Runtime Verification of Component-Based Systems*

Yliès Falcone[1], Mohamad Jaber[2], Thanh-Hung Nguyen[2],
Marius Bozga[2], and Saddek Bensalem[2]

[1] INRIA, Rennes - Bretagne Atlantique, France
[2] VERIMAG, Université Grenoble I, France
Firstname.Lastname@inria.fr, Firstname.Lastname@imag.fr

Abstract. Verification of component-based systems still suffers from limitations such as state space explosion since a large number of different components may interact in an heterogeneous environment. Those limitations entail the need for complementary verification methods such as *runtime verification* based on dynamic analysis and prone to scalability. In this paper, we integrate runtime verification into the BIP (Behavior, Interaction, and Priority) framework. BIP is a powerful component-based framework for the construction of heterogeneous systems. Our method augments BIP systems with monitors checking a user-provided specification. This method has been implemented in RV-BIP, a prototype tool that we used to validate the whole approach on a robotic application.

1 Introduction

A component-based approach consists in building complex systems by composing components (building blocks). This confers numerous advantages (e.g., productivity, incremental construction, compositionality) that allow to deal with complexity in the construction phase. Component-based systems (CBS) are desirable because they allow reuse of sub-systems as well as their incremental modification without requiring global changes. Their development requires methods and tools supporting a concept of architecture which characterizes the coordination between components. An architecture structures a system and involves components and relationships between the externally visible properties of those components. The global behavior of a system can, in principle, be inferred from the behavior of its components and its architecture. Component-based design is based on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis on coordination constraints between components even if local computations on components are not visible (i.e., components are "black boxes").

BIP (Behavior Interaction Priority). BIP is a general framework supporting rigorous design. It uses a dedicated language and an associated toolset supporting the design flow. The BIP language allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described with Labelled Transition Systems (LTS) extended with data and functions written in C. The description of coordination

* This work is partially supported by the FP7 IP ASCENS.

between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between the interactions and is used also to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a system. It confers BIP strong expressiveness that cannot be matched by other existing formalism dedicated to CBS [1]. Moreover, BIP has a rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model and its implementation.

Runtime-verification (RV) [2,3,4] is an effective technique to ensure, at runtime, that a system meets a desirable behavior. It can be used in numerous application domains, and more particularly when integrating together unreliable software components. In RV, a run of the system under scrutiny is analyzed incrementally using a decision procedure: a *monitor*. This monitor may be generated from a user-provided high level specification (e.g., a temporal formula, an automaton). This monitor aims to detect violation or satisfaction w.r.t. the given specification. Generally, it is a state machine processing an execution sequence (step by step) of the monitored program, and producing a sequence of verdicts (truth-values taken from a truth-domain) indicating specification fulfillment or violation. Recently [4] a new framework has been introduced for runtime verification. This expressive framework, leveraged by a finite-trace semantics and an expressive truth-domain, allows to monitor all specifications expressing a linear temporal behavior.

The proposed approach (informal overview). We introduce a complementary validation technique for CBS in general and BIP systems in particular. We leverage the BIP framework by integrating a component-based version of the runtime verification framework introduced in [4]. Given a specification, our method uniformly integrates a monitor as an additional component in a BIP system that is able to runtime check the satisfaction or violation of the specification. The whole method is implemented in a prototype tool, RV-BIP, that automatically instrument BIP systems with monitors. Thanks to the code generator of BIP, the generated self-monitoring system can be directly translated into an actual C module embedded in the global system whose behavior is checked at runtime against the specification. The whole approach has been evaluated on a real robotic application and our experiments validate the relevance of our method.

Paper Organization. The paper is structured as follows. In Section 2 we give a minimal introduction to the BIP framework. Section 3 defines an abstract RV framework for CBS described in BIP. Section 4 shows how the abstract RV framework is implemented for BIP systems. Section 5 describes RV-BIP, a prototype implementation of our method, used to evaluate our method on a robot application. Section 6 is dedicated to related work. Finally, Section 7 raises some concluding remarks and open perspectives.

Notations. In this paper, we use the following notations. For two domains of elements E and F, we note $[E \rightarrow F]$ (resp. $[E \rightharpoonup F]$) the set of functions (resp. partial functions) from E to F. When elements of E depends on the elements of F, we note $\{e \in E\}_{f \in F'}$, where $F' \subseteq F$, for $\{e \in E \mid f \in F'\}$ or $\{e\}_{f \in F'}$ when clear from context.

2 BIP - Behavior Interaction Priority

In this section we recall the necessary concepts of the BIP framework [5]. BIP is a component-based framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer models the collaboration between components. Interactions are described using sets of ports and connectors between them [6]. The *priority* layer is used to enforce scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

2.1 Component-Based Construction

BIP offers primitives and constructs for modeling and composing complex behaviors from atomic components. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from atomic components by specifying connectors and priorities.

Atomic Components. An atomic component is endowed with a set of local variables X taking values in a domain $Data$. Atomic components synchronize and exchange data with other components through the notion of *port*.

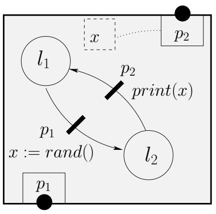
Definition 1 (Port). *A port $p[X']$, where $X' \subseteq X$, is defined by a port identifier p and some data variables in a set X' (referred as the support set).*

Definition 2 (Atomic component). *An atomic component B is defined as a tuple $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where:*

- *(P, L, T) is an LTS over a set of ports P. L is a set of control locations and $T \subseteq L \times P \times L$ is a set of transitions.*
- *X is a set of variables.*
- *For each transition $\tau \in T$:*
  - *g_τ is a Boolean condition over X: the guard of τ,*
  - *$f_\tau \in \{x := f^x(X) \mid x \in X\}^*$: the computation step of τ, a list of statements.*

For $\tau = (l, p, l') \in T$ a transition of the internal LTS, l (resp. l') is referred as the source (resp. destination) location and p is a port through which an interaction with another component can take place. Moreover, a transition $\tau = (l, p, l') \in T$ in the internal LTS involves a transition in the atomic component of the form $(l, p, g_\tau, f_\tau, l')$ which can be executed only if the guard g_τ evaluates to `true`, and f_τ is a computation step: a set of assignments to local variables in X. In the rest of this article, we use the dot notation to denote the elements of atomic components, e.g., $B.P$ denotes the set of ports of an atomic component B.

