# RVMT-BIP: A Tool for the Runtime Verification of Multi-Threaded Component-Based Systems

*Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga*

**Verimag Research Report n$^o$ TR-2016-2**

May 13, 2016

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

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Équation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

# RVMT-BIP: A Tool for the Runtime Verification of Multi-Threaded Component-Based Systems

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

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

May 13, 2016

**Abstract**

RVMT-BIP is a tool for the runtime verification of multi-threaded component-based systems described in the Behavior, Interaction, Priority (BIP) framework against linear-time and logic-independent properties. RVMT-BIP implements rigorous semantics-preserving transformations of BIP systems to instrument and inject a sound and concurrency-preserving global-state reconstructor for on-the-fly monitoring. Our experiments on several multi-threaded BIP systems show that RVMT-BIP generally induces a cheap runtime overhead, especially when the system is highly concurrent.

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

**Reviewers:**

**How to cite this report:**

@techreport {TR-2016-2,
title = {RVMT-BIP: A Tool for the Runtime Verification
of Multi-Threaded Component-Based Systems},
author = {Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, Marius Bozga},
institution = {{Verimag} Research Report},
number = {TR-2016-2},
year = {2015}
}

# 1 Introduction

We address the problem of verifying multi-threaded component-based systems (CBSs) defined in the Behavior Interaction Priority (BIP) framework against linear-time properties expressing their desired behavior. BIP is a powerful and expressive component framework for the formal construction of heterogeneous systems. In BIP, a formal operational semantics defines the coordination of atomic components with interactions (“glue" code). We consider properties referring to the global states of the system, which, in particular, implies that properties can not be “projected" and checked on individual components. Generally, it is not possible to ensure or verify such properties using static verification techniques, either because of the state-explosion problem or because they can only be decided with runtime information or user interaction.

We provide a tool-supported complementary verification technique for CBSs using runtime verification. In this context, the challenge that arises is that, because of the multi-threaded execution of components, a global-state of the system may never be available at runtime because of at least one executing component. Hence, a viable monitoring solution has to be able to decide global-state properties from the partial-state information, while preserving the concurrency of systems and the performance gained from multi-threading. In [17], we formally present our solution to this problem which consists in synthesizing a sound and concurrency-preserving global-state reconstructor that can feed a runtime monitor with global-state information. The theoretical results and the properties of this approach are presented in full in [17] and are briefly recalled in Section 3. The subject of this paper is to report on RVMT-BIP, a full implementation and evaluation of [17]. RVMT-BIP is available for download at [15].

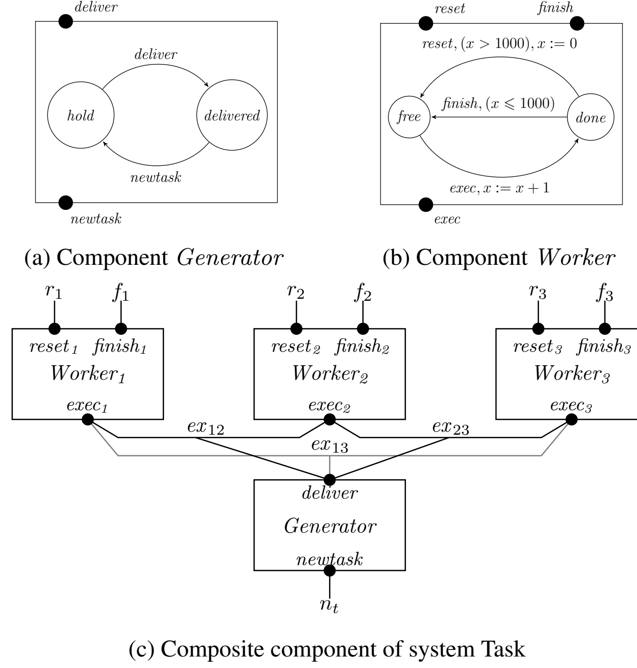
**Outline.** This paper is organized as follows. Section 2 briefly presents the basic semantics of BIP. Section 3 overviews the approach for the monitoring of multi-threaded CBSs. Section 4 presents the architecture of RVMT-BIP. Section 5 presents (i) the CBSs and properties of our case studies, (ii) the experimental results and a discussion on the performance of RVMT-BIP when monitoring properties over these CBSs and (iii) the evaluation of the functional correctness of RVMT-BIP.

# 2 Overview of Behavior Interaction Priority (BIP)

We provide a succinct description of the BIP framework and refer to [2, 16] for the detailed and fully-formalized operational semantics.

BIP (Behavior, Interaction, Priority) is a powerful and expressive framework for the formal construction of heterogeneous systems [3]. BIP supports a construction methodology of components as the superposition of three layers: *behavior*, *interaction*, and *priority*. Layering favors a clear separation between behavior and structure. The behavior layer describes the operational semantics of atomic components. Atomic components are transition systems endowed with a set of local variables and a set of *ports* labeling individual transitions. Ports are used for synchronization and communication with other components. Transitions can be guarded by some constraints over local variables. Local variables of an atomic component can be sent or modified through the interacting ports. The interaction layer defines a set of connectors over the ports of atomic components describing the synchronizations (so-called interactions) between atomic components. An interaction is a synchronous action among (some of) the components which have one of their ports involved in the interaction. The priority layer describes scheduling policies for interactions. Composite components are built from connected atomic components along with a set of priority rules.

