Fully-automated Runtime Enforcement of Component-based Systems with Formal and Sound Recovery

Yliès Falcone, Mohamad Jaber

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Fully-automated Runtime Enforcement of Component-based Systems with Formal and Sound Recovery*

Yliès Falcone[1], **Mohamad Jaber**[2]

[1] Univ. Grenoble-Alpes, Inria, LIG, F-38000, Grenoble, France - e-mail: Ylies.Falcone@imag.fr
[2] American University of Beirut, CMPS, Beirut, Lebanon - e-mail: mj54@aub.edu.lb

The date of receipt and acceptance will be inserted by the editor

Abstract. We introduce runtime enforcement of specifications on component-based systems (CBS) modeled in the BIP (Behavior, Interaction and Priority) framework. Runtime enforcement is an increasingly popular and effective dynamic validation technique aiming to ensure the correct runtime behavior (w.r.t. a formal specification) of a system using a so-called enforcement monitor. BIP is a powerful and expressive component-based framework for the formal construction of heterogeneous systems. Because of BIP expressiveness however, it is difficult to enforce complex behavioral properties at design-time.

We first introduce a theoretical runtime enforcement framework for component-based systems where we delineate a hierarchy of enforceable properties (i.e., properties that can be enforced) according to the number of observational steps a system is allowed to deviate from the property (i.e., the notion of k-step enforceability). To ensure the observational equivalence between the correct executions of the initial system and the monitored system, we show that i) only stutter-invariant properties should be enforced on CBS with our monitors, and ii) safety properties are 1-step enforceable. Second, given an abstract enforcement monitor for some 1-step enforceable property, we define a series of formal transformations to instrument (at relevant locations) a CBS described in the BIP framework to integrate the monitor. At runtime, the monitor observes and automatically avoids any error in the behavior of the system w.r.t. the property. Third, our approach is fully implemented in RE-BIP, an available tool integrated in the BIP tool suite. Fourth, to validate our approach, we use RE-BIP to i) enforce deadlock-freedom on a dining philosophers benchmark, and ii) ensure the correct placement of robots on a map.

* The work reported in this article has been done in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology). The work presented in this paper is supported by the University Research Board (URB) at American University of Beirut.

1 Introduction

Users wanting to build complex and heterogeneous systems dispose of a variety of complementary verification techniques to detect bugs and errors. Techniques are often categorized as static or dynamic according to the analyzed information. Both take as input some system representation, perform some analysis, and yield a verdict indicating the (partial) correctness of the system in addition to some form of feedback to the user. Upon the detection of an error in the system, the user's activity enters a new phase consisting in correcting the system and then submitting the corrected system to the analysis technique. This process is usually time-consuming and not guaranteed to converge within the time frame associated to system implementation. Hence, it is often the case that bugs remain in systems in operation.

Motivations. We aim at marrying software synthesis and dynamic analysis to tackle the aforementioned issue and to provide users with a technique that can guarantee the correctness of (off-the-shelf) systems at runtime. Similar to use runtime verification (cf. [40,31,38,5]) to complement model-checking, we use *runtime enforcement* (cf. [42,33, 22,28]) to complement model repair. While model repair targets correctness-by-construction, runtime enforcement, as proposed in this paper, targets *correctness-at-operation*. Runtime enforcement is a dynamic technique aiming at ensuring the correct runtime behavior of systems using a so-called *enforcement monitor*. At runtime, the monitor consumes information from the execution (e.g., events) and modifies it whenever necessary by, e.g., suppressing forbidden events. Enforcing properties at runtime has been only studied for monolithic systems. Moreover, existing research efforts take instrumentation for granted and do not formally define how to actually modify the runtime behavior of the monitored system.

We target component-based systems (CBSs) expressed in the Behavior, Interaction and Priority (BIP) framework [10, 9,4]. BIP allows to build complex systems by coordinating

the behavior of a set of atomic components. Behavior is described with Labeled Transition Systems that are extended with data and C functions. The coordination between components is done with interactions and priorities between interactions. This layered architecture confers a strong expressiveness to BIP [10]. 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. However, because of BIP expressiveness, it remains difficult to enforce at design-time complex behavioral properties.

Dynamic validation techniques for CBSs. Our general objective is to propose dynamic verification techniques to complement existing static verification techniques for BIP systems. More precisely, we are interested in checking and guaranteeing the conformance of the executions of a BIP system w.r.t. a property φ. Figure 1 illustrates the differences between runtime verification and runtime enforcement for BIP systems. In both techniques, we consider as input an initial BIP system (center of the figure). Only some of the states of the system (marked pink) have an effect on the satisfaction/violation of φ. In previous work, we proposed a runtime verification approach for BIP systems [26] (illustrated on the left-hand side of Fig. 1). From φ and the model of the BIP system, we identify the components (green components) that have to be instrumented. At runtime, the instrumented system produces events that feed a (synthesized) verification monitor at runtime (component $\mathcal{M}$), which, in turn, produces verdicts indicating property satisfaction/violation. As illustrated in Fig. 1, the states traversed by the execution of the system at runtime are colored as follows: (1) light blue for current satisfaction, (2) dark blue for definitive satisfaction, and (3) red for violation. In the execution of the system, intermediate states are added between an executing state whenever the monitor should re-evaluate the property. We have shown in [26] that the initial and runtime verified systems are bisimilar (i.e., no trace is added nor removed). A drawback of this approach is that, upon the violation of φ, the system continues its execution in violating states.

To circumvent this limitation, we propose an original runtime enforcement technique specific to CBSs in general and instantiate it for BIP systems in particular. The right-hand side of Fig. 1 illustrates how runtime enforcement is applied to an existing BIP system. Similarly to the runtime verification technique proposed in [26], the runtime enforcement technique proposed in this paper instruments the system so that it can interact with an enforcement monitor. We synthesize two new BIP components: a runtime enforcement monitor and a so-called "disabler" as BIP components. The runtime enforcement monitor ($\mathcal{E}$) is in charge of checking the property φ and influencing the execution of the system according to the result. More precisely, in case of violation, the enforcement monitor rolls the system back to its previous state, and, in case, of satisfaction, the enforcement monitor lets the system progress. Whenever, the system goes through a transition that leads to a violation (and is rolled back), the disabler prevents this transition from being executed again so to prevent livelocks. Contrarily to the runtime verified system, some traces of the initial system are absent from the system where the property is enforced: the resulting system only produces the behaviors that are correct w.r.t. φ.

Contributions. This paper shows how to easily integrate correctness properties into a CBS. Our approach favors the design and correctness of safety-critical systems by allowing a separation of concerns for system designers. Indeed, the functional part of the system and its safety requirements can be designed in separation, and then latter integrated together with our approach. The resulting supervised system prevents any error from happening. More specifically, the contributions of this paper are as follows:

- to introduce runtime enforcement for CBSs to avoid errors at runtime;
- to introduce a new runtime enforcement paradigm where actions of the system can be canceled (by rolling the system back) and alternative executions can be explored: the runtime enforcement paradigm defined in this paper *prevents* the occurrence of misbehavior;
- to instrument CBSs to observe and minimally alter their behavior;
- to define formal transformations that takes as input a CBS and a desired property to produce a supervised system where the property is enforced: the resulting system produces only the correct executions (of the initial system) w.r.t. the considered property.
- to implement the instrumentation and the transformations in RE-BIP, an available toolset;
- to validate the whole approach by enforcing properties over non-trivial systems (where a static hand-coding of the properties using connectors and priorities would have not been tractable).

Challenges. When synthesizing enforcement monitors for CBSs, the main challenges are:

- to handle the possible interactions and synchronizations between components: when intervening on the behavior of a component by e.g., suppressing the execution of a transition, we need to ensure that the synchronized components are also prevented from performing a connected transition;
- to preserve the observational equivalence between the (correct executions of the) initial system and the monitored system: for this purpose, i) we leverage priorities in BIP, and ii) we identify the set of stutter-invariant properties for which enforcement monitors can be synthesized and integrated into a system;
- to define an efficient and complete instrumentation technique: the monitor receives all events of interest of the property while not degrading the performance of the system.