Example 1 (Atomic component). The figure below shows an example of atomic component with two ports p_1, p_2, a variable x, and two control locations l_1, l_2.



At location l_1, the transition labelled by the port p_1 is possible (the guard evaluates to `true` by default). When an interaction through p_1 takes place, a random value is assigned to the variable x through $x := rand()$. From the control location l_2, the transition labelled by the port p_2 is possible, the variable x is not modified, the value of x is printed and exported through p_2.

Semantics of Atomic Components. The semantics of an atomic component is an LTS over configurations and ports, formally defined as follows:

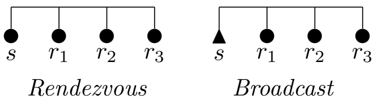
Definition 3 (Semantics of Atomic Components). *The semantics of the atomic component* $(P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ *is an LTS* (P, Q, T_0) *s.t.*

- $Q = L \times [X \to Data]$,
- $T_0 = \{((l, v), p, (l', v')) \in Q \times P \times Q \mid \exists\tau = (l, p, l') \in T : g_\tau(v) \wedge v' = f_\tau(v)\}$.

A configuration is a pair $(l, v) \in Q$ where $l \in L$ is a control location, and $v \in [X \to Data]$ is a valuation of the variables in X. The evolution of configurations $(l_1, v) \overset{p(v_p)}{\to} (l_2, v')$, where v_p is a valuation of variables attached to port p, is possible if there exists a transition $(l_1, p[x_p], g_\tau, f_\tau, l_2)$, s.t. $g_\tau(v) = \mathtt{true}$. As a result, the valuation v of variables is modified to $v' = f_\tau(v[x_p \leftarrow v_p])$.

Creating composite components. Assuming some available atomic components B_1, $\ldots, B_n$, we show how to connect $\{B_i\}_{i\in I}$ with $I \subseteq [1, n]$ using *connectors*.

A connector γ is used to specify possible interactions, i.e., the sets of ports that have to be jointly executed. Two types of ports (*synchron*, *trigger*) are defined, in order to specify the feasible interactions of a connector. A *trigger* port is active: it can initiate an interaction without synchronizing with other ports. It is graphically represented by a triangle. A *synchron* port is passive: it needs synchronization with other ports for initiating an interaction. It is graphically represented by a circle. A feasible interaction of a connector is a subset of its ports s.t. either it contains some trigger, or it is maximal.



The figure on the left shows two connectors: *Rendezvous* (only the maximal interaction $sr_1r_2r_3r_4$ is possible), *Broadcast* (all the interactions containing the trigger port s are possible).

Formally, a connector is defined as follows:

Definition 4 (Connector). *A connector* γ *is a tuple* $(\mathcal{P}_\gamma, t, G, F)$*, where:*

- $\mathcal{P}_\gamma = \{p_i[x_i] \mid p_i \in B_i.P\}_{i\in I}$ *s.t.* $\forall i \in I : \mathcal{P}_\gamma \cap B_i.P = \{p_i\}$,
- $t : \mathcal{P}_\gamma \to \{\mathtt{true}, \mathtt{false}\}$ *s.t.* $t(p) = \mathtt{true}$ *if p is trigger (and* `false` *if synchron),*
- G *is a Boolean expression over the set of variables* $\cup_{i\in I}\, x_i$ *(the guard),*
- F *is an update function defined over the set of variables* $\cup_{i\in I}\, x_i$.

$\mathcal{P}_\gamma$ is the set of connected ports called the support set of γ. The ports in $\mathcal{P}_\gamma$ are tagged with function t indicating whether they are trigger or synchron. Moreover, for each $i \in I$, x_i is a set of variables associated to the port p_i.

A communication between the atomic components of $\{B_i\}_{i\in I}$ through a connector $(\mathcal{P}_\gamma, G, F)$ is defined using the notion of *interaction*:

Definition 5 (Interaction). *A set of ports $a = \{p_j\}_{j \in J} \subseteq \mathcal{P}_\gamma$ for some $J \subseteq I$ is an interaction of γ if one of the following conditions holds: (1) there exists $j \in J$ s.t. p_j is trigger; (2) for all $j \in J$, p_j is synchron and $\{p_j\}_{j \in J} = \mathcal{P}_\gamma$.*

An interaction a has a guard and two functions G_a, F_a, respectively obtained by projecting G and F on the variables of the ports involved in a. We denote by $\mathcal{I}(\gamma)$ the set of interactions of γ. Synchronization through an interaction involves two steps. First, the guard G_a is evaluated, then the update function F_a is applied. If there are several possible interactions inside a connector, we choose the interaction involving the maximum[1] number of ports. One can also add priorities to reduce non-determinism whenever several interactions are enabled. Then, the interaction with the highest priority is chosen.

Definition 6 (Composite Component). *A composite component is defined from a set of available atomic components and a set of connectors. The connection of the $\{B_i\}_{i \in I}$ using the set Γ of connectors is denoted $\Gamma(\{B_i\}_{i \in I})$.*

Note that a composite component obtained by composition of a set of atomic components can be composed with other components in a hierarchical and incremental fashion using the same operational semantics.

Definition 7 (Semantics of Composite Components). *A state q of a composite component $C = \Gamma(B_1, \ldots, B_n)$, where Γ connects the B_i's for $i \in I$, is an n-tuple $q = (q_1, \ldots, q_n)$ where $q_i = (l_i, v_i)$ is a state of B_i. Thus, the semantics of C is precisely defined as a transition system $(Q, A, \longrightarrow)$, where:*

- $Q = B_1.Q \times \ldots \times B_n.Q$,
- $A = \cup_{\gamma \in \Gamma} \{a \in \mathcal{I}(\gamma)\}$ *is the set of all possible interactions,*
- $\longrightarrow$ *is the least set of transitions satisfying the following rule:*

$$\frac{\exists \gamma \in \Gamma : \gamma = (P_\gamma, G, F) \qquad \exists a \in \mathcal{I}(\gamma) \qquad G_a(v(X)) \\ \forall i \in I :\ q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{ai}(v(X)) \qquad \forall i \notin I :\ q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

where $a = \{p_i\}_{i \in I}$, X is the set of variables attached to the ports of a, v is the global valuation of variables, and F_{ai} is the partial function derived from F restricted to the variable associated to p_i.