***Example 1 (Atomic component and composition)*** *Figure 1 depicts a task system, called Task, 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. Figures 1a and 1b show the atomic components of system Task.* $\mathit{Generator}$ *delivers a new task to the workers through the port* $\mathit{deliver}$ *and is received by the workers through the port* $\mathit{exec}$*. Reception of a new task by each worker causes a move from the location free to the location* $\mathit{done}$ *and the local variable* $x$ *(i.e., number of task fulfillment) is incremented. Whenever a worker complete a assigned task, the worker is re-initialized by moving back to the location free. If the value of* $x$ *is less than 1000, re-initialization is done by the transition labeled by the port finish , otherwise the transition labeled by the port* $\mathit{reset}$ *re-initialized the worker by resetting the value of* $x$ *to zero. Figure 1c depicts a composite component of system Task.*




(a) Component *Generator* (b) Component *Worker*

(c) Composite component of system Task

Figure 1: Definition of atomic components and their composition of System Task in BIP

# 3 Monitoring Multi-Threaded Component-Based Systems [17, 16]

Figure 2 overviews our approach. Recall that according to [2], a BIP system with global-state semantics $S_g$ (sequential model), is (weakly) bisimilar to the corresponding partial-state model $S_p$ (concurrent model). Moreover, $S_p$ generally runs faster than $S_g$ because of its parallelism. Thus, if a trace of $S_g$ satisfies $\varphi$, then the corresponding trace of $S_p$ satisfies $\varphi$ as well.

We overview our concurrency-preserving approach to monitoring $S_p$ using on-the-fly global-state reconstruction [17] (described in full in [16]). We define transformations 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 $\varphi$. The witness trace is built by a global-state reconstructor which accumulates the local (partial) states of the components to produce a consistent global trace.

More concretely, our approach proceeds as follows: from a linear-time property $\varphi$ and an input composite system $\gamma\,(B_1, \ldots, B_n)$ where the $B_i$'s ($i \in [1, n]$) are atomic components and $\gamma$ is a set of interactions defining the composition of components:

1. we synthesize a global-state reconstructor which reconstructs on-the-fly a global trace (component $\mathrm{RGT}$ - Reconstructor of Global Trace),
2. we synthesize a component monitor as a component $\mathrm{Monitor}$,
3. we instrument the components $B_i$, $i \in [1, n]$, so that their local states can be observed and they can be connected to $\mathrm{RGT}$ with a new set of interactions $\gamma'$,
4. we build a new composite system $\gamma'\,(B'_1, \ldots, B'_n, \mathrm{RGT}, \mathrm{Monitor})$ where $B'_i$, $i \in [1, n]$, are the instrumented components, and $\varphi$ is checked at runtime.

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 (see Theorem 1 in [17]). Moreover, our method is correct in that the global-state reconstructor synthesized as a BIP component always produces the correct witness trace (see Theorem 2 in [17]).

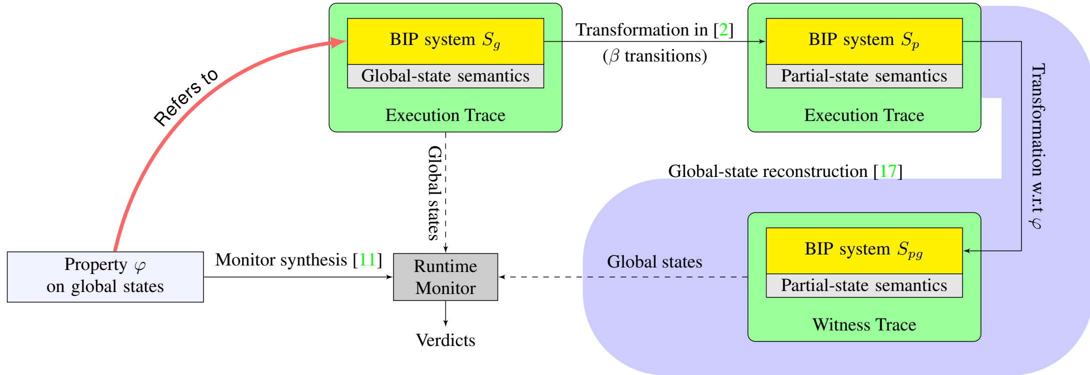



Figure 2: Approach overview

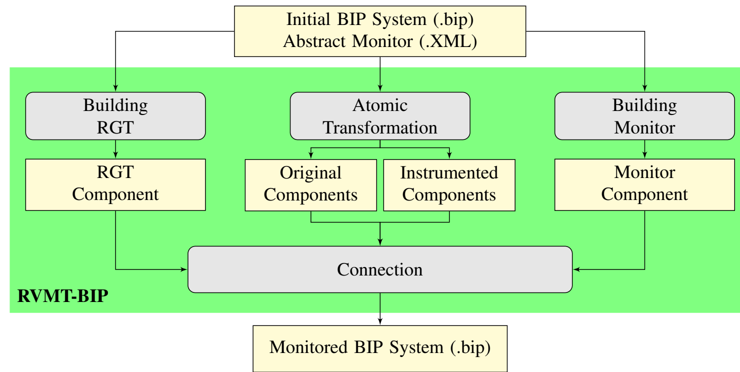



Figure 3: Overview of RVMT-BIP work-flow

# 4 Architecture of RVMT-BIP

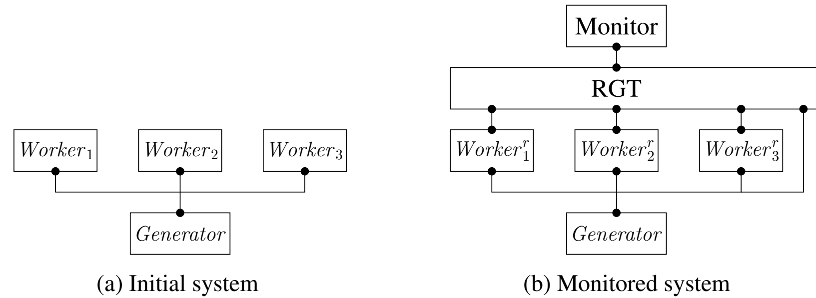
RVMT-BIP (Runtime Verification of Multi-Threaded BIP) is a Java implementation of ca. 2,200 LOC and is part of BIP distribution. RVMT-BIP takes as input a BIP CBS and a monitor description for a property, and outputs a new BIP system whose behavior is monitored against the property while running concurrently. RVMT-BIP uses the following modules (see Fig. 3):

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

- Module *Building* RGT takes as input the initial BIP system and a monitor description and produces component RGT (Reconstructor of Global Trace) which reconstructs and accumulates global states at runtime to produce the global trace.