Note, this paper is an extended version of a paper that appeared in the 30^{th} ACM/SIGAPP Symposium On Applied

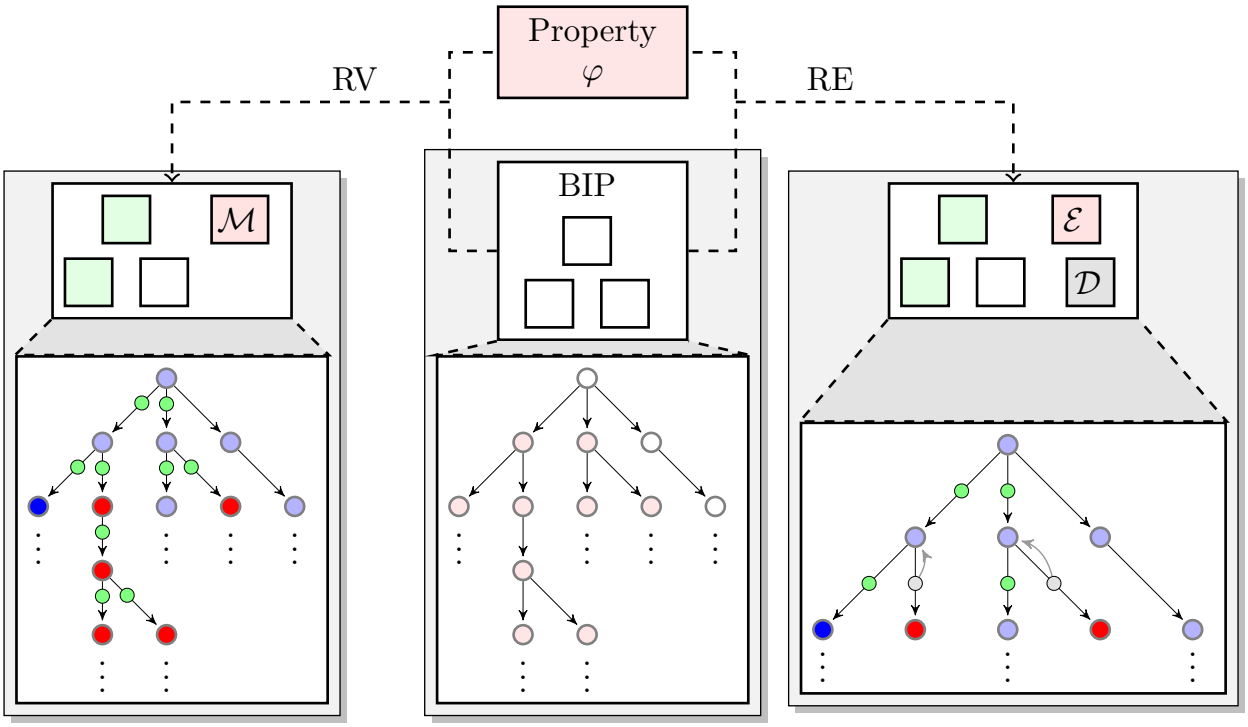


Fig. 1. Runtime verification (RV) versus runtime enforcement (RE) for BIP systems.

Computing - Software Testing and Verification Track [17]. More specifically, this paper brings the following additional contributions. First, we extend our abstract runtime enforcement framework for component-based systems. Second, for the runtime enforcement of BIP systems, we introduce the notion of *disabler*, which is a component preventing instrumented BIP systems to livelock. Third, we release RE-BIP, a toolset for the runtime enforcement of safety properties on component-based systems expressed in the BIP framework. Fourth, we validate our approach by i) illustrating how to enforce deadlock freedom on a dining-philosophers example and ii) to enforce the correct placement of robots on a map. Finally, we provide proofs for the propositions of this paper.

Paper organization. The remainder of this paper is structured as follows. Section 2 introduces some preliminaries and notations. In Sec. 3, we recall the necessary concepts of the BIP framework. Section Sec. 4 presents the language used to describe properties and the instrumentation principles of a CBS for a property. Section 5 presents, at an abstract level, a runtime enforcement framework for component-based systems. Section 6 instantiates the framework proposed in Sec. 5: it shows how to instrument a BIP system to incorporate an enforcement monitor. Section 7 describes RE-BIP, a full implementation of our framework and its evaluation using two benchmarks. Section 8 discusses related work and presents the complementary advantages of our runtime enforcement approach over existing validation techniques. Section 9 draws some conclusions and perspectives. Appendix A contains the proof of correctness of our enforcement monitors for BIP systems.

2 Preliminaries and Notation

We introduce some preliminary concepts and notations.

Functions and partial functions. For two domains of elements E and F, we note $[E \to F]$ (resp. $[E \rightharpoonup F]$) the set of functions (resp. partial functions) from E to F. When elements of E depend 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. For two functions $v \in [X \to Y]$ and $v' \in [X' \to Y']$, the substitution function noted v/v', where $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as: $v/v'(x) = v'(x)$ if $x \in X'$ and $v(x)$ otherwise. A predicate over some domain E is a function in the set $[E \to \{\texttt{true}, \texttt{false}\}]$ where `true` and `false` are the usual Boolean constants. Given, some predicate p over some domain E and some element $e \in E$, we abbreviate $p(e) = \texttt{true}$ (resp. $p(e) = \texttt{false}$) by $p(e)$ (resp. $\neg p(e)$).

Sequences. $\mathbb{N}$ denotes the set of natural numbers. Given a set of elements E, a sequence of length $n \in \mathbb{N}$ over E is denoted $e_1 \cdot e_2 \cdots e_n$ where $\forall i \in [1, n] : e_i \in E$, and the empty sequence is noted ϵ_E, or ϵ when clear from context. When a sequence s is a prefix of a sequence s', we note it $s \preceq s'$. When elements of a sequence are assignments, the sequence is delimited by square brackets, e.g., $[x_1 := expr_1; \ldots; x_n := expr_n]$. Concatenation of assignments or sequences of assignments is denoted by ";". The set of all (finite) sequences over E is noted E^*.

Transition Systems. Labeled Transition System (LTS) are used to define the semantics of (BIP) systems. An LTS defined over an alphabet Σ is a 3-tuple $\langle \mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans} \rangle$ where Lab is a set of labels, Sta is a non-empty set of states and $\mathrm{Trans} \subseteq \mathrm{Sta} \times \mathrm{Lab} \times \mathrm{Sta}$ is the transition relation. A transition $\langle s, e, s' \rangle \in \mathrm{Trans}$ means that the LTS can move from state s to state s' by consuming label e. We abbreviate $\langle s, e, s' \rangle \in \mathrm{Trans}$ by $s \xrightarrow{e}_{\mathrm{Trans}} s'$ or by $s \xrightarrow{e} s'$ when clear from context. Moreover, $s \xrightarrow{e}$ is a short for $\exists s' \in \mathrm{Sta} : s \xrightarrow{e} s'$. The *traces* of LTS $L = \langle \mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans} \rangle$, noted $\mathrm{traces}(L)$, are the finite sequences over $\mathrm{Lab} \cup \mathrm{Sta}$ of the form

$$s_0 \cdot e_0 \cdot s_1 \cdots s_n,$$

for some $n \in \mathbb{N}$, where $s_i \in \mathrm{Sta}$ and $e_i \in \mathrm{Lab}$. A trace starts from the initial state s_0, and follows transition relation Trans. A state $s \in \mathrm{Sta}$ is *reachable* in L if s occurs in one of the traces of L.

3 Behavior Interaction Priority

BIP [4] allows to construct 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 extended with C functions and data and labeled with communication ports. The *interaction* layer models the collaboration (i.e., synchronization and data transfer) between components. The *priority* layer specifies scheduling policies on the interaction layer.

3.1 Atomic Components

An atomic component B is endowed with a set of local variables $B.vars$ ranging over a domain Data. Atomic components synchronize and exchange data through *ports*.

Definition 1 (Port). A port $\langle p, x_p \rangle$ in B is defined by a port identifier p, and a set of attached local variables x_p, where $x_p \subseteq B.vars$. We denote $\langle p, x_p \rangle$ as p, and x_p as $p.vars$.

Definition 2 (Atomic component). An atomic component is a tuple $\left\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \right\rangle$, where:

- $\langle P, L, T \rangle$ 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 finite set of variables,
- for each transition $\tau \in T$, g_τ is a Boolean condition over X: the guard of τ, and $f_\tau \in \{x := f^x(X) \mid x \in X\}^*$: the computation of τ, a sequence of assignments to local variables in X.

For a transition $\tau = \langle l, p, l' \rangle \in T$, l (resp. l') is referred to as the source (resp. destination) location and p is a port for interacting with another component. Moreover, τ involves a transition $\langle l, p, g_\tau, f_\tau, l' \rangle$ which can be executed only if g_τ holds.

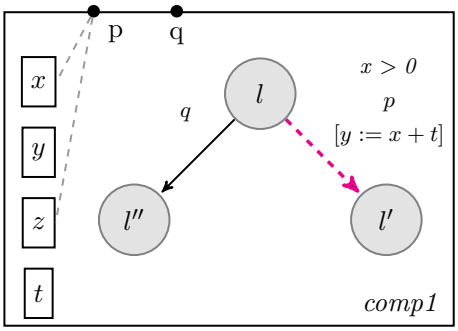


Fig. 2. Atomic component.