The meaning of the above rule is the following: if there exists an interaction a s.t. all its ports are enabled in the current state and its guard $(G_a(v(X)))$ evaluates to `true`, then we can fire the interaction. When a is fired, not involved components stay in the same state, and, involved components evolve according to the interaction.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior, possibly restricted using priorities.

Definition 8 (Priority). *Let $C = (Q, A, \longrightarrow)$ be the behavior of the composite component $\Gamma(B_1, \ldots, B_n)$. A* priority model *π is a strict partial order on the set of interactions A. Given a priority model π, we abbreviate $(a, a') \in \pi$ to $a \prec a'$. The component $\pi(C)$ is defined by the behavior $(Q, A, \longrightarrow_\pi)$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:*

[1] If there are several maximal interactions, the choice between them is at random.

$$\frac{q \xrightarrow{a} q' \qquad \nexists a' \in A, \nexists q'' \in Q : a \prec a' \wedge q \xrightarrow{a'} q''}{q \xrightarrow{a}_{\pi} q'}$$

An interaction is enabled in $\pi(C)$ only if it is enabled in C, and, it is maximal according to π among the active interactions in C.

Finally, we consider systems defined as a parallel composition of components together with an initial state.

Definition 9 (System). *A system $\mathcal{S}$ is a pair $(B, Init)$ where B is a component and $Init$ is the initial state of B.*

3 An RV Framework for Component-Based Systems

We adapt classical RV frameworks dedicated to monitoring of sequential monolithic programs to CBS in general, and, to BIP systems in particular. We consider a composite component $C = \Gamma(B_1, \ldots, B_n)$ and a priority model π, where the runtime semantics of $\pi(C)$ is an LTS $(Q, A, \longrightarrow_{\pi})$ as introduced in Definitions 7 and 8.

3.1 Specifications for Component-Based Systems (CBS)

Considered specifications of CBS are state-based specifications expressing some desired behavior. We do not assume any particular specification formalism except that we require it expresses a subset of the possible linear behaviors of CBS. In order to make our approach as general as possible, we only describe the events of the possible specification language. We also assume the existence of a monitor synthesis algorithm from this specification formalism (see Section 3.2). For this purpose, the existing solutions (e.g., [7]) provided by the research efforts in RV can be easily adapted.

We follow a classical approach where events are built over a set of atomic propositions $Prop$. Intuitively, an atomic proposition is a Boolean expression over the states of the components (e.g., "in the component B_1, the variable x should be positive if in the component B_2 the variable y is negative"). More formally, an event of $\pi(C)$ is defined as a state formula over the atomic propositions expressed on components involved in $\pi(C)$. The set of events is defined with the following grammar:

$$\begin{aligned}
\Sigma(\pi(C)) &: \text{Atom} \vee \text{Atom} \mid \text{Atom} \wedge \text{Atom} \mid \text{Atom} \Rightarrow \text{Atom} \mid \neg\ \text{Atom} \\
\text{Atom} &: \text{component.var} == \text{val} \mid \text{component.var} \geq \text{val} \\
&\quad \mid \text{component.loc} == \text{a\_location} \mid \text{component.port} == \text{a\_port} \\
\text{component.var} &: \cup_{i \in [1,n]} B_i.X \\
\text{val} &: v \in \mathit{Data} \\
\text{a\_location} &: s \in \cup_{i \in [1,n]} B_i.L \\
\text{a\_port} &: p \in \cup_{i \in [1,n]} B_i.P
\end{aligned}$$

Let us note $Prop(e)$ the set of atomic propositions used in an event $e \in \pi(C)$. For $ap \in Prop(e)$ we define $used(ap)$ as the set of pairs made of a component and variables that are used to define ap:

```
used(ap) = match (ap) with
        component.var == val → (component,var)
        component.var ≥ val → (component,var)
        component.loc == a_location → (component,loc)
        component.port == a_port → (component,port)
```

3.2 Verification Monitors [4]

A monitor is a procedure consuming events fed by a BIP system and producing an appraisal on the sequence of events read so far. We follow a general approach considering verification monitors as deterministic finite-state machines producing a truth-value (a verdict) in an expressive 4-valued truth-domain $\mathbb{B}_4 \stackrel{\text{def}}{=} \{\bot, \bot_c, \top_c, \top\}$, introduced in [3] and used in [4]. $\mathbb{B}_4$ consists of the possible evaluations of a sequence of events and its possible futures relatively to the specification used to generate the monitor:

- The truth-value $\top_c$ (resp. $\bot_c$) denotes "currently true" (resp. "currently false") and expresses the satisfaction (resp. violation) of the specification "if the system execution stops here".
- The truth-value $\top$ (resp. $\bot$) is a definitive verdict denoting the satisfaction (resp. violation) of the specification: the monitor can be stopped.

We define the notion of monitor for a specification defined relatively to a set of events Σ expressed on a composite component.

Definition 10 (Monitor). *A* monitor $\mathcal{A}$ *is a tuple* $(\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$. *The finite set* $\Theta^{\mathcal{A}}$ *denotes the control states and* $\theta^{\mathcal{A}}_{\text{init}} \in \Theta^{\mathcal{A}}$ *is the initial state. The complete function* $\longrightarrow_{\mathcal{A}} \colon \Theta^{\mathcal{A}} \times \Sigma \to \Theta^{\mathcal{A}}$ *is the transition function. In the following we abbreviate* $\longrightarrow_{\mathcal{A}} (\theta, a) = \theta'$ *by* $\theta \xrightarrow{a}_{\mathcal{A}} \theta'$. *The function* $ver^{\mathcal{A}} : \Theta^{\mathcal{A}} \to \mathbb{B}_4$ *is an output function, producing verdicts (i.e., truth-values) in* $\mathbb{B}_4$ *from control states.*

Such monitors are independent from any specification formalism used to generate them and are able to check any specification expressing a linear temporal specification [4]. Intuitively, runtime verification of a specification with such monitors works as follows. An execution sequence is processed in a lock-step manner. On each received event, the monitor produces an appraisal on the sequence read so far. For a formal presentation of the semantics of the monitor and a formal definition of sequence checking, we refer to [4]. In the remainder, we consider a monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$.

3.3 Runs and Traces of BIP Systems

Runs of BIP systems. Each state $q \in Q$ in the LTS of a component can be seen as an environment mapping variables used in the specification over an alphabet Σ to values.