- Module *Building* Monitor takes as input the initial BIP system and a monitor description and then outputs the atomic component implementing the monitor (following [11]). Component Monitor receives and consumes the reconstructed global trace generated by component RGT at runtime and emits verdicts.

- Module *Connections* constructs the new composite and monitored component. The module takes as input the output of the *Atomic Transformation*, *Building* RGT and *Building* Monitor modules




(a) Initial system

(b) Monitored system

Figure 4: Illustration of the transformation of RVMT-BIP

and then outputs a new composite component with new connections. The new connections are purposed to synchronize instrumented components and component RGT in order to transfer updated states of the components to RGT. Each instrumented component interacts with component RGT independently.

***Example 2 (RVMT-BIP transformation)*** *Figure 4 depicts how RVMT-BIP transforms system Task depicted in Fig. 4a. Recall that system Task consists of four atomic components $Generator$, $Worker_1$, $Worker_2$ and $Worker_3$ and a set of interactions to deliver generated tasks by the $Generator$ to the $Workers$. The transformation is performed for a property stating the homogeneous distribution of tasks among the workers which is related to the state of $Workers$. The monitored system obtained after transformation (Fig. 4b) consists of component $Generator$ (left intact), $Worker_1^r$, $Worker_2^r$ and $Worker_3^r$ (instrumented version of $Worker_1$, $Worker_2$ and $Worker_3$ respectively), components* RGT *and* Monitor, *additional interactions (for sending updated states of instrumented components to component* RGT *and for sending reconstructed global states to component* Monitor*), and modified interactions to notify component* RGT *about the execution of interactions.*

# 5 Evaluation

We report on our evaluation of RVMT-BIP with some case studies on executable BIP systems. Executing these systems with a multi-threaded controller results in a faster run since the systems benefit from the parallel threads. These systems can also execute with a single-threaded controller which forces them to run sequentially.

## 5.1 Case Studies

We present the case studies used in our evaluation.

### 5.1.1 Process Completion of Demosaicing

**Description of Demosaicing.** Demosaicing is an algorithm[1] for digital image processing used to reconstruct a full color image from the incomplete color samples output from an image sensor. Demosaicing works on $5 \times 5$ 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 5 shows a simplified version of the processing network of Demosaicing. Demosaicing contains a $Splitter$ and a $Joiner$ process, a pre-demosaicing ($Demopre$) and a post-demosaicing ($Demopost$) processes and three internal demosaicing $Demo$ processes that run in parallel. The real model used in our experiments contains ca. 1,000 lines of code, consists of 26 atomic components interacting through 35 interactions.

[1] Demosaicing has been used in [21] for implementing multi-threaded timed CBSs.

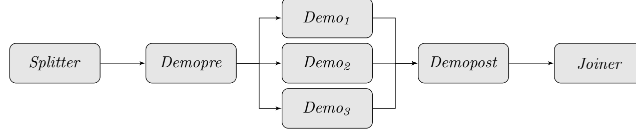
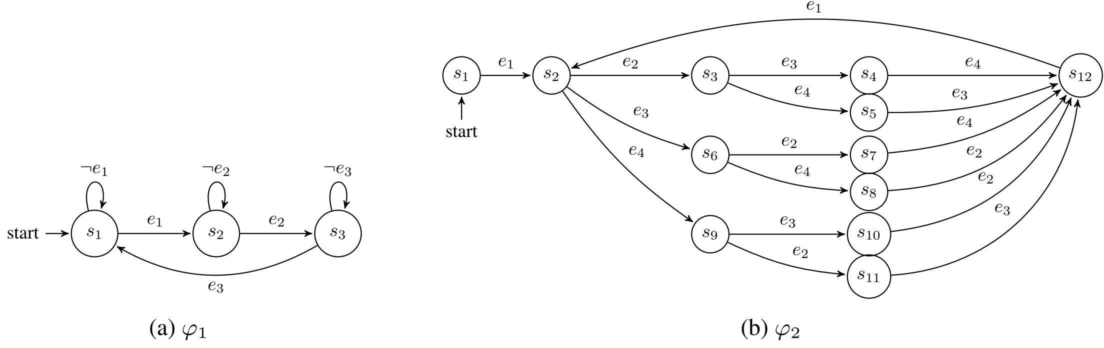



Figure 5: Processing network of system Demosaicing




(a) $\varphi_1$

(b) $\varphi_2$

Figure 6: Automata of the properties of system Demosaicing

**Specifying process completion.** We consider two properties of Demosaicing related to process completion.

1. It is necessary that first $Splitter$ receives the image then transmits it to the internal demosaicing units. After the demosacing process is completed $Joiner$ must receive the processes image. $Splitter$ transmits the output through port $transmit$ and $Joiner$ receives the input through port $get_img$. We use variable $port$ to record the last executed port. Each demosaicing unit has a boolean variable $done$ which is set to `true` whenever the demosaicing process completes. This requirement is formalized as property $\varphi_1$ defined by the automaton depicted in Fig. 6a where the events are $e_1 : Splitter.port == transmit$, $e_2 : (Demo_1.done \wedge Demo_2.done \wedge Demo_3.done)$ and $e_3 : Joiner.port == get_img$. From the initial state $s_1$, the automaton moves to state $s_2$ when $Splitter$ finishes its image transmission. From state $s_2$, the automaton moves to state $s_3$ when all the internal demosaicing units finish their process. The reception of the processed images by $Joiner$ causes a move from state $s_3$ to $s_1$.