In the sequel, $l \xrightarrow{p}_T l'$ (resp. $l \xrightarrow{p}_T$) is a short for $\langle l, p, l' \rangle \in T$ (resp. $\exists l' \in L : l \xrightarrow{p}_T l'$). Given a transition $\tau = \langle l, p, g_\tau, f_\tau, l' \rangle$, $\tau.src$, $\tau.port$, $\tau.guard$, $\tau.func$, and $\tau.dest$ denote l, p, g_τ, f_τ, and l', respectively. Also, the set of variables used in a transition is defined as $\mathrm{var}(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$. We use the dot notation to refer to elements of a component: for an atomic component $B = \left\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \right\rangle$, $B.P$ refers to P, $B.L$ refers to L, etc; for an atomic component B (without definition of the elements), $B.ports$ denotes the set of ports of B, $B.locs$ denotes its set of locations, etc.

Example 1 (Atomic component). Figure 2 depicts an atomic component with variables x, y, z, and t, two ports p and q (p is attached to variables x and z), and three control locations l, l', and l''. At location l, the transition labeled by port q is possible (the guard evaluates to `true` by default) and the (dashed) transition labeled by port p is possible provided x is positive. When an interaction through p takes place, variable y is assigned to the value of $x + t$.

Definition 3 (Semantics of atomic components). The semantics of atomic component $\left\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \right\rangle$ is the LTS $\langle P, Q, T_0 \rangle$, where: (1) $Q = L \times [X \to \mathrm{Data}] \times (P \cup \{\mathtt{null}\})$; and (2) $T_0 = \{\langle \langle l, v, p \rangle, p'(v_{p'}), \langle l', v', p' \rangle \rangle \in Q \times P \times Q \mid \exists \tau = \langle l, p', l' \rangle \in T : g_\tau(v) \wedge v' = f_\tau(v/v_{p'})\}$, where $v_{p'} \in [p'.vars \to \mathrm{Data}]$.

A configuration is a triple $\langle l, v, p \rangle \in Q$ where $l \in L$, $v \in [X \to \mathrm{Data}]$ is a valuation of variables in X, and $p \in P$ is the port of the last-executed transition (or `null` otherwise).

The evolution $\langle l, v, p \rangle \xrightarrow{p'(v_{p'})} \langle l', v', p' \rangle$, where $v_{p'}$ is a valuation of the variables in $p'.vars$, is possible if there exists a transition $\langle l, p', g_\tau, f_\tau, l' \rangle$, s.t. $g_\tau(v) = \mathtt{true}$. Valuation v is modified to $v' = f_\tau(v/v_{p'})$.

3.2 Composite Components

Assuming some atomic components $B_1, \ldots, B_n$, we connect the components in $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ using a *connector*. A connector γ is used to specify possible interactions, i.e., the sets of ports that have to be jointly executed. Two types

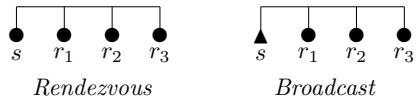


Fig. 3. Using connectors to obtain rendezvous and broadcast.

of ports (*synchron*, *trigger*) are defined in order to specify the feasible interactions of a connector. A *trigger* port (represented by a triangle) is active: the port can initiate an interaction without synchronizing. A *synchron* port (represented by a circle) needs synchronization with other ports to initiate an interaction.

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

- $\mathcal{P}_\gamma = \{p_i \mid p_i \in B_i.P\}_{i \in I}$ s.t. $\forall i \in I : \mathcal{P}_\gamma \cap B_i.P = \{p_i\}$,
- $t \in [\mathcal{P}_\gamma \to \{\texttt{true}, \texttt{false}\}]$ s.t. $t(p) = \texttt{true}$ if p is trigger (and `false` otherwise),
- G is an expression over variables in $\cup_{i \in I}\ p_i.vars$ (the guard),
- F is an update function for variables in $\cup_{i \in I}\ p_i.vars$.

Figure 3 depicts two connectors:

- *Rendezvous*: only the maximal interaction $\{s, r_1, r_2, r_3\}$ is possible,
- *Broadcast*: all interactions containing trigger port s are possible, that is $\big\{ \{s\}, \{s, r_1\}, \{s, r_2\}, \{s, r_3\}, \{s, r_1, r_2\}, \{s, r_2, r_3\}, \{s, r_1, r_3\}, \{s, r_1, r_2, r_3\} \big\}$.

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 either there exists $j \in J$ s.t. p_j is trigger, or, 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, 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 and $\mathcal{I}(\gamma_1) \cup \ldots \cup \mathcal{I}(\gamma_n)$ by $\mathcal{I}(\gamma_1, \ldots, \gamma_n)$. Synchronization through an interaction involves two steps: evaluating G_a, and applying update function F_a.

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

The composite component defined from atomic components $\{B_i\}_{i \in [1,n]}$ and a set of connectors Γ is noted $\Gamma(\{B_1, \ldots, B_n\})$.

Definition 7 (Semantics of composite components). A state q of composite component $\Gamma(\{B_1, \ldots, B_n\})$ is an n-tuple $\langle q_1, \ldots, q_n \rangle$ where $q_i = \langle l_i, v_i, p_i \rangle$ is a state of B_i. The semantics of $\Gamma(\{B_1, \ldots, B_n\})$ is an LTS $\langle Q, A, \longrightarrow \rangle$, 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:

$$\frac{\exists \gamma \in \Gamma : \gamma = \langle P_\gamma, t, G, F \rangle \quad \exists a \in \mathcal{I}(\gamma) : a = \{p_i\}_{i \in I} \quad G_a(v(X)) \quad \forall i \in I :\ q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{a_i}(v(X)) \quad \forall i \notin I :\ q_i = q_i'}{\langle q_1, \ldots, q_n \rangle \xrightarrow{a} \langle q_1', \ldots, q_n' \rangle}$$

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

Interaction a can be fired, whenever all its ports are enabled and its guard ($G_a(v(X))$) holds. Then, involved components evolve according to a and not involved components remain in the same state. Several interactions can be enabled at the same time. Priorities reduce non-determinism: one of the interactions with the highest priority is chosen in a non-deterministic manner.

Definition 8 (Priority). A priority model π over $\Gamma(\{B_1, \ldots, B_n\})$ is a partial order $\pi \subseteq \Gamma \times \Gamma$ on the set of interactions Γ. Adding priority model π to $\Gamma(\{B_1, \ldots, B_n\})$ defines a new composite component $\pi\big(\Gamma(\{B_1, \ldots, B_n\})\big)$ whose behavior is defined by $\langle Q, A, \longrightarrow_\pi \rangle$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:

$$\frac{q \xrightarrow{a} q' \qquad \neg \left(\exists a' \in A, \exists q'' \in Q : \langle a, a' \rangle \in \pi \wedge q \xrightarrow{a'} q'' \right)}{q \xrightarrow{a}_\pi q'}$$

Interaction a is enabled whenever a is maximal according to π. In BIP, maximal progress is expressed at the level of connectors.

Definition 9 (Maximal progress). Given a connector γ and a priority model π, we have:

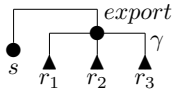
$$\forall a, a' \in \mathcal{I}(\gamma) : (a \subset a' \wedge \langle a', a \rangle \notin \pi) \implies a \prec a'.$$

Finally, system is composed of a composite component and an initial state.

Definition 10 (System). A system $\langle B, Init \rangle$ consists of a composite component B and an initial state $Init \in B_1.L \times \ldots \times B_n.L$.

Hierarchical connectors [10]. Given a connector γ we denote by $\gamma.export$ the exported port of connector γ, which is used to build hierarchical connectors. In that case, we use upward and downward update functions instead of update functions only.

An example of hierarchical connectors is depicted on the right. All interactions containing s and an interaction of γ are possible, i.e., $\big\{ \{s, r_1\}, \{s, r_2\}, \{s, r_3\}, \{s, r_1, r_2\}, \{s, r_1, r_3\}, \{s, r_2, r_3\}, \{s, r_1, r_2, r_3\} \big\}$.



4 Properties and Instrumentation of Component-Based Systems

In this section, we define the basis of a specification language that can be used to express properties of CBSs (Sec. 4.1). To remain general, we refrain from introducing any particular operator for specifying temporal behavior but specify the basic ingredients of an adequate specification language for CBSs (namely the atomic propositions and events). Moreover, given a property of interest, we specify at a high-level, the expected behavior of an instrumentation process for CBSs (Sec. 4.2).

