps of system. Columns *no monitor* reports the execution time of the systems without monitors when varying the number of threads. Columns *with monitor* reports 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). 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 compared it to RV-BIP [9]. Table 3 shows the results of a performance evaluation of monitoring Demosaicing and Task. 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 [9].

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.

Runtime verification of single-threaded CBSs. In [9], we proposed a first approach for the runtime verification of CBSs. The approach takes a CBS and a regular property as input and generates a monitor implemented as a component which is then integrated within an existing CBS. At runtime, the monitor consumes the global trace (i.e., sequence of global states) and yields verdicts regarding property satisfaction. The technique in [9] 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 [9] when dealing with multi-threaded CBSs: a monitor as synthesized in [9] can be plugged to component RGT which is reconstructing the global states of the system.

Decentralized runtime verification. The approaches in [4,7] 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/satisfactions, local monitors need to communicate, because their trace are only partial w.r.t. the global behavior of the system.

Monitoring safety properties in concurrent systems. The approach in [16] 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 [16] 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 [16] 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.

Parallel runtime verification of monolithic sequential programs. Berkovich et al. [5] introduce parallel algorithms to speed up the runtime verification of sequential programs against complex LTL formulas using a graphics processing unit (GPU). Monitoring threads directly execute on the GPU. The approach in [5] is not tailored to CBSs and is a complementary technique that adds significant computing power to the system to handle the monitoring overhead. Note that, as shown by our experiments, our approach preserves the performance of the monitored system. Finally, our approach is not bound to any particular logic, and allows for Turing-complete monitors.

8 Future Work

A first direction is to consider monitoring for fully decentralized and 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 [6]. A lot of work has been done in order to monitor properties on a distributed (monolithic) systems; e.g., [15] for online monitoring of CTL properties, [11] for online monitoring of LTL properties, [14] for offline monitoring of properties expressed in a variant of CTL, and [17] 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].

Acknowledgement. The authors are supported by the Artemis ARROWHEAD project under grant agreement number 332987 (ARTEMIS/ECSEL Joint Undertaking, supported by the European Commission and French Public Authorities). The work reported in this article is in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

References

1. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: FORTE 2008. pp. 116–133 (2008)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM 2006. pp. 3–12 (2006)
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
4. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: FM 2012. pp. 85–100 (2012)
5. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: GPU-based runtime verification. In: IPDPS 2013. pp. 1025–1036 (2013)
6. 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)
7. Falcone, Y., Cornebize, T., Fernandez, J.: Efficient and generalized decentralized monitoring of regular languages. In: FORTE 2014. pp. 66–83 (2014)
8. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? STTT 14(3), 349–382 (2012)
9. 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)
10. Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989)
11. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: IPDPS 2015. pp. 494–503 (2015)
12. Nazarpour, H.: Runtime Verification of Multi-Threaded BIP. `http://www-verimag.imag.fr/~nazarpou/rvmt.html`
13. Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., Combaz, J.: Monitoring multi-threaded component-based systems. Tech. Rep. TR-2015-5, Verimag Research Report (2015), available at `http://www-verimag.imag.fr/TR/TR-2015-5.pdf`
14. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: OPODIS 2003. pp. 171–183 (2003)
15. Sen, A., Garg, V.K.: Formal verification of simulation traces using computation slicing. IEEE Trans. Computers 56(4), 511–527 (2007)
16. 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)
17. 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)