2. Moreover, the internal demosaicing units ($Demo_1$, $Demo_2$, $Demo_3$) should not start the demosaicing process until the pre-demosaicing unit finishes its process. The pre-demosaicing unit sends its output to the internal demosaicing units through port $transmit$ and each internal demosaicing unit starts the demosaicing process by executing a transition labeled by port $start$. This requirement is formalized as property $\varphi_2$ which is defined by the automaton depicted in Fig. 6b 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 $s_2$. The internal demosaicing units can start in any order. Moreover, all demosaicing units must eventually start their internal process and if so 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.

#### 5.1.2 Reader-Writer 1

**Description of Reader-Writer 1.** The system Reader-Writer 1 consists of a set of independent composite components. Each composite component (Fig. 7) involves four components: $Reader$, $Writer$, $Clock$, and

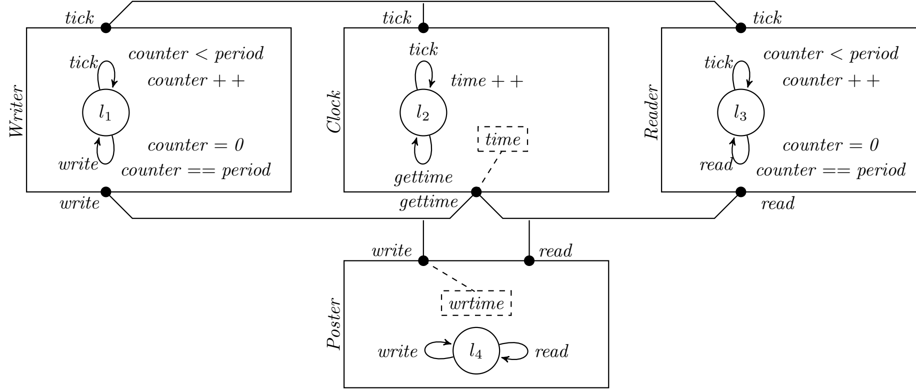



Figure 7: Model of one of the instances of the composite component of system Reader-Writer 1

$Poster$. $Reader$ and $Writer$ communicate with each other through $Poster$. The data generated by each writer is written to $Poster$ and can then be accessed by $Reader$.

**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 the moments of reading and writing actions respectively, then the difference between $t_2$ and $t_1$ must be less than a specific duration $\delta$, 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 $\varphi_3$ which is defined by the automaton depicted in Fig. 8a, 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 $s_1$ to $s_2$. When $Reader$ reads from $Poster$, the automaton moves from $s_2$ to $s_1$. $Reader$ is allowed to read from $Poster$ 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 $Poster$. $\varphi_3$ also allows to read from $Poster$ 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 $Poster$.

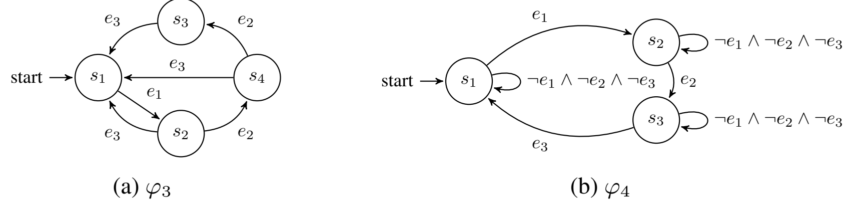
### 5.1.3 Reader-Writer 2

**Description of Reader-Writer 2.** System Reader-Writer 2 is a more complex version of Reader-Writer 1 and involves several writers. This system has six components: $Reader$, $Writer_1$, $Writer_2$, $Writer_3$, $Clock$ and $Poster$. The $Writer$s are synchronized together. $Reader$ and $Writer$s communicate with each other through $Poster$. The data generated by each writer is written to $Poster$ and can then be accessed by $Reader$.

**Specification of the execution order.** The 3 writers should periodically write data to a poster in a specific order. 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, $Writer_3$ can write only when $Writer_2$ finishes writing, and the same goes on for the next periods. 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 $\varphi_4$ defined by the automaton depicted in Fig. 8b where:

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




(a) $\varphi_3$ (b) $\varphi_4$

Figure 8: Automata of the properties of system Reader-Writer

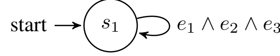



Figure 9: Automaton of the property of system Task

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

#### 5.1.4 Task Management

We consider the system Task which is described in Example 1 with 100,000 generated tasks by $Generator$.

**Specification of the task distribution.** A desirable property of system Task is the homogeneous distribution of the tasks among the workers which depends on the execution time of each worker. Different tasks may have different execution times for different workers. Obviously, the faster a worker completes each task, the higher is the number of its accomplished tasks. After executing a task, the value of the variable $x$ of a worker is increased by one. Moreover, the absolute difference between the values of variable $x$ of any two workers must always be less than a specific integer value (which is 300 for this case study). This requirement is formalized as property $\varphi_5$ which is defined by the automaton depicted in Fig. 9 where $e_1 : |worker_1.x - worker_2.x| < 300$ , $e_2 : |worker_2.x - worker_3.x| < 300$ and $e_3 : |worker_1.x - worker_3.x| < 300$. The property holds as long as $e_1$, $e_2$ and $e_3$ hold.

### 5.2 Evaluating Functional Correctness

We evaluated the functional correctness of RVMT-BIP, that is whether the synthesized global-state reconstructors and monitors produce sound and complete verdicts. Each of the systems of our case study is correct by design, and we run monitored versions of these for several hours without any error reported. To assess the error detection, we built mutated versions of the systems whose behaviors eventually lead to a violation of the properties[2]. We built one mutant per pair of system and property. Our monitors were able to detect and kill all the mutants. 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.