Preliminaries and notation. A property Π over Σ is a subset of Σ^*. If a sequence σ belongs to Π, we note it $\Pi(\sigma)$. To evaluate sequences of system events against properties, we shall use an expressive truth-domain $\mathbb{B}_4$ [5] allowing to characterize a sequence according to whether or not it currently satisfies the property, and whether its possible future continuations will certainly satisfy or violate the property. Truth-domain $\mathbb{B}_4$ contains truth-values true ($\top$), false ($\bot$), currently-true ($\top_c$), and currently-false ($\bot_c$). Given a sequence $\sigma \in \Sigma$ and $\Pi \subseteq \Sigma^*$, the evaluation of σ against Π [24] is defined as:

$$[\![\sigma]\!]^{\Pi}_{\mathbb{B}_4} = \begin{cases} \top & \text{if } \Pi(\sigma) \wedge \forall \sigma' \in \Sigma^* : \Pi(\sigma \cdot \sigma'), \\ \top_c & \text{if } \Pi(\sigma) \wedge \exists \sigma' \in \Sigma^* : \neg\Pi(\sigma \cdot \sigma'), \\ \bot_c & \text{if } \neg\Pi(\sigma) \wedge \exists \sigma' \in \Sigma^* : \Pi(\sigma \cdot \sigma'), \\ \bot & \text{if } \neg\Pi(\sigma) \wedge \forall \sigma' \in \Sigma^* : \neg\Pi(\sigma \cdot \sigma'). \end{cases}$$

Safety properties are the prefix closed subsets of Σ^*. We note $Safety(\Sigma)$ the set of *safety* properties over Σ.

4.1 Specifying Properties of Component-Based Systems

Following [26], we consider events as Boolean expressions over atomic propositions. Atomic propositions express conditions on components (e.g., a condition on the lastly executed port, current locations of atomic components, values of variables). More formally, an event of $\pi(C)$ is defined as a state formula over the atomic propositions over components involved in $\pi(C)$. Let AP be the set of atomic propositions defined with the following grammar (where $* \in \{=, \leq\}$):

Atom	::=	$\text{cpnt}_1.\text{var}_1 * \text{cpnt}_2.\text{var}_2$ | cpnt.var $*$ a_val
		| cpnt.loc = a_loc | cpnt.port = a_port
cpnt.var	::=	$x \in \cup_{i\in[1,n]} B_i.vars$
a_val	::=	$v \in \text{Data}$
a_loc	::=	$s \in \cup_{i\in[1,n]} B_i.locs$
a_port	::=	$p \in \cup_{i\in[1,n]} B_i.ports$

In the sequel, we suppose that all atomic propositions appearing in the property affect its truth-value. We use $Prop : \Sigma \to 2^{AP}$ for the set of atomic propositions used in an event $e \in \pi(C)$. More formally, $Prop$ is defined inductively using the rules in Table 1. For $ap \in Prop(e)$, $\text{used}(ap)$ is the set of pairs formed by the components and the variables (or locations or ports) used to define ap. The set $\text{used}(ap)$ is defined using a *pattern-matching*:

$$\begin{aligned} &\texttt{used (ap)} = \texttt{match ap with} \\ &\quad \mid \texttt{cpnt}_1\texttt{.var}_1 * \texttt{cpnt}_2\texttt{.var}_2 \rightarrow \{(\texttt{cpnt}_1, \texttt{var}_1), (\texttt{cpnt}_2, \texttt{var}_2)\} \\ &\quad \mid \texttt{cpnt.var} * \texttt{val} \rightarrow \{(\texttt{cpnt}, \texttt{var})\} \\ &\quad \mid \texttt{cpnt.loc} = \texttt{a\_loc} \rightarrow \{(\texttt{cpnt}, \texttt{loc})\} \\ &\quad \mid \texttt{cpnt.port} = \texttt{a\_port} \rightarrow \{(\texttt{cpnt}, \texttt{port})\} \end{aligned}$$

In the following, we shall use regular properties which are properties that can be (equivalently) defined by a regular expression or a finite-state automaton.

4.2 Instrumentation of Component-Based Systems

At the abstract level considered in this section, we specify the requirement on the instrumentation of CBSs for a given property of interest. Such requirement shall guide the design of an actual instrumentation of BIP systems for runtime enforcement, as done in Sec. 6.

Intuitively, instrumentation should allow to assess whether each atomic action of the considered system modified the state of the system in such a way that new information is relevant for the evaluation of the property of interest. Hence, the instrumentation should consider the semantics of the system and examine each atomic action. If the action is relevant to the property (because it may modify the truth-value of the property), it should use the information in the state reached after the action to produce the corresponding event for the property.

Let us recall that the semantics of a BIP system is expressed as an LTS where the states of the LTS are the global states of the BIP system and labels of the LTS are the interactions between components in the BIP system (atomic steps). Instrumentation can be modeled as a function $\text{inst} : \text{Lab} \times \text{Sta} \to \Sigma \cup \{\epsilon\}$ that takes as input a label and a state of the LTS (semantics of a CBS) and generates either the event that holds in the specification or ϵ if none of the events in the property is concerned.[1] More precisely, given a property Π defined over an alphabet Σ, $\text{inst}((la, q'))$ is defined as follows:

$$\begin{cases} e \text{ if } \exists e \in \Sigma, \exists ap \in Prop(e) : \text{used}(ap) \cap \text{def}(la) \neq \emptyset \\ \epsilon \text{ otherwise} \end{cases}$$

where $e \in \Sigma$ is the event that holds according to the information in q' and la and $\text{def}(la)$ is the classical set of variables that are defined on transition la. Considering the events of the property in Example 2,

- for a label la_1 that modifies variable $comp_1.x$ (e.g., la_1 is of the form $comp_1.x := \ldots$) and a state q'_1 where $comp_1.x$ is equal to -2, we have $\text{inst}((la_1, q'_1)) = comp_1.x < 0$;
- for a label la_2 that does not modify variable $comp_1.x$, we have $\text{inst}((la_2, q'_2)) = \epsilon$, for any state q'_2.

[1] In Sec. 6, we shall concretely define the instrumentation function for CBSs defined in the BIP framework.

$$
\begin{array}{rcl}
Prop(\text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2) & \stackrel{\text{def}}{=} & \{\text{component}_1.\text{var}_1 == \text{component}_2.\text{var}_2\}, \\
Prop(\text{component.var} == \text{val}) & \stackrel{\text{def}}{=} & \{\text{component.var} == \text{val}\}, \\
Prop(\text{component.var} \geq \text{val}) & \stackrel{\text{def}}{=} & \{\text{component.var} \geq \text{val}\}, \\
Prop(\text{component.loc} == \text{a\_location}) & \stackrel{\text{def}}{=} & \{\text{component.loc} == \text{a\_location}\}, \\
Prop(\text{component.port} == \text{a\_port}) & \stackrel{\text{def}}{=} & \{\text{component.port} == \text{a\_port}\}, \\
Prop(e_1 \vee e_2) & \stackrel{\text{def}}{=} & Prop(e_1) \cup Prop(e_2), \\
Prop(e_1 \wedge e_2) & \stackrel{\text{def}}{=} & Prop(e_1) \cup Prop(e_2), \\
Prop(e_1 \implies e_2) & \stackrel{\text{def}}{=} & Prop(e_1) \cup Prop(e_2), \\
Prop(\neg e) & \stackrel{\text{def}}{=} & Prop(e).
\end{array}
$$

Table 1. Rules defining function $Prop$.

5 Abstract Runtime Enforcement for Component-Based Systems

We introduce an abstract runtime enforcement framework specific to CBSs that i) defines a hierarchy of enforceable properties stemming from the constraints arising when enforcing properties on CBSs (Sec. 5.1), and ii) defines how to compose an enforcement monitor with the behavior of a CBS to incorporate monitors (Sec. 5.2).

5.1 Enforceable Properties for Component-Based Systems

k-step tolerance and stutter-invariance delineate enforceable properties. [2]

k-step tolerance. k-step tolerance models the maximal number of steps for which the system can deviate from the property and can still be rolled back. The number of steps may depend on the criticality of the system, the controlability endowed to our enforcement monitors on the system, or the number of system states that the monitor can memorize. Moreover, when a monitor intervenes, it should not destroy any (future) correct behavior and should determine that a deviation is definitive. In other words, when a property is k-step tolerant, on any execution sequence, if the last k prefixes of the execution sequence do not satisfy the property, on the last event the monitor can decide about the action to be taken: either the next event leads to property satisfaction (verdict $\top_c$ or $\top$) and the monitor does not intervene, or the next event leads the property to property violation (verdict $\bot$) and the monitor rolls back the system. It is thus legitimate for the monitor to intervene.

Definition 11 (k-step tolerance). Property Π is k-step tolerant, if

$$\max\left\{|\sigma| \mid \sigma \in \Sigma^* \wedge \exists \sigma' \in \Sigma^* : (1) \wedge (2)\right\} < k,$$

where:

(1) $[\![\sigma']\!]^{\Pi}_{\mathbb{B}_4} = \top_c$, and
(2) $\forall \sigma_p \in \Sigma^* : \sigma_p \preceq \sigma \wedge \sigma_p \neq \epsilon \implies [\![\sigma' \cdot \sigma_p]\!]^{\Pi}_{\mathbb{B}_4} = \bot_c$.

The set of k-step tolerant properties over alphabet Σ is noted $Tol(k, \Sigma)$. $\Pi \subseteq \Sigma^*$ is k-step-tolerant, if the length of its maximal factor σ for which there exists a sequence σ' (without the factor) that evaluates to $\top_c$ (condition (1)) and all sequences $\sigma' \cdot \sigma_p$ obtained by appending a (non-empty) prefix σ_p of σ to σ' evaluate to $\bot_c$ (condition (2)). Constant k additionally represents the maximal "roll-back distance", i.e., the number of observational steps, an enforcement monitor can roll the system back.