Definition 11 (Environments in a component). *The set of possible environments in* $\pi(C)$ *is* $Env \stackrel{\text{def}}{=} \cup_{i\in[1,n]}\big(\{state_i \to Q^i\} \cup [B_i.X \to Data]\big)$. *The environment defined by a state* $q = (q_1, \ldots, q_n)$, *where* $q_i = (l_i, v_i)$ *for each* $i \in [1, n]$, *is* $[\![q]\!] \in Env$ *s.t.* $[\![q]\!] \stackrel{\text{def}}{=} \cup_{i\in[1,n]}(\cup_{x_i \in B_i.var}\{x \mapsto v_i(x)\} \cup \cup_{i\in[1,n]}\{state^i \mapsto l_i\}$.

After an interaction bringing the component in a state q, an event e is fired if the state-formula associated to e holds, noted $q \models e$, i.e., when e evaluates to true in $[\![q]\!]$, i.e.,

$[\![q]\!](e) = \mathtt{true}$. Let us note that, after reaching a state of the LTS corresponding to the runtime behaviors of a BIP component, it is always possible to determine whether an event is fired or not, i.e., whether the corresponding state-formula holds or not.

We present the notion of *run* of a composite component and how it is monitored.

Definition 12 (Run of a composite component). *A run of length m of a system $(\pi(C), Init)$ is the sequence of environments $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ s.t.: $q^0 = Init$, and, $\forall i \in [0, m-1] : q^i \in Q \wedge \exists a_i \in A : q^i \xrightarrow{a_i}_\pi q^{i+1}$.*

Definition 13 (Monitoring a run of a system). *The verdict $[\![\mathcal{A}]\!](q^0 \cdot q^1 \cdots q^m)$ stated by $\mathcal{A}$ for a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ is $ver^{\mathcal{A}}(\theta^m)$ where $\forall i \in [0, m-1] : \theta_i \xrightarrow{e}_{\mathcal{A}} \theta_{i+1}$ where e is the unique event enabled in θ_i s.t. $q^{i+1} \models e$.*

Building a trace from a run. As one of the challenges in RV is to lower the performance impact on the target program, we should take care of minimizing the information sent to the monitor. Making the monitor processing directly the run of the target program directly is not a reasonable solution because it would yield prohibiting overhead. Our proposal is to send a relevant abstraction of the run to the monitor that we call a *trace*. Intuitively, given a run, the obtained trace is its minimal abstraction (information wise) that permits to evaluate the specification as if the run was not abstracted. Given $Spec(\Sigma)$, a specification defined over a vocabulary of events Σ, we design an abstraction function $\downarrow_\alpha^\Sigma$ building this minimal abstraction. We thus define a notion of informativeness of environments built from states. Intuitively, an environment ρ_1 is less informative than an environment ρ_2 if it has less variables defined, i.e., $\rho_1 \sqsubseteq \rho_2$ if $Dom(\rho_1) \subseteq Dom(\rho_2)$ and $\forall x \in Dom(\rho_1) : \rho_1(x) = \rho_2(x)$. When monitoring a CBS our aim will be to dynamically build the least informative environment so that the monitoring activity of the system amounts to monitoring with the global state.

Definition 14 (Abstraction function). *The abstraction function $\downarrow_\alpha^\Sigma: Q \to Env$ is the least function s.t.: $\forall q \in Q : \downarrow_\alpha^\Sigma (q) = \rho$ and ρ is s.t.: $\forall x \in Dom([\![q]\!])$:*

$$\rho(x) = \begin{cases} [\![q]\!](x) & \text{if } \exists e \in \Sigma, \exists ap \in Prop(e) : used(ap) = (B_i, x), \text{ with } x \in B_i.X; \\ undef & \text{otherwise.} \end{cases}$$

Property 1 (Abstraction preserves event evaluation). The previous abstraction function adheres to the two following principles:

- soundness: $\forall e \in \Sigma, \forall q \in Q : \downarrow_\alpha^\Sigma (q) \models e \Leftrightarrow q \models e$,
- completeness: $\forall e \in \Sigma, \forall q \in Q : \downarrow_\alpha^\Sigma (q) \models e \vee \downarrow_\alpha^\Sigma (q) \not\models e$.

Soundness states that the concrete and abstracted evaluations are the same. Completeness states that evaluation of all specification events remains possible: abstraction does not erase the needed information from the environment.

Definition 15 (Trace of a composite component). *The trace defined from a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ through an abstraction function $\downarrow_\alpha^\Sigma$ is the sequence of environments defined as $\downarrow_\alpha^\Sigma(q^0) \cdot \downarrow_\alpha^\Sigma(q^1) \cdots \downarrow_\alpha^\Sigma(q^m)$.*

The notion of trace evaluation by a monitor directly follows from the notion of run evaluation. Moreover, the following theorem, which is a direct consequence of Property 1, states that, for runtime verification, there is no difference regarding property evaluation to process the trace instead of the run.

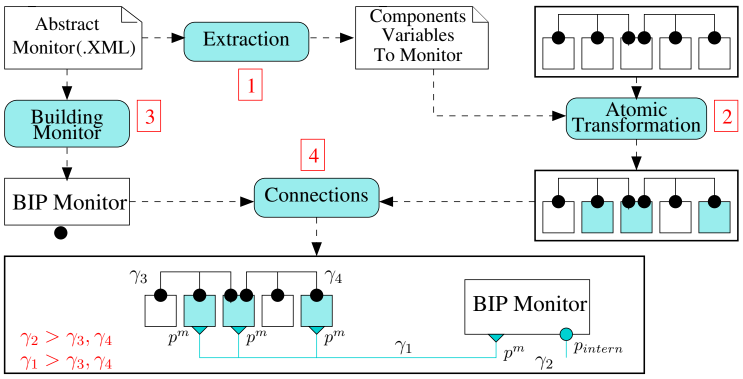


Fig. 1. Overview of the work-flow

Theorem 1 (Trace evaluation vs run evaluation by a monitor). *For $\mathcal{A}$ defined on Σ, the abstraction function $\downarrow_\alpha^\Sigma$, and a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$, we have:*

$$[\![\mathcal{A}]\!]\left([\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]\right) = [\![\mathcal{A}]\!]\left(\downarrow_\alpha^\Sigma(q^0) \cdot \downarrow_\alpha^\Sigma(q^1) \cdots \downarrow_\alpha^\Sigma(q^m)\right)$$

In the next section, we will instrument BIP systems in such a way that, given a specification, the minimal abstraction function (information-wise) is dynamically generated.