### 5.3 Evaluating Performance

We report on the performance of the global-state reconstructors and monitors synthesized with RVMT-BIP. All the experiments are conducted on 64-bit Debian Linux 8 using an Intel processor E5-2630 v3 (20M Cache, 2.40 GHz) Core i7 and 4GB main memory.

[2] We do not report on the performance on monitoring mutated versions of the systems as the occurrence time of the error was non-deterministic.

Table 1: Results of monitoring with RVMT-BIP

| system | # executed interactions | execution time and overhead according to the number of threads | | | | | | | | | | # events | # extra executed interactions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | |
| Demosaicing (26,35) | 1,300 | 18.98 | 10.24 | 7.75 | 6.85 | 6.58 | 6.09 | 6.33 | 6.45 | 6.29 | 6.27 | n/a | n/a |
| *Demosaicing (27,69)* | 3,051 | 19.02 | 11.53 | 8.17 | 7.43 | 6.68 | 6.50 | 6.27 | 6.05 | 6.03 | 6.18 | 1,300 | 1,751 |
| $\varphi_1$ *(11)* | | 0.1% | 12.6% | 5.4% | 8.5% | 4.3% | 6.6% | < 0.1% | < 0.1% | < 0.1% | < 0.1% | | |
| *Demosaicing (27,46)* | 1,850 | 18.68 | 11.05 | 7.65 | 7.70 | 6.77 | 6.38 | 6.22 | 6.45 | 6.17 | 6.35 | 400 | 550 |
| $\varphi_2$ *(4)* | | < 0.1% | 7.9% | < 0.1% | 12.4% | 2.8% | 4.8% | < 0.1% | < 0.1% | < 0.1% | < 0.1% | | |
| ReaderWriter 1 (12,9) | 120,000 | 61.48 | 29.67 | 20.03 | 20.00 | 20.05 | 20.21 | 20.60 | 21.54 | 21.92 | 22.13 | n/a | n/a |
| *ReaderWriter 1 (13,12)* | 200,000 | 62.53 | 38.29 | 21.96 | 22.28 | 22.62 | 22.71 | 22.88 | 23.48 | 24.15 | 24.47 | 40,000 | 80,000 |
| $\varphi_3$ *(3)* | | 1.6% | 27.7% | 9.6% | 11.4% | 12.8% | 12.4% | 11.0% | 9.0% | 10.1% | 10.5% | | |
| ReaderWriter 2 (6,7) | 20,000 | 32.06 | 21.45 | 12.04 | 11.37 | 11.33 | 11.37 | 11.44 | 11.49 | 11.53 | 11.58 | n/a | n/a |
| *ReaderWriter 2 (7,12)* | 85,000 | 33.92 | 22.72 | 13.90 | 13.77 | 14.09 | 14.36 | 14.83 | 15.18 | 15.41 | 15.57 | 20,000 | 65,000 |
| $\varphi_4$ *(5)* | | 5.8% | 5.9% | 15.4% | 21.1% | 24.3% | 26.2% | 29.6% | 32.1% | 33.5% | 34.4% | | |
| Task (4,10) | 399,999 | 117.28 | 70.18 | 60.91 | 60.06 | 58.98 | 60.01 | 60.93 | 61.77 | 63.13 | 65.45 | n/a | n/a |
| *Task (5,16)* | 600,197 | 123.98 | 71.73 | 62.28 | 63.26 | 62.79 | 62.78 | 63.35 | 64.57 | 65.61 | 66.27 | 100,198 | 200,198 |
| $\varphi_5$ *(3)* | | 5.7% | 2.2% | 2.2% | 5.3% | 6.4% | 4.4% | 3.9% | 4.5% | 3.9% | 1.2% | | |

#### 5.3.1 Evaluation Principles

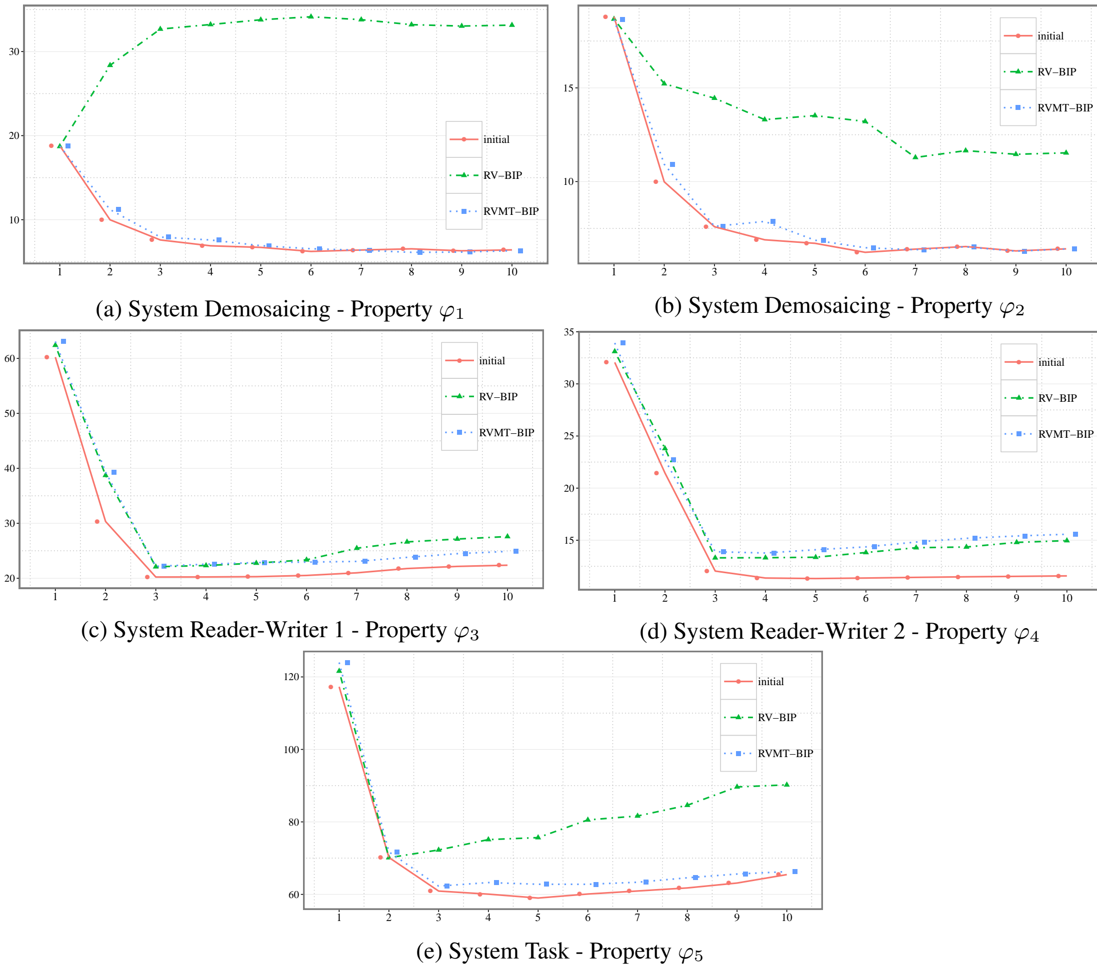
Following the work-flow depicted presented in Section 4 (see Fig. 3), for each system, and all its properties, we synthesize two monitored versions, one with RVMT-BIP with an asynchronous global-state reconstructor and a monitor, and one with RV-BIP [11] with only a monitor. We run each system by varying the 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. We note that additional steps are introduced in the concurrent transitions of the system. 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. More precisely, the research questions addressed by our experiments are:

1. What is the performance of monitoring and is it insensitive to the degree of parallelism?
2. What kind of systems/properties the tool can handle efficiently?

#### 5.3.2 Results

Figure 10 and Table 1 report the timings obtained when checking specifications *complete process* property on Demosaicing, *data freshness* and *execution ordering* property on Reader-Writer systems and *task distribution* property on Task. Each measurement is an average value obtained over 100 executions of these systems. In Table 1, the columns have the following meanings:

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




(a) System Demosaicing - Property $\varphi_1$

(b) System Demosaicing - Property $\varphi_2$

(c) System Reader-Writer 1 - Property $\varphi_3$

(d) System Reader-Writer 2 - Property $\varphi_4$

(e) System Task - Property $\varphi_5$

Figure 10: Execution time according to the number of threads

- Column *extra executed interactions* reports the number of additional interactions (i.e., execution of interactions which are added into the initial system for monitoring purposes).

**On the performance of RVMT-BIP (see the results in Table 1 visualised in Figure 10).** For the larger system Demosaicing (in terms of components), we can observe that the observed overhead is 12.6% in the worst case with 2 threads and a reasonable overhead in other cases. This indicates a good general performance of the generated monitors. Moreover, the overhead is insensitive to the number of threads used to execute Demosaicing. The trend in the execution time of the monitored system follows the trend observed for the initial system; and in particular the peak performance is obtained in both cases with 8 threads.

**RV-BIP vs. RVMT-BIP.** To illustrate the advantages of monitoring multi-threaded systems with RVMT-BIP, we compared the performance of RVMT-BIP and RV-BIP ([11]); see Table 2 for the results. Monitoring with RV-BIP amounts to use a standard runtime verification technique, i.e., not tailored to multi-threaded systems. At runtime, the RV-BIP monitor consumes the global trace (i.e., sequence of global states) of the system (global snapshots are obtained by synchronization among the components) and yields verdicts regarding property satisfaction which is aimed to efficiently handle CBSs with sequential executions.

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

1. Fixing a system and a property, the number of events received by the monitors of RV-BIP and RVMT-BIP are similar, because both techniques produce monitored system that are observationally equivalent to the initial [11, 17]. Moreover, increasing the number of threads does not change the global behavior of the system, therefore the number of events is not affected by the number of threads.

2. Fixing a system and a property, the number of extra interactions imposed by RVMT-BIP is greater than the one imposed by RV-BIP. In the monitored system obtained with RVMT-BIP, after the execution of an interaction, the components that are involved in the interaction and concerned in property independently send their updated state to component $\mathrm{RGT}$ (whenever their internal computation is finished). In the monitored system obtained with RV-BIP, after the execution of an interaction influencing the truth value of the property, all the updated states will be sent at once (synchronously) to the component monitor. Hence, the evaluation of an event in RV-BIP is done in one step and the number of extra interactions imposed by RV-BIP is the same as the number of monitored events (Table 2).

3. In spite of the higher number of extra interactions imposed by RVMT-BIP, during a multi-threaded execution, the fewer synchronous interactions of monitored components imposed by RV-BIP induces a significant overhead. This phenomenon is especially visible for the two most concurrent systems: Demosaicing and Task (see Figs. 10a, 10b and 10e).

4. *On the independence of components*: Consider systems Demosaicing and Task, which consist of independent components with low-level synchronization and high degree of parallelism, and for which the monitored property requires the states of these independent components. On the one hand, at runtime, RV-BIP imposes synchronization among the components concerned with property and the component monitor. It results in a loss of the performance when executing with multiple threads. On the other hand, RVMT-BIP collects updated states of the components independently right after their state update. Consequently, with RVMT-BIP, the system performance in multi-threaded setting is preserved (see Figs. 10a, 10b and 10e) as a negligible overhead is observed. This is a usual and complex problem which depends on many factors such as platform, model, external codes, compiler, etc. This renders the computation of the number of threads leading to peak performance complex.