Proposition 1 (On k-step tolerant properties). *There exists a hierarchy of properties organized according to k-step tolerance:*

1. $\forall k, k' \in \mathbb{N} : k \leq k' \implies Tol(k, \Sigma) \subseteq Tol(k', \Sigma)$*; and*
2. *for regular properties, k-step tolerance is decidable.*

Proof.

1. The proposition is a straightforward consequence of k-step tolerance (Definition 11).
2. For a regular property, to decide whether it is k-step tolerant or not, one should compute a deterministic finite-state automaton recognizing the sequences contained in the property. Then, one computes the paths between two accepting states that go (only) through non-accepting states. If the length of the maximal path is strictly lower than k, then the property is k-step tolerant.

In the following, we consider monitors that can memorize *one* state of the system and thus restore it up to one observational step.

Proposition 2. *Safety properties are* 1 *step-tolerant in the sense of Definition 11: $Safety(\Sigma) = Tol(1, \Sigma)$.*

Proof. Following the definition of finitary safety properties [34], for a safety property $\Pi \in Safety(\Sigma)$, we have:

$$\begin{array}{l}
\forall \sigma \in \Sigma^* : \\
\quad \sigma \in \Pi \implies (\forall \sigma' \in \Sigma^* : (\sigma' \preceq \sigma \implies \sigma' \in \Pi)).
\end{array}$$

[2] Contrarily to other runtime enforcement frameworks such as [42,33], we do not consider specifications over infinite sequences but finite sequences. Considering only finite sequences avoids dealing with the enforceability issues due to the semantics of the specification formalism (over infinite sequences). For monolithic systems, using enforcement monitors with storing capabilities such as the one in [33] or [28], all properties over finite sequences are enforceable (see [25] for a detailed explanation).

(Manna and Pnueli show that this is a necessary and sufficient condition [34].) That is, safety properties are prefix-closed languages. Using the definition of function $[\![\cdot]\!]^{\Pi}_{\mathbb{B}_4}$, for a safety property Π, $\mathrm{codom}([\![\cdot]\!]^{\Pi}_{\mathbb{B}_4}) \subseteq \{\bot, \top_c, \top\}$.

Remark 1 (k-step tolerant properties, with $k > 1$). Generally, k-step tolerant properties when $k > 1$ are neither safety nor co-safety properties, but are rather "transactional properties".

For a 1-step tolerant property, following Proposition 2, when a monitor detects a deviation on one event, it is legitimate to intervene because all deviations from the normal behavior are definitive. In other words, at any time a monitor ensures that, either the system has performed a correct execution, or if the last "observable" event brings the system into an incorrect state, then the system is brought back to an equivalent state where it is allowed again to deviate from the correct behavior by one event.

However, k-step tolerance is not the only requirement for a property to be enforceable on component-based systems. Properties should also be stutter-invariant, as discussed next.

Stutter-invariance. Stutter-invariance [44] stems from the required instrumentation of CBSs for enforcement monitoring. Monitors should be able to observe the changes in the system that can impact the satisfaction of atomic propositions. Since monitors should be able to revert the global state of a system one step in the past, instrumenting a transition implies to instrument all synchronized transitions (through a port/interaction). This is a consequence of BIP semantics. Note, even if an instrumented transition does not interfere with the variables observed by the monitor, it is necessary to instrument it for recovering purposes. Those transitions might be synchronized with other transitions through some interactions. In that case, when executing one of these (instrumented) interactions, the monitor receives the same event while the system has not changed. Thus, the evaluation of the property should not change.

Definition 12 (Stutter-invariance [44]). Two sequences of events $\sigma, \sigma' \in \Sigma^*$ are stutter-equivalent if there exist $a_0, \ldots, a_k \in \Sigma$ for some $k \in \mathbb{N}$ s.t. σ and σ' belong to the set defined by regular expression $a_0^+ \cdot a_1^+ \cdots a_k^+$. A property $\Pi \subseteq \Sigma^*$ is stutter-invariant, if for any stutter-equivalent sequences $\sigma, \sigma' \in \Sigma^*$, we have ($\sigma \in \Pi$ and $\sigma' \in \Pi$) or ($\sigma \notin \Pi$ and $\sigma' \notin \Pi$).

Remark 2. Determining whether a property is stutter-invariant is decidable for regular properties using an automata-based representation [44]. Determining whether a property is a safety property is obviously decidable for regular properties. For these purposes, the automata-based representation of the property is the monitor.

Enforceable properties on CBSs. Properties enforced on CBSs should comply to the k-step tolerance and stutter-invariance principles previously discussed. Naturally, we can define a notion of k-step enforceability as follows.

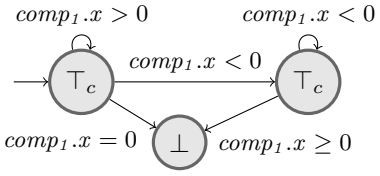


Fig. 4. Runtime oracle.

Definition 13 (k-step enforceability). The set of k-step enforceable properties on CBSs is the set of stutter-invariant k-step tolerant properties.

Note, as a consequence of Proposition 1, there exists a hierarchy of k-step enforceable properties with $k \in \mathbb{N}$.

Based on Proposition 2, in the following, we focus on 1-step enforceable properties, i.e., the set of *stutter-invariant* safety properties as the enforceable properties on CBSs.[3]

5.2 Abstract Runtime Enforcement for Component-Based Systems

Runtime oracle. A runtime oracle is a deterministic finite-state machine that consumes events and produces verdicts.

Definition 14 (Runtime oracle [25]). An *oracle* $\mathcal{O}$ is a tuple $\left\langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma, \rightarrow_{\mathcal{O}}, \mathbb{B}_4, \text{verdict}^{\mathcal{O}} \right\rangle$. The finite set $\Theta^{\mathcal{O}}$ denotes the control states and $\theta^{\mathcal{O}}_{\text{init}} \in \Theta^{\mathcal{O}}$ is the initial state. The complete function $\rightarrow_{\mathcal{O}}$: $\Theta^{\mathcal{O}} \times \Sigma \rightarrow \Theta^{\mathcal{O}}$ is the transition function. In the following, we abbreviate $\rightarrow_{\mathcal{O}} (\theta, a) = \theta'$ by $\theta \xrightarrow{a}_{\mathcal{O}} \theta'$. Function $\text{verdict}^{\mathcal{O}} : \Theta^{\mathcal{O}} \rightarrow \mathbb{B}_4$ is an output function, producing verdicts (i.e., truth-values) in $\mathbb{B}_4$ from control states.

Runtime oracles are independent from any formalism used to generate them and are able to check any linear-time property [24].[4] Intuitively, evaluating a property with an oracle works as follows. An execution sequence is processed in a lockstep manner. On each received event, the oracle produces an appraisal on the sequence read so far. Similar runtime oracles have been defined and used in [5,24] for the runtime verification of monolithic systems, and in [26] for the runtime verification of CBSs. For the formal semantics of oracles and a formal definition of sequence checking, we refer to [5,24, 26].

Example 2 (Runtime oracle). Figure 4 shows an example of a runtime oracle corresponding to the property defined by regular expression $e_1^* \cdot e_2^*$, where e_1 (resp. e_2) denotes that variable x in component $comp_1$ is strictly positive (resp. strictly negative). The underlying property is a safety property (and is thus 1-step tolerant) and is stutter-invariant.

[3] The complexity of the instrumentation depends on the number of steps one wants to be able to roll-back the system (see Sec. 6). Considering more than one step is left for future work.

[4] The runtime oracle is synthesized from a specification, using some monitor-synthesis algorithm. We assume the oracle to be consistent: in any state, it should evaluate logically-equivalent events in the same way.

Enforcement Monitor. An enforcement monitor (EM) is a finite-state machine that transforms a sequence of events from the program to one that evaluates to "good verdicts" of the oracle ($\top, \top_c$). The remaining description of the EM and how it interacts with the system serves as an abstract description of our instrumentation of CBSs in Sec. 6. Compared to enforcement monitors synthesized from properties [42,33,28], the ones introduced in this paper feature the ability to emit *cancellation events* to revert the system back to a state where the underlying property is satisfied.