4 Verifying the Runtime Behavior of BIP Systems

This section presents how we instrument and integrate an abstract monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$ into a BIP system made of a composite component $C = \Gamma(B_1, \ldots, B_n)$ and priority rules π. The work-flow is as follows (see Fig. 1):

1. From the input abstract monitor we extract the list of components and their corresponding variables used by the monitor (Section 4.1).
2. For each component and its corresponding variables extracted from the monitor we instrument the selected components so as to observe them (Section 4.2).
3. From the monitor we generate the corresponding atomic component. Then, we add the generated component (a monitor in BIP) to the input composite component (Section 4.3).
4. Finally, we add the new connections between the instrumented atomic components and the monitor in BIP (Section 4.4).

4.1 Extraction of Needed Information

The first step is to retrieve from the monitor the set of components and their corresponding variables that should be monitored. For each selected component, transitions are instrumented to observe the just needed set of variables. For a specification expressed over $\Sigma(\pi(\Gamma(B_1, \ldots, B_n)))$ and its monitor, $comp(\Sigma)$ is the subset of $\cup_{i\in[1,n]}\{B_i\}$ corresponding to the set of components that should be monitored. We also define $occur(\Sigma)$

to be the subset of $\{B_i.loc \mid i \in [1,n]\} \cup \{B_i.port \mid i \in [1,n]\} \cup \bigcup_{i\in[1,n]} B_i.X$ denoting the set of variables used in the specification. Then from $occur(\Sigma)$, we sort the variables according to the component B_i (where $B_i \in comp(\Sigma)$) they are related to: $c\_v() = [1,n] \rightarrow \{B_i.loc\}_{i\in[1,n]} \cup \{B_i.port\}_{i\in[1,n]} \cup \bigcup_{i\in[1,n]} B_i.X$ s.t. $c\_v(i)$ is the set of variables related to component B_i.

4.2 Instrumentation of Atomic Component

For a composite component $\Gamma(B_1,\ldots,B_n)$, we transform each atomic component B_i, $i \in [1,n]$, so that it is able to interact with the monitor, if necessary.

Definition 16 (Instrumenting atomic components). *Given* $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ *s.t.* $B = B_i \in \{B_1,\ldots,B_n\}$, *we define a new atomic component*

$$B^m = \begin{cases} B & \text{if } B \notin comp(\Sigma) \\ (P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau\in T^m}, \{f_\tau\}_{\tau\in T^m}) & \text{otherwise} \end{cases}$$

where, $(P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau\in T^m}, \{f_\tau\}_{\tau\in T^m})$ *is defined as follows:*

– $X^m = X \cup \{loc \mid B_i.loc \in c\_v(i)\} \cup \{port \mid B_i.port \in c\_v(i)\}$;
– $P^m = P \cup \{p^m[c\_v(i)]\}$,
– $L^m = L \cup \{l_\tau\}_{\tau\in inst(T)}$, *where* $inst(T)$ *is defined as follows:*

$$inst(T) = \begin{cases} T & \text{if } \{B_i.loc, B_i.port\} \cap c\_v(i) \neq \emptyset \\ \{\tau \in T \mid c\_v(i) \cap var(f_\tau) \neq \emptyset\} & \text{otherwise} \end{cases}$$

– $T^m = T \setminus inst(T) \cup \{in(\tau) \mid \tau \in inst(T)\} \cup \{out(\tau) \mid \tau \in inst(T)\}$, *where,*
  - $in(\tau) = (l, p, f_{in(\tau)}, g_\tau, l_\tau)$, *where*

$$f_{in(\tau)} = \begin{cases} f_\tau & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [loc := \text{“}l\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [port := \text{“}p\text{”}] & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \in c\_v(i) \\ f_\tau; [loc := \text{“}l\text{”}; port := \text{“}p\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \in c\_v(i) \end{cases}$$

  - $out(\tau) = (l_\tau, p^m, f_{out(\tau)}, \texttt{true}, l')$, *where* $f_{out(\tau)} = []$.

We note $B^m = Instrum(B)$. In X^m, loc and $port$ are variables containing a location name and a port name respectively. In P^m, p^m designates the port created for interacting with the monitor. Moreover, $inst(T)$ is the set of transitions that should be instrumented: we instrument atomic components whose variables are needed by the monitor. T^m designates the transitions in the instrumented atomic component. We instrument the transitions in the corresponding atomic component that are modifying a variable involved with the monitor. If the state or the port of an atomic component is needed, all transitions are instrumented. For each transition $\tau \in inst(T)$ that should be instrumented we add a new transition to interact with the monitor. Transitions are also instrumented by adding new statements to save the state and the port name, if necessary.

Example 2 (Instrumentation of an atomic component). Figure 2 illustrates the instrumentation of the atomic component depicted on the left-hand side into the instrumented component on the right-hand side. For instance, supposing that the state should be monitored, from the transition $\tau_0 = (l_0, p_1, f_{\tau_0}, \texttt{true}, l_1)$ with $f_{\tau_0} = [\text{done} := 0]$, we create a new state l_{τ_0} and the transitions $in(\tau_0) = (l_0, p_1, f_{in}, \texttt{true}, l_{\tau_0})$ with $f_{in} = [done := 0; loc := \text{“}l0\text{”}; port := \text{“}p_1\text{”}]$, and $out(\tau_0) = (l_{\tau_0}, p_1, [], \texttt{true}, l_1)$.

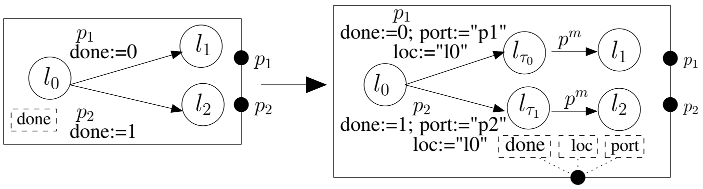


Fig. 2. Instrumentation of an atomic component

4.3 Creating an Atomic Component from a Monitor

From an abstract monitor (cf. Definition 10) given as an XML file, we construct the corresponding atomic component in BIP that interacts with the instrumented atomic components and produces verdicts following the behavior of the original monitor.