5. *Synchronization of independent components*: In RV-BIP, the thread synchronizations and the synchronization of components with the monitor induce a huge overhead especially when concurrent component are concerned with the desired property (Fig. 10a).

6. *Synchronized components*: We observe that, for system ReaderWriter 2, the overhead obtained with RVMT-BIP monitor is slightly higher than the one obtained with RV-BIP monitors (see Fig. 10d). Indeed, system ReaderWriter 2 consists of 3 writers synchronized by a clock component. Moreover, property $\varphi_4$ is defined over the states of all the writers. As a matter of fact, if one of the writer needs to communicate with component $\mathrm{RGT}$, then all the other writers need to wait until the communication ends. That is, when the concurrency of the monitored system is limited by internal synchronizations, RVMT-BIP becomes less interesting to use as a monitoring solution.

7. *Synchronized components in independent composite components*: If the initial system (i) consists of set of independent composite components working independently and concurrently, (ii) the components in each composite are highly synchronized (low degree of parallelism in each composition) and (iii) the desired property is defined over the states of the components of a specific composite component, then RVMT-BIP performs similarly to RV-BIP. Indeed, in the monitored system the independent entities (i.e., composite component) are able to run as concurrently as in the initial system and the overhead is caused by the synchronized components. However, by increasing the number of threads, RVMT-BIP monitors offers better performance (Fig. 10c).

8. *On the number of threads*: We note that increasing the number of threads may lead after some point to a gradual performance reduction (both in the initial and monitored systems), see Figs. 10e and 10c.

Table 2: Results of monitoring with RV-BIP

| system | # executed interactions | execution time and overhead w.r.t. different number of threads | | | | | | | | | | # events | # extra executed interactions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | |
| Demosaicing (26,35) | 1,300 | 18.98 | 10.24 | 7.75 | 6.85 | 6.58 | 6.09 | 6.33 | 6.45 | 6.29 | 6.27 | n/a | n/a |
| *Demosaicing (27,37)* | 2,450 | 19.66 | 27.34 | 32.28 | 32.61 | 33.03 | 32.23 | 31.17 | 31.24 | 31.22 | 31.81 | 1,300 | 1,300 |
| $\varphi_1$ *(11)* | | 3.5% | 167% | 316% | 376% | 402% | 429% | 392% | 384% | 369% | 407% | | |
| *Demosaicing (27,37)* | 1,700 | 19.50 | 14.79 | 13.87 | 13.11 | 13.13 | 12.75 | 11.18 | 11.34 | 11.19 | 11.16 | 400 | 400 |
| $\varphi_2$ *(11)* | | 2.7% | 44.4% | 78.8% | 91.4% | 99.7% | 109% | 76.5% | 75.7% | 78.0% | 78.0% | | |
| ReaderWriter 1 (12,9) | 120,000 | 61.48 | 29.67 | 20.03 | 20.00 | 20.05 | 20.21 | 20.60 | 21.54 | 21.92 | 22.13 | n/a | n/a |
| *ReaderWriter 1 (13,11)* | 1600,000 | 61.97 | 37.77 | 21.94 | 22.13 | 22.62 | 23.14 | 25.09 | 26.21 | 26.73 | 27.18 | 40,000 | 40,000 |
| $\varphi_3$ *(3)* | | 0.8% | 26.0% | 9.5% | 10.6% | 12.8% | 14.5% | 21.8% | 21.7% | 21.9% | 22.7% | | |
| ReaderWriter 2 (6,7) | 20,000 | 32.06 | 21.45 | 12.04 | 11.37 | 11.33 | 11.37 | 11.44 | 11.49 | 11.53 | 11.58 | n/a | n/a |
| *ReaderWriter 2 (7,9)* | 40,000 | 33.11 | 23.80 | 13.31 | 13.32 | 13.37 | 13.82 | 14.28 | 14.35 | 14.79 | 14.96 | 20,000 | 20,000 |
| $\varphi_4$ *(5)* | | 3.2% | 10.9% | 10.5% | 17.1% | 18.0% | 21.5% | 24.8% | 24.8% | 28.2% | 29.2% | | |
| Task (4,10) | 399,999 | 117.28 | 70.18 | 60.91 | 60.06 | 58.98 | 60.01 | 60.93 | 61.77 | 63.13 | 65.45 | n/a | n/a |
| *Task (5,12)* | 500,197 | 121.61 | 70.12 | 72.25 | 75.11 | 75.66 | 80.54 | 81.62 | 84.58 | 89.65 | 90.21 | 100,198 | 100,198 |
| $\varphi_5$ *(3)* | | 3.6% | $<$ 0.1% | 18.6% | 25.0% | 28.2% | 34.0% | 33.9% | 36.9% | 42.01% | 37.8% | | |

# 6 Related Work

We discuss the research efforts related to our general objective of monitoring user-provided properties over multi-threaded systems. We discuss first the various works done on the runtime verification of multi-threaded (non CBS-based) programs and then the verification techniques specific to CBSs.

**Monitoring multi-threaded programs.** The approach in [12] presents a tool for runtime model checking safety properties over multi-threaded C/C++ programs. The input program is instrumented so that its behavior can be checked by monitoring its traces. Similarly, the approach in [20] presents a technique to detect violations of safety properties in multi-threaded programs.

Our approach has several noteworthy differences. First, RVMT-BIP is tailored to CBSs and leverages the architecture and composition of components for runtime verification purposes (when it reconstructs global states). Second, RVMT-BIP is not restricted to the verification of safety properties but can handle any linear-time property for which a monitor can be synthesized as a finite-state machine. Moreover, this state-machine can be generated by several existing monitor-synthesis tools (e.g., Java-MOP [8]) since it uses a generic format to express monitors.