Definition 15 (Enforcement monitor). The EM associated to runtime oracle $\mathcal{O} = \left\langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma, \rightarrow_{\mathcal{O}}, \mathbb{B}_4, \text{verdict}^{\mathcal{O}} \right\rangle$ is a tuple $\mathcal{E} = \left\langle \Theta^{\mathcal{E}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma \cup \overline{\Sigma} \cup \{com\}, \rightarrow_{\mathcal{E}} \right\rangle$ where:

- $\Theta^{\mathcal{E}} \subseteq \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}$ where $\overline{\Theta}^{\mathcal{O}} = \left\{\theta_e \mid e \in \Sigma \wedge \theta \in \Theta^{\mathcal{O}}\right\}$ is a set of fresh states (copies of states in $\Theta^{\mathcal{O}}$) s.t. $\Theta^{\mathcal{E}}$ is reachable from $\theta^{\mathcal{O}}_{\text{init}}$ with $\rightarrow_{\mathcal{E}}$,
- $\overline{\Sigma} = \{\overline{e} \mid e \in \Sigma\}$ is the set of *cancellation events*,
- $\rightarrow_{\mathcal{E}}$ is the transition function defined as

$$\begin{aligned}
&\rightarrow_{\mathcal{E}} = \theta_1 \cup \theta_2, \text{with}\\
&\theta_1 = \{\langle\theta, e, \theta_c\rangle, \langle\theta_c, com, \theta'\rangle \mid \exists \langle\theta, e, \theta'\rangle \in \rightarrow_{\mathcal{O}}\\
&\qquad\qquad \wedge \text{verdict}^{\mathcal{O}}(\theta') \in \{\top, \top_c\}\}\\
&\theta_2 = \{\langle\theta, e, \theta_e\rangle, \langle\theta_e, \overline{e}, \theta\rangle \mid \exists \theta' \in \Theta : \langle\theta, e, \theta'\rangle \in \rightarrow_{\mathcal{O}}\\
&\qquad\qquad \wedge \text{verdict}^{\mathcal{O}}(\theta') = \bot\}.
\end{aligned}$$

An EM follows the structure of a runtime oracle. For each transition $\langle\theta, e, \theta'\rangle$ in $\rightarrow_{\mathcal{O}}$ leading to a state associated to verdict true or currently-true (i.e., $\text{verdict}^{\mathcal{O}}(\theta') \in \{\top, \top_c\}$), we add: (1) a transition $\langle\theta, com, \theta_c\rangle$ leading to a fresh intermediate state θ_c and; (2) a transition $\langle\theta_c, com, \theta'\rangle$ going to state θ' labeled by the committing event, i.e., com. For each transition $\langle\theta, e, \theta'\rangle$ leading to a state θ' associated to verdict false (i.e., $\text{verdict}^{\mathcal{O}}(\theta') = \bot$), we add: (1) a transition $\langle\theta, e, \theta_e\rangle$ leading to a fresh intermediate state θ_e and; (2) a transition $\langle\theta_e, \overline{e}, \theta\rangle$ back to state θ labeled by the corresponding cancellation event. Note that, as long as the enforcement monitor remains in states in $\Theta^{\mathcal{O}}$, the underlying property is satisfied (i.e., the current trace of the system evaluates to $\top$ or $\top_c$).

Composing a system with a monitor. We define the composition of a system with an enforcement monitor as follows.

Definition 16 (Composing a system with an enforcement monitor). Given a system LTS $L = \langle \text{Lab}, \text{Sta}, \text{Trans}\rangle$, and an enforcement monitor $\mathcal{E} = \left\langle \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma \cup \overline{\Sigma} \cup \{com\}, \rightarrow_{\mathcal{E}} \right\rangle$ for a safety property where states in $\Theta^{\mathcal{O}}$ are associated to verdicts currently-true and true ($\top_c$ and $\top$) and states in $\overline{\Theta}^{\mathcal{O}}$ are associated to verdict bad ($\bot$), the composition, noted $L \otimes_{\text{inst}} \mathcal{E}$, is the LTS $\left\langle \text{Lab} \cup \{\epsilon\}, \text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}), \text{Mon} \right\rangle$ where the transition relation $\text{Mon} \subseteq \left(\text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})\right) \times \text{Lab} \times \left(\text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})\right)$ is defined by the three following semantic rules (where $e = \text{inst}((la, q'))$):

$$\frac{q \xrightarrow{la}_{\text{Trans}} q' \qquad e = \epsilon}{\langle q, \theta\rangle \xrightarrow{la}_{\text{Mon}} \langle q', \theta\rangle} \quad (1)$$

$$\frac{q \xrightarrow{la}_{\text{Trans}} q' \qquad \theta \xrightarrow{e}_{\mathcal{E}} \theta_c \qquad \theta_c \xrightarrow{com}_{\mathcal{E}} \theta' \qquad \theta' \in \Theta^{\mathcal{O}}}{\langle q, \theta\rangle \xrightarrow{la}_{\text{Mon}} \langle q', \theta'\rangle} \quad (2)$$

$$\frac{q \xrightarrow{la}_{\text{Trans}} q' \qquad \theta \xrightarrow{e}_{\mathcal{E}} \theta_e \qquad \theta_e \in \overline{\Theta}^{\mathcal{O}} \qquad \theta_e \xrightarrow{\overline{e}}_{\mathcal{E}} \theta}{\langle q, \theta\rangle \xrightarrow{\epsilon}_{\text{Mon}} \langle q, \theta\rangle} \quad (3)$$

At runtime, an enforcement monitor executes in a lockstep manner with the system. The above rules can be understood as follows:

- Rule (1). When the system emits an event la that is not of interest for the enforcement monitor (i.e., an event s.t. $\text{inst}((la, q')) = \epsilon$), the enforcement monitor lets the system execute without intervening.
- Rule (2). When the system emits an event la that leads to a state θ' associated to either verdict currently-true or verdict true according to the oracle ($\theta' \in \Theta^{\mathcal{O}}$), the enforcement monitor simply follows the system by executing the committing event.
- Rule (3). When the system emits an event la that leads to a state θ_e associated to verdict false according to the oracle ($\theta_e \in \overline{\Theta}^{\mathcal{O}}$), the enforcement monitor executes a cancellation event. State q' can be understood as an unstable state: it is a state where the system never actually stays in because the enforcement monitor inserts immediately a cancellation event. When the enforcement monitor executes event $\overline{e}$, the rule says that the effect of the execution step that leads to event e should be "reverted" on the system: the system and monitor return to their previous state.

Remark 3. The rules in Definition 16 describe the semantics of $S = L \otimes_{\text{inst}} \mathcal{E}$ w.r.t. L and $\mathcal{E}$. On the practical side, the implementation of S consists of i) S which is an instrumented version of L, and ii) $\mathcal{E}$ the enforcement monitor. The implementation of L must contain transitions to communicate with $\mathcal{E}$, and consequently, those transitions change any state of L to a corresponding non-stable state where a decision must be taken (rollback or commit/continue). If the next state of L is a bad state according to the underlying property, L, rollbacks by executing $\overline{e}$. Otherwise, it commits the next state. Henceforth, the implementation of the semantics of S must explicitly include the communication between L and $\mathcal{E}$.

In the following section, we shall define how to implement S following the above rules for a system L and an enforcement monitor $\mathcal{E}$.

On the behavior of the monitored system. Consider a safety property over Σ and a system emitting events over Lab composed with the enforcement monitor (obtained from the property), using an instrumentation function $\text{inst} : \text{Lab} \times \text{Sta} \rightarrow \Sigma \cup \{\epsilon\}$.

Proposition 3 (Weak simulation between the monitored system and the initial system). *Systems* $L = \langle \mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans} \rangle$ *with initial state* $s_0 \in \mathrm{Sta}$ *and* $L \otimes_{\mathrm{inst}} \mathcal{E} = \left\langle \mathrm{Lab} \cup \{\epsilon\}, \mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}), \mathrm{Mon} \right\rangle$ *with initial state* $r_0 \in \mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})$ *as per Definition 16 are such that* L *simulates* $L \otimes_{\mathrm{inst}} \mathcal{E}$ *due to the existence of a relation* $R \subseteq (\mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})) \times \mathrm{Sta}$ *that satisfies the following conditions:*

1. $(r_0, s_0) \in R$
2. $\forall (r, s) \in R, \forall la \in \mathrm{Lab}, \forall r' \in \mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}) :$ $r \xrightarrow{la}_{\mathrm{Mon}} r' \implies (\exists s' \in \mathrm{Sta} : s \xrightarrow{la}_{\mathrm{Trans}} s' \wedge (r', s') \in R)$
3. $\forall (r, s) \in R : r \xrightarrow{\epsilon}_{\mathrm{Mon}} r' \implies (r', s) \in R.$

The proof of this proposition is in Appendix A.2 (p. 22).

Consequently, any execution of the composition seen through the instrumentation function inst does not deviate from the property. Moreover, traces of the composition are stutter-free traces of the initial LTS, that is, traces where duplicate states following an ϵ transition are removed. Note that the initial LTS deviates by at most 1 event before being corrected.