Definition 17 (Building monitors in BIP). *From a monitor* $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$*, we define the corresponding atomic component* $M^{\mathcal{A}} = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ *as an atomic component implementing its behavior:*

- $P = \{p^m[X], p_{intern}[]\}$,
- $L = \Theta^{\mathcal{A}} \cup \{q_{mi}\}_{q_i\in\Theta^{\mathcal{A}}}$,
- $T = T_1 \cup T_2$, *where*
  - $T_1 = \{(q_i, p^m, [], \texttt{true}, q_{mi}) \mid q_i \in \Theta^{\mathcal{A}}\}$,
  - $T_2 = \{(q_{mi}, p_{intern}, a, \texttt{print}(ver^{\mathcal{A}}(q_i')), q_i') \mid q_i \xrightarrow{a}_{\mathcal{A}} q_i' \wedge (q_i, p_M, q_{mi}) \in T_1\}$,
- $X = occur(\Sigma)$.

We note $M^{\mathcal{A}} = BuildMon(\mathcal{A})$ and call $M^{\mathcal{A}}$ a BIP monitor. T_1 denotes the set of transitions interacting with the composite component. T_2 is the set of transitions used to display verdicts following the behavior of the original monitor $\mathcal{A}$. The set of variables of the monitor is the set of variables used in the specification (as in Section 4.1).

Example 3 (Transforming an abstract monitor into a BIP monitor). Fig. 3 illustrates the transformation of Definition 17. The atomic component in Figure 3(a) is transformed into the BIP monitor in Figure 3(b).

4.4 Connections

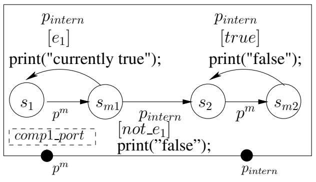
The next step of our transformation is to define the connectors between the transformed components B^m and the BIP monitor $M^{\mathcal{A}}$.

Definition 18 (Connections). *Given* $\mathcal{A}$ *and* $\pi(\Gamma(B_1, \ldots, B_n))$*, the monitored composite component is* $\pi^m(\Gamma^m(B_1^m, \ldots, B_n^m, M^{\mathcal{A}}))$*, where:*

- $B_i^m = Instrum(B_i)$*, for* $i \in [1, n]$*, (see Definition 16);*
- $M^{\mathcal{A}} = BuildMon(\mathcal{A})$*, (see Definition 17);*
- $\Gamma^m = \gamma \cup \{\gamma_1 = (P_{\gamma_1}, \texttt{true}, F_{\gamma_1}), \gamma_2 = (M^{\mathcal{A}}.p_{intern}, \texttt{true}, \emptyset)\}$*, where,*
  - $P_{\gamma_1} = \{B_i.p^m[X_i^m]\}_{B_i\in comp(\Sigma)}\} \cup \{M^{\mathcal{A}}.p^m\}$*, where all ports are synchron;*
  - F_{γ_1}*, the update function, is the identity data transfer from the variables in the ports of the interacting components* B_i *(*$i \in 51, n]$*) to the corresponding variables in the monitor port;*
  - *the type of the port* $M^{\mathcal{A}}.p_{intern}$ *in the connector* γ_2 *is synchron (one and only one interaction is defined by this connector:* γ_2*, see Definition 5);*

```
<VerificationMonitor>
    <State id="s1" initial="true">
        <Transition event="e1" nextState="s1" output="currently_true"/>
        <Transition event="not_e1" nextState="s2" output="false"/>
    </State>
    <State id="s2">
        <Transition event="true" nextState="s2" output="false"/>
    </State>
 </VerificationMonitor>
```

(a) Abstract Monitor



(b) BIP Monitor

Fig. 3. Transforming an abstract monitor into a BIP Monitor

– $\pi^m = \pi \cup \{a \prec a' \mid a \in \cup_{\gamma \in \Gamma} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma_1) \cup \mathcal{I}(\gamma_2)\}$.

The interactions defined by γ_1 and γ_2 have more priority than those defined by Γ (illustrated in Fig. 1). It ensures that, after execution of an interaction by the involved components, the monitor produces verdict before involving other interactions.

4.5 Summary and Discussion

We propose a 4-stage approach to introduce runtime verification for CBS. Our method directly integrates an abstract monitor in a CBS. Thanks to the BIP framework, monitoring of a specification can be taken into account at design stage. Moreover, the actual system, automatically generated from the augmented BIP model, is runtime-checked.

The correctness proof is omitted due to the lack of space, and, relies on the following informal arguments. Our transformations do not modify the data nor the behavior induced by the initial interactions. No deadlock is introduced because the synthesized BIP monitor is always ready to receive events from the instrumented components. Finally, the priorities introduced when connecting the instrumented components to the BIP monitor (Section 4.4) guarantee that the monitor always receives fresh data, i.e., the latest system state.

5 Implementation and Evaluation

5.1 RV-BIP: A Tool for Runtime Verification of BIP Systems

RV-BIP is a Java implementation ($\sim$ 2500 LOC) of the transformations described in Section 4, and, is part of the BIP distribution. RV-BIP takes as input a BIP system and an abstract monitor (an XML file) and then outputs a new BIP system whose behavior is monitored. It uses the following modules (see Fig. 1):

– *Extraction:* this module extracts the components and the corresponding variables used in the monitor. It takes as input an abstract monitor and then outputs a list of components with their corresponding variables,
– *Atomic Transformation:* this module instruments the atomic components selected from the extraction module. It takes as input the output of the *Extraction* module and a BIP file containing the original BIP system,
– *Building Monitor:* this module takes as input an abstract monitor and then outputs the corresponding atomic component,
– *Connections:* this module constructs the new composite component whose behavior is monitored. It takes as input the output from the *Atomic Transformation* and *Building Monitor* modules and then outputs a new composite component.