We note also the existence of several monitoring tools for programs written in various general-purpose programming languages (e.g., Java-MOP [8], RuleR [1], and MarQ [19] for Java programs; E-ACSL [9], RiTHM [14] for C programs). However, none of these tools includes techniques to soundly monitor multi-threaded programs while preserving their concurrency.

**Verification techniques for CBSs.** To the best of our knowledge, RVMT-BIP is the first tool dedicated to the runtime verification of multi-threaded CBSs. RVMT-BIP is dedicated to the BIP framework but the underlying principles can also be used in most CBS frameworks with formal operational semantics. We review some of them in the following and discuss their available verification techniques.

Fractal [6] is a modular and extensible component model. A Fractal component is an encapsulated runtime entity. A Fractal component is a black box that does not provide any introspection or intercession capability. An interface of a Fractal component is an access point to the component (similar to a port in BIP), that support a finite set of operations. The approach in [10] presents a prototype to runtime check the satisfaction of RV-FTPL (Runtime Verification for FTPL) formulas on Fractal components expressing the authorized evolutions of configurations. FTPL is a temporal logic with patterns to characterize the correct reconfigurations of CBSs under some temporal and architectural constraints. The approach in [10] considers CBSs which architecture evolves at runtime, the verified properties are related to the architecture configurations of the systems, and do not refer to the internal states of components. Moreover, the underlying monitoring algorithms are not tailored to handle multi-threaded CBSs.

The Grid Component Model (GCM) [4] is an extension of Fractal built to accommodate requirements in distributed systems. VerCors [13] is a platform for the specification, analysis, verification, and validation of the GCM-based applications. SOFA (SOFtware Appliances) system [7] is a development and verification framework for large-scale distributed software systems based on hierarchical components. One of the

objective of SOFA is to provide model-checking tools for distributed CBSs. Similarly, Carmen tool [18] is a component model checker for the Fractal and SOFA component model.

# 7 Future work

Several perspectives are opened by the results obtained in this study.

**Further reducing the runtime overhead.** We consider three avenues to further reduce the runtime overhead imposed by monitors. First, using static analysis techniques can help to compute unnecessary runtime checks that can then be avoided at runtime. Second, the detection of the maximum degree of parallelism and level of synchronization among the components concerned in monitored property can lead to an optimized instrumentation technique that could offer better monitoring performance. Third, a dynamic instrumentation technique, enabling the monitor to remove some interactions with components when they are not needed anymore, would reduce the overhead even more.

**Extension to completely distributed models.** Another perspective is to extend the proposed framework for monitoring fully decentralized and completely distributed models where a central controller does not exist. We plan to customize our transformations for generating distributed monitors. Then, using the techniques presented in [5], we plan to automatically generate correct and efficient distributed implementations running on distributed platforms.

**Application to timed components.** Another perspective is to extend the proposed framework for the real-time version of BIP [3] with timed components (whose interactions are guarded by clocks) and monitor them against timed specifications.

# References

[1] Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. 20(3), 675–706 (2010), http://dx.doi.org/10.1093/logcom/exn076 6

[2] 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) 2, 3, ??

[3] 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) 2, 7

[4] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: Gcm: a grid extension to fractal for autonomous distributed components. annals of telecommunications-annales des télécommunications 64(1-2), 5–24 (2009) 6

[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) 7

[6] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Software: Practice and Experience 36(11-12), 1257–1284 (2006) 6

[7] Bureš, T., Hnětynka, P., Plášil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: Software Engineering Research, Management and Applications, 2006. Fourth International Conference on. pp. 40–48. IEEE (2006) 6

[8] Chen, F., Rosu, G.: Java-mop: A monitoring oriented programming environment for java. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 546–550. Springer (2005) 6

[9] Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of c programs. In: Proceedings of SAC '13: the 28th Annual ACM Symposium on Applied Computing. pp. 1230–1235. ACM (2013) 6

[10] Dormoy, J., Kouchnarenko, O., Lanoix, A.: Runtime verification of temporal patterns for dynamic reconfigurations of components. In: Formal Aspects of Component Software, pp. 115–132. Springer (2011) 6

[11] 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) **??**, 4, 5.3.1, 5.3.2, 1

[12] Gopalakrishnan, Y.Y.X.C.G., Kirby, R.M.: Runtime model checking of multithreaded c/c++ programs 6

[13] Henrio, L., Kulankhina, O., Liu, D., Madelaine, E.: Verifying the correct composition of distributed components: Formalisation and tool. arXiv preprint arXiv:1502.03515 (2015) 6

[14] Navabpour, S., Joshi, Y., Wu, C.W.W., Berkovich, S., Medhat, R., Bonakdarpour, B., Fischmeister, S.: RiTHM: a tool for enabling time-triggered runtime verification for c programs. In: ACM Symposium on the Foundations of Software Engineering (FSE). pp. 603–606 (2013) 6

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

[16] Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., Combaz, J.: Monitoring concurrent component-based systems. Tech. Rep. TR-2015-5, Verimag Research Report (2015), available at `http://www-verimag.imag.fr/TR/TR-2015-5.pdf` 2, 3, 3

[17] Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., Combaz, J.: Monitoring multi-threaded component-based systems. In: Abraham, E., Huisman, M. (eds.) Proceedings of the 12th International Conference on integrated Formal Methods. LNCS (2016) 1, 3, 3, 3, **??**, 1

[18] Plšek, A., Adámek, J.: Carmen: Software component model checker. In: Quality of Software Architectures. Models and Architectures, pp. 71–85. Springer (2008) 6

[19] Reger, G., Cruz, H.C., Rydeheard, D.E.: Marq: Monitoring at runtime with QEA. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 596–610 (2015) 6

[20] Sen, K., Roşu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Formal Methods for Open Object-Based Distributed Systems, pp. 211–226. Springer (2005) 6

[21] 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) 1