Corollary 1. *Given* $\Pi \in \mathit{Safety}(\Sigma)$*, its enforcement monitor* $\mathcal{E}$*, and a system LTS* L*, we have:*

$$\forall \sigma \in \mathrm{traces}(L \otimes_{\mathrm{inst}} \mathcal{E}) : \mathrm{inst}(\sigma) \in \Pi \\ \wedge \mathrm{stutter\text{-}free}(\sigma_{\downarrow}) \in \mathrm{traces}(L),$$

where:

- *function* inst *is extended to traces in the usual way,*
- $\sigma_{\downarrow}$ *is trace* σ *where the information about the monitor state is erased, and*
- $\mathrm{stutter\text{-}free}(\sigma_{\downarrow})$ *is the stutter-free version of* $\sigma_{\downarrow}$ *where states following an* ϵ *label are removed:* $\mathrm{stutter\text{-}free}(\epsilon) = \epsilon$, $\mathrm{stutter\text{-}free}(q) = q$, $\mathrm{stutter\text{-}free}(t \cdot la \cdot q) = \mathrm{stutter\text{-}free}(t) \cdot la \cdot q$, *and* $\mathrm{stutter\text{-}free}(t \cdot \epsilon \cdot q) = \mathrm{stutter\text{-}free}(t)$.

Summary. This section presents an abstract runtime enforcement framework for component-based systems. Section 4.1 presents how to formally retrieve from a property the set of locations, variables, and ports in atomic components that should be observed to evaluate the property (functions used and *Prop*). These functions shall guide the instrumentation of concrete component-based systems, as exemplified in the next section for BIP systems. Section 5.1 delineates the set of enforceable properties for component-based systems. We show that, contrarily to monolithic systems, only stutter-free properties can be enforced on component-based systems because of the constraints stemming from the synchronization between components. Moreover, we introduce a new paradigm for runtime enforcement where monitors are allowed to roll-back the monitored systems up to k execution steps (the notion of k-step tolerance). Section 5.2 formalizes how such monitors enforce a 1-step enforceable property on a system which behavior is described by an LTS. Enforcement is defined in such a way that the executions of the composed system are the executions of the initial system that follow the property.

6 Runtime Enforcement for BIP Systems

We integrate a runtime oracle $\mathcal{O} = \langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\mathrm{init}}, \Sigma, \rightarrow_{\mathcal{O}}, \mathbb{B}_4, \mathrm{verdict}^{\mathcal{O}} \rangle$ for some (enforceable) property into a BIP system $\left(\pi(\Gamma(\{B_1, \ldots, B_n\})), \left\langle l_0^1, \ldots, l_0^n \right\rangle\right)$ through a series of formal transformations. Certain transformations are defined w.r.t. an atomic component $B = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$.

6.1 Analysis and Extraction of Information

For a property expressed over $\Sigma(\pi(\Gamma(\{B_1, \ldots, B_n\})))$:

- $\mathrm{mon\_vars}(B_i)$ is the set of variables used in the property related to component B_i, formally:

$$\mathrm{mon\_vars}(B_i) \stackrel{\mathrm{def}}{=} \{B_i.x \mid \exists e \in \Sigma, \exists ap \in \mathit{Prop}(e) : (B_i, x) \in \mathrm{used}(ap)\},$$

- occur is the set of all monitored variables, formally:

$$\mathrm{occur} \stackrel{\mathrm{def}}{=} \bigcup_{i \in [1,n]} \mathrm{mon\_vars}(B_i).$$

For instance for the property described by the runtime oracle in Fig. 4, we have $\mathrm{mon\_vars}(\mathit{comp1}) = \{\mathit{comp1}.x\}$.

6.2 Instrumenting Transitions

To instrument the system in such a way that enforcement is as efficient as possible, we should only instrument the transitions that may modify some monitored variables. We denote by $\mathrm{select\_trans}(B)$ the set of the transitions that should be instrumented in B. A transition is instrumented if either (1) it modifies some monitored variables, or (2) some monitored variables are assigned to its port. If the property refers to the (current) location or to an executed port of a component B (e.g., if $B.\mathit{loc} = l_0$ appears in the property), then all transitions of B should be instrumented. Formally: $\mathrm{select\_trans}(B) \stackrel{\mathrm{def}}{=}$

$$\begin{cases} B.\mathit{trans} & \text{if } \{B.\mathit{loc}, B.\mathit{port}\} \cap \mathrm{mon\_vars}(B) \neq \emptyset, \\ \{\tau \in B.\mathit{trans} \mid (\mathrm{var}(\tau.\mathit{func}) \cup \tau.\mathit{port}.\mathit{vars}) \cap \mathrm{mon\_vars}(B) \neq \emptyset\} & \text{otherwise.} \end{cases}$$

For the component in Fig. 2,

$$\mathrm{select\_trans}(\mathit{comp1}) = \{(l, p, x > 0, [y := x + t], l')\},$$

since variable x is attached to port p and $\mathit{comp1}.x \in \mathrm{mon\_vars}(\mathit{comp1})$.

Instrumenting a transition consists in splitting it into four transitions. First, we reconstruct the initial transition. Second, we create a transition to interact with the monitor through port p^m. Finally, we create two transitions: one to recover (through port p^r) when the property is violated and another to continue (through port p^c) otherwise. In case of recovery, the modified variables are restored. Ports p^m, p^r, and p^c are special; their purpose is detailed in Sec. 6.3.

Definition 17 (Instrumenting a transition). For any transition $\tau = \langle l, g, p, f, l' \rangle$ in T, $\text{inst\_trans}(\tau) = \{\tau^i, \tau^m, \tau^c, \tau^r\}$, where:

- $\tau^i = \langle l, g, p, f^i, l_m \rangle$, where f^i is equal to f followed by:
  - $[loc := \text{“}l'\text{”}]$ if $B_i.loc \in \text{mon\_vars}(B_i) \wedge B_i.port \notin \text{mon\_vars}(B_i)$,
  - $[port := \text{“}p\text{”}]$ if $B_i.loc \notin \text{mon\_vars}(B_i) \wedge B_i.port \in \text{mon\_vars}(B_i)$,
  - $[loc := \text{“}l'\text{”}; port := \text{“}p\text{”}]$ if $B_i.loc \in \text{mon\_vars}(B_i) \wedge B_i.port \in \text{mon\_vars}(B_i)$.
- $\tau^m = \langle l_m, \texttt{true}, p^m, [\,], l_r \rangle$,
- $\tau^c = \langle l_r, \texttt{true}, p^c, [\,], l' \rangle$,
- $\tau^r = \langle l_r, \texttt{true}, p^r, f^r, l \rangle$, where $f^r = [x_1 := x_1^{\text{tmp}}; \ldots; x_j := x_j^{\text{tmp}}]$, with $\{x_1, \ldots, x_j\} = \{x \mid x \in p.vars \vee x := f^x(X) \in f\}$.

Example 3 (Instrumenting a transition). Figure 5 illustrates the instrumentation of the dashed transition in Fig. 2. Upon a recovery transition (with port p_r), the component restores all the variables that are modified when executing the associated transition. Recall that some of the variables could be modified indirectly through the port of the transition (p), e.g., x and z.

Recall that an interaction synchronizes a set of transitions and its execution implies firing all its corresponding transitions. Hence, recovering implies to restore the previous global state of the system. For this purpose, instrumenting a transition $\tau \in \text{select\_trans}(B_i)$ implies the instrumentation of all transitions synchronizing with τ through an interaction. We define rec_trans to be the set of all transitions that should be instrumented. We also define rec_comp to be the set of components that contain at least one instrumented transition, and rec_i to be the set of connectors synchronizing on at least one instrumented transition. Formally:

(1) $\text{rec\_trans-i} \stackrel{\text{def}}{=} \cup_{i \in [1,n]} \text{select\_trans}(B_i)$,
(2) $\text{rec\_trans} \stackrel{\text{def}}{=} \text{rec\_trans-i} \cup \{\tau \mid \exists \gamma \in \Gamma, \exists \tau_k \in \text{rec\_trans-i} : \{\tau.port, \tau_k.port\} \subseteq P_\gamma\}$,
(3) $\text{rec\_comp} \stackrel{\text{def}}{=} \{B_i \mid B_i.T \cap \text{rec\_trans} \neq \emptyset\}$,
(4) $\text{rec\_i} \stackrel{\text{def}}{=} \{a \in \Gamma \mid \exists \tau \in \text{rec\_trans} : \tau.port \in P_\gamma\}$.