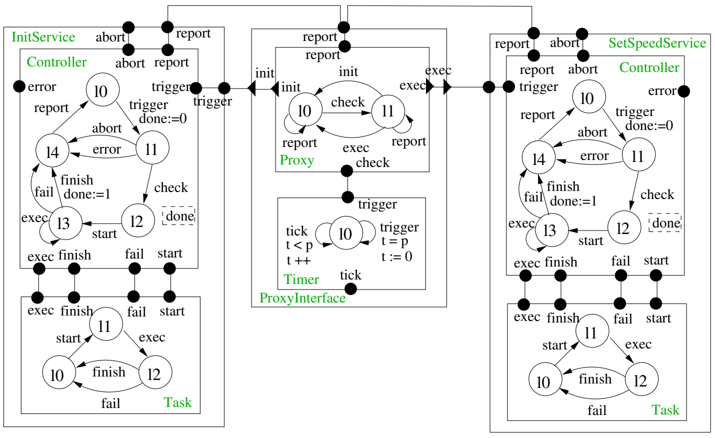


Fig. 4. Two services involving the ordering specification

5.2 Case Study: A Robotic Application

We experimented RV-BIP on a robotic application modeled in BIP: Dala robot [8,9]. The Dala robot is a large and realistic interactive system. It is an infinite system (in terms of states and transitions) that cannot be model-checked.

The functional level of the Dala robot consists of a set of modules. A module is composed of a set of services corresponding to different tasks and a set of posters where the produced data is stored and exchanged between different modules. In this section, due to the lack of space, we present a simplified model of the modules with only the services related to two properties among those we runtime checked.

Execution order: Figure 4 shows a simplified model of Dala. It consists of 3 components: *ProxyInterface*, *InitService* and *SetSpeedService*. *ProxyInterface* communicates with the control layer using the mailbox by executing the transition *check*. *InitService* is responsible for the initialization of the module and *SetSpeedService* performs the main task of the module. According to the received request, *Proxy* triggers either *InitService* or *SetSpeedService*. Each service has a status variable *done*: value 1 means that the corresponding task has been successfully executed. A service can be triggered through the port *trigger*, then it executes its task by taking the transition *start* and finally it returns to the initial location by the transition *finish* when the task is done. The execution order of some services are important. In this module, *InitService* initializes the robot and should be successfully executed before *SetSpeedService* sets the speed parameter of the robot. This requirement is formalized as "φ_1 and φ_2", see Table 1.

Data freshness: In Dala, the modules communicate by a set of posters. Data generated by a module is written in a poster that can be accessed by another module. The behavior of the robot might depend on this data, therefore it is necessary that the data is up to date: the data read by a service of a module (called $Reader$) must be fresh enough compared to the moment it has been written (by a service called $Writer$). If t_1 and t_2 respectively are the moments of reading and writing actions, then the difference between t_1 and t_2 must be less than a specific duration δ, i.e., $|t_2 - t_1| \leq \delta$.

Table 1. Formalization of the requirements for the Dala robot

φ_1: $(e_1)^*$, where, e_1: $(SetSpeedService.port{=}{=}\text{“}trigger\text{”} \wedge ProxyInterface.port{=}{=}\text{“}exec\text{”}){\Rightarrow}(InitService.done{=}{=}1)$	
φ_3: $(e_1)^*$, where, e_1: $(Reader.port{=}{=}\text{“}read\text{”} \wedge poster.port{=}{=}\text{“}read\text{”} \wedge Clock.port{=}{=}\text{“}getTime\text{”})$ ${\Rightarrow}(Clock.time{-}poster.wrtime{\leq}2)$	
φ_2: $(e_1 e_2)*$, where, e_1: $InitService.port{=}{=}\text{“}finish\text{”}$ e_2: $SetSpeedService.port{=}{=}\text{“}trigger\text{”}$	φ_4: $(e_1(\epsilon{+}e_2{+}e_2 e_2)e_3)^*$, where, e_1: $Writer.port{=}{=}\text{“}write\text{”}$ e_2: $Clock.port{=}{=}\text{“}tick\text{”}$ e_3: $Reader.read{=}{=}\text{“}read\text{”}$

This requirement is formalized as “φ_3 and φ_4”, see Table. 1. In the model, the time counter is implemented by a component *Clock*, and the *tick* transition occurs every second.

Experiments: Table 2 reports results on checking the ordering and freshness properties of the Dala robot. *Ordering violated* and *Ordering guaranteed* correspond to the model presented in Fig. 4: the first one might have the violation of the ordering specification whereas the second one always guarantees it. Each consists of an *InitService*, a *SetSpeedService* and ten other services (corresponding to different tasks). It is similar for *Data freshness violated* and *Data freshness guaranteed*: the first might have the violation of the freshness specification whereas the second always guarantees it. We consider two modules: the first has a service responsible for writing data and five other services; the second has a service responsible for reading data produced by the first module and also five other services. In Table 2, *time-no-monitor* indicates the execution time without monitoring; *specification* is the monitored specification; the *optimized* column reports the execution time and the overhead obtained with the monitor that interacts only with the two components involved in the specification; and the *not-optimized* column reports the execution time and the overhead obtained with a monitor that observes all components of the system (even the ones that are not involved in the specification).

The results substantiate our claim that if we monitor only components involved in the specification, using the abstraction technique defined in Section 3 and implemented in Section 4, the overhead is reduced significantly.

6 Related Work

Static verification of component-based system. With the growing demand of scalability and complexity for systems, verification techniques should be used to determine whether a designed system meets its requirements. Static formal verification [10,11,12] is based on mathematical techniques to prove or disprove the correctness of a design w.r.t. a given formal specification. It searches for input patterns which lead to violations of the desired properties and prove the correctness when such violations do not exist. Existing formal verification methods for component-based systems are based on either static analysis or on model-checking [13,14,15].

Approaches based on static analysis consist in computing specific invariants in order to abstract the state space. Though this kind of approaches is less sensitive to state explosion, it still suffers from some limitations. First these techniques are rather limited in terms of the properties they can check: they are mostly limited to safety properties and thus some interesting behavioral properties remains out of the scope of these techniques. Moreover, since these approaches rely on abstraction and over approximation of the state space, they yield several false positives.

Behavioral approaches such as model-checking are based on an exhaustive exploration of the state space of the model obtained from the operational semantics of the specification language. For large systems, this exploration leads to a very large number of states (the well-known state explosion problem). Despite recent advances in model-checking, the state-explosion problem is far from being solved and refrain the use of these methods in component-based systems where the state space tends to become huge due to the number of possible configurations and interactions between components. On the other hand, techniques based on compositional verification [16,17,18] (less sensitive to state explosion) are not applicable when the behavior of some parts of the system is unknown - as it can be the case in BIP when using external C functions.

A compositional verification method based on invariants for checking safety properties in component-based systems is provided in [19,20]. Although the method has been successfully applied to large-scale and complex systems, the use of invariants can deal only with safety properties and might produce many false positive counter examples.

Another compositional approach is design-by-contract [21,22] that considers a property provided by a component as a contract between this component and its environment. For instance [23] provides a method that searches an implementation model that satisfies a given contract. Although the experimental results are promising, it is not always possible to find an implementation model that satisfies a given property. Moreover, the composition of contracts in concurrent systems can be very expensive.

Dynamic verification of component-based systems. Specification and verification of the behavior of CBS have received some research endeavor. A first series of approaches specify the behavior of components in terms of pre and post-conditions (e.g., with JML) or assertions (e.g., using Eiffel). More recently and closer to our work is the LIME specification language [24] that allows runtime monitoring of temporal properties for component interfaces. Components are seen as black boxes and LIME specifications describe how components should interact with an external application by describing a desired behavior on the calls and returns over the interface.

Comparison with our approach. The limitations of static validation techniques lead us to investigate the use of *runtime verification* as an alternative and complementary technique to validate CBS. Compared to previous dynamic techniques, our approach offer several advantages. First, it uses the latest advances in runtime verification using an expressive 4-valued truth-domain allowing our monitor to be generated using any monitor synthesis framework. Note also that the proposed RV framework only uses information about the events used in the specification. The monitors presented in this paper are bounded to regular properties, however, the expressiveness of the BIP language confers our monitors a potential to be Turing-complete. Moreover, our approach is not limited to monitoring component interfaces. It is often the case that components come with an abstract behavioral model, i.e., components are gray boxes instead of black boxes. Our monitoring framework supports both kinds of approaches. Furthermore, the specifications considered for BIP systems use locations spanning over several components allowing the specification of global behaviors of the system in composition.

Table 2. Results of monitoring the requirements Execution order and Data freshness

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383
		φ_2	1.953	3	9.192	384
Ordering guaranteed	1.836	φ_1	1.984	8.0	8.9	384
		φ_2	1.889	2.8	8.896	384
Data freshness violated	1.638	φ_3	1.684	2,8	4.337	164
		φ_4	1.682	2,6	3.773	130
Data freshness guaranteed	1.634	φ_3	1.678	2,6	4.383	168
		φ_4	1.690	3,4	3.782	131

7 Conclusion and Future Work

This paper introduces runtime verification as a complementary validation technique for component-based systems written in the BIP framework. Our technique is based on a general and expressive runtime verification framework. It dynamically builds a minimal abstraction of the current runtime state of the system so as to lower the performance impact. By generating monitors directly as BIP components, we are able to generate actual monitored C programs. Our approach has been implemented in RV-BIP that smoothly integrate in the existing BIP tool-set. Finally, experimental evaluations on a robotic application substantiate our claims and the feasibility of our approach.

Several research perspectives can be considered. A first direction is to combine the recent advances in RV that use static analysis (see e.g., [25]). In RV, using static analysis techniques may reduce the overhead induced by a monitor by disabling unnecessary runtime checks. Also related to overhead reduction, a dynamic instrumentation technique, enabling the monitor to remove connectors when they are not needed anymore, would reduce the overhead even more. Another possible direction is to extend the proposed framework for runtime enforcement [26]. Runtime enforcement is an extension of RV aiming at circumventing property violation and provides better confidence in system behaviors. A more practical direction is to connect RV-BIP to the various existing monitor synthesis tools available within the RV community.

References

1. Bliudze, S., Sifakis, J.: A Notion of Glue Expressiveness for Component-Based Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
2. Runtime Verification (2001-2010), `http://www.runtime-verification.org`
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. 20, 651–674 (2010)
4. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: [27], pp. 40–59
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12 (2006)

6. Bliudze, S., Sifakis, J.: The algebra of connectors—structuring interaction in BIP. IEEE Transactions on Computers 57, 1315–1330 (2008)
7. Stolz, V.: Temporal assertions with parametrised propositions. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 176–187. Springer, Heidelberg (2007)
8. Fleury, S., Herrb, M., Chatila, R.: GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In: Proceedings of Intelligent Robots and Systems, IROS 1997, pp. 842–848. IEEE, Los Alamitos (1997)
9. Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., Nguyen, T.H.: Toward a more dependable software architecture for autonomous robots. IEEE Robotics and Automaton Magazine, Special issue on Soft. Engineering for Robotics 16, 67–77 (2008)
10. Umrigar, Z.D., Pitchumani, V.: Formal verification of a real-time hardware design. In: DAC 1983: Proceedings of the 20th Design Automation Conference, pp. 221–227. IEEE Press, Los Alamitos (1983)
11. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
12. Clarke, E.M., Emerson, E.A.: Synthesis of synchronisation skeletons for branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
13. McMillan, K.: Symbolic model checking. Kluwer Academic Publishers, Boston (1993)
14. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the 5th Syposium on Logic in Computer science, pp. 428–439 (1990)
15. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19, 7–34 (2001)
16. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of the 4th Annual Symposium on LICS, pp. 353–362. IEEE Computer Society Press, Los Alamitos (1989)
17. Chang, E., Manna, Z., Pnueli, A.: Compositional verification of real-time systems. In: Symposium on Logic in Computer Science. IEEE, Los Alamitos (1994)
18. Long, D.E.: Model Checking, Abstraction, and Compositional Reasoning. PhD thesis, Carnegie Mellon (1993)
19. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. Software Journal, Special Issue on Automated Compositional Verification 4, 181–193 (2010)
20. Bensalem, S., Bogza, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: FMCAD (2010)
21. Meyer, B.: Applying design by contract. Computer 25, 40–51 (1992)
22. Abadi, M., Lamport, L.: Composing specifications. ACM Transaction on Programming Languages and Systems 15, 73–132 (1993)
23. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Reasoning about safety and progress using contracts. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 436–451. Springer, Heidelberg (2010)
24. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The lime interface specification language and runtime monitoring tool. In: [27], pp. 93–100
25. Bodden, E., Lam, P., Hendren, L.J.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In: [28], pp. 183–197
26. Falcone, Y.: You should better enforce than verify. In: [28], pp. 89–105
27. Bensalem, S., Peled, D. (eds.): RV 2009. LNCS, vol. 5779. Springer, Heidelberg (2009)
28. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)