6.3 Instrumenting Atomic Components

Let $T_B^r = \text{rec\_trans} \cap B.trans$ be the set of transitions that should be instrumented in B (noted T^r when clear from context). We create new temporary/recovery variables used to store the values of the variables that could be modified

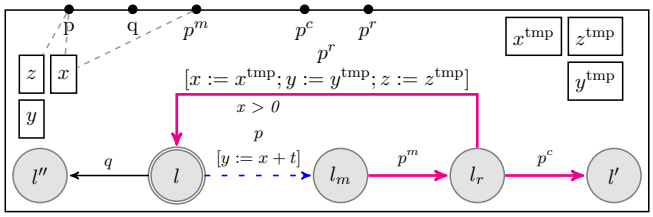


Fig. 5. Instrumenting an atomic component.

on an instrumented transition. More precisely, for each variable that can be modified through a function or attached to a port of an instrumented transition, we create a corresponding temporary variable for it. Given a set of transitions, we define the set of variables that should be recovered as follows: $\text{rec\_vars}(T') \stackrel{\text{def}}{=} \bigcup_{\tau \in T'} \tau.port.vars \cup \text{var}(\tau.func)$. If the enforcement monitor needs to observe the (current) location or the port being executed, we create two new variables[5] *port* and *loc* that store the name of the next location and the name of the port being executed, respectively. We create three new ports:

1. p^m is used to *send the values of monitored variables* to the monitor,
2. p^c is used to receive a *continue* notification from the monitor,
3. p^r is used to receive a *recovery* notification from the monitor.

Finally, we split each of its instrumented transitions in T^r according to Definition 17, and we create new locations accordingly.

Definition 18 (Instrumenting an atomic component). We define the instrumentation function inst that transforms an input atomic component: $\text{inst} \stackrel{\text{def}}{=}$

$$\begin{cases} B & \text{if } B \notin \text{rec\_comp}, \\ \langle P^{\text{inst}}, L^{\text{inst}}, T^{\text{inst}}, X^{\text{inst}}, \{g_\tau\}_{\tau \in T^{\text{inst}}}, \{f_\tau\}_{\tau \in T^{\text{inst}}} \rangle & \text{otherwise.} \end{cases}$$

where:

- $X^{\text{inst}} = X \cup \{v \mid B_i.v \in \text{mon\_vars}(B_i)\} \cup \{x^{\text{tmp}} \mid x \in \text{rec\_vars}(T^r)\}$ where, if $B_i.loc \in \text{mon\_vars}(B_i)$ (resp. $B_i.port \in \text{mon\_vars}(B_i)$), *loc* (resp. *port*) is initialized to l_0^i (resp. $\texttt{null}$), recovery/temporary variables are initialized to the values of their corresponding variables,
- $P^{\text{inst}} = P \cup \{\langle p^m, \text{mon\_vars}(B_i) \rangle\}, \langle p^c, \emptyset \rangle, \langle p^r, \emptyset \rangle\rangle$,
- $L^{\text{inst}} = L \cup \{l_\tau^m \mid \tau \in T^r\} \cup \{l_\tau^r \mid \tau \in T^r\}$,
- $T^{\text{inst}} = (T \setminus T^r) \cup (\bigcup_{\tau \in T^r} \text{inst\_trans}(\tau))$.

Example 4 (Instrumenting an atomic component). Figure 5 shows the instrumentation of atomic component $comp_1$ (see Example 1 and Fig. 2). Note that only the dashed transition of $comp_1$ is instrumented. Also, the variables attached to port p^m (i.e., only $comp1.x$ in this example) are those extracted

[5] Variables created by the transformations have fresh name w.r.t. existing variables of the input system.

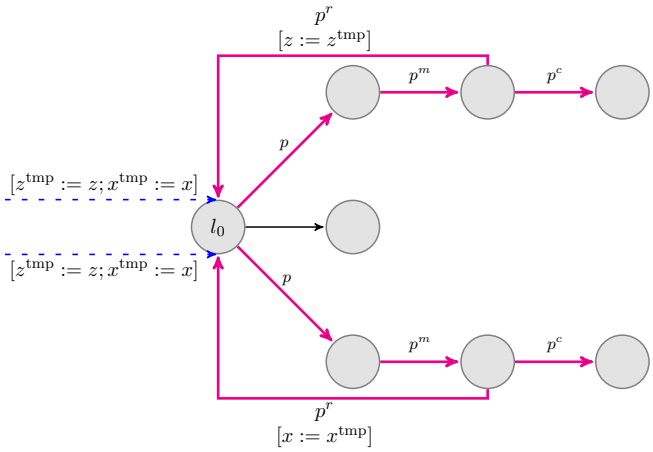


Fig. 6. Injecting backup into an atomic component.

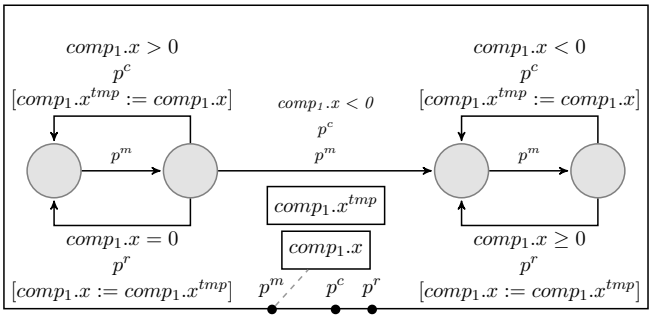


Fig. 7. Enforcement monitor.

from the oracle (see Fig. 4), i.e., monitored variables of that component. Moreover, the function of the recovery transition (i.e., labeled with with p^r) recovers the variables that could be modified, i.e., x, y, and z since variables x and z are attached to port p and y is assigned on the transition.

In the sequel, we consider an instrumented atomic component $B^{\text{inst}} = \text{inst}(B)$. After instrumenting an atomic component, we must also create a backup (in temporary variables) of the variables that could be modified after executing an instrumented transition. For each transition, we select all transitions of the destination that are instrumented, and backup the variables that could be modified on them.

Definition 19 (Backup injection). We define the backup injection function inj that transforms an input (already instrumented) atomic component s.t. $\text{inj}(B^{\text{inst}}) = B^{\text{rec}} = \langle P^{\text{inst}}, L^{\text{inst}}, T^{\text{rec}}, X^{\text{inst}}, \{g_\tau\}_{\tau \in T^{\text{rec}}}, \{f_\tau\}_{\tau \in T^{\text{rec}}} \rangle$, where:

$$T^{\text{rec}} = \Big\{ \langle l, g, p, f; [x_1^{\text{tmp}} := x_1; \ldots; x_j^{\text{tmp}} := x_j], l' \rangle \mid \tau = \langle l, g, p, f, l' \rangle \in T^{\text{inst}} \wedge \{x_1, \ldots, x_j\} = \text{rec\_vars}(\{\tau^i \in B^{\text{inst}}.T^r \mid \tau^i.src = l' \wedge \tau^i.port \in P\}) \Big\}.$$

Next, we consider $B^{\text{rec}} = \text{inj}(B^{\text{inst}})$ with injected backup.

Example 5 (Backup injection). Figure 6 shows backup injection (see the dotted transitions). Variables x and z are backed up on transitions entering l_0 because they are modified on two outgoing transitions.

6.4 Creating an Enforcement Monitor in BIP from an Oracle

We present how a runtime oracle $\mathcal{O}$ is transformed into a BIP enforcement monitor $\mathcal{E}$ that mimics the behavior of the enforcement monitor associated to $\mathcal{O}$ (see Definitions 14 and 15). The generated BIP enforcement monitor receives events from the instrumented atomic components and processes them to produce the same verdicts as the initial abstract oracle. Depending on the state of $\mathcal{E}$, it notifies the instrumented atomic components to continue or to recover.

The enforcement monitor contains a copy of the monitored variables and a backup/temporary copy of them. When the instrumented system executes an interaction that synchronizes at least one instrumented transition, it interacts with the enforcement monitor through port p^m and sends the modified values of monitored variables. Depending on those values, the enforcement monitor produces a verdict and notifies the original system to continue or to recover, accordingly. In case of recovery (resp. continue), the supervised system should also recover (resp. backup) its monitored variables. The behavior of the enforcement monitor is formalized as follows.

Definition 20 (Building an enforcement monitor). From oracle $\mathcal{O}$, we define enforcement monitor $\mathcal{E} = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$ as an atomic component:

- $X = \text{occur} \cup X^{\text{tmp}}$ with $X^{\text{tmp}} = \{x^{\text{tmp}} \mid x \in \text{occur}\}$,
- $P = \{\langle p^m, \text{occur} \rangle, \langle p^c, \emptyset \rangle, \langle p^r, \emptyset \rangle\}$,
- $L = L^\top \cup L^m$ with $L^\top = \{q \mid q \in \Theta^{\mathcal{O}} \wedge \text{verdict}^{\mathcal{O}}(q) \in \{\top, \top_c\}\}$ and $L^m = \{q^m \mid q \in L^\top\}$,
- $T = T^m \cup T^r \cup T^c$ with
  - $T^m = \{ \langle q, p^m, \texttt{true}, [\,], q^m \rangle \mid q \in L^\top \}$,
  - $T^c = \{ \langle q^m, p^c, e, f^c, q' \rangle \mid q \xrightarrow{e}_{\mathcal{O}} q' \wedge \text{verdict}^{\mathcal{O}}(q') = \top \}$, where $f^c = [x_1^{\text{tmp}} := x_1; \ldots; x_j^{\text{tmp}} = x_j]$ if $X^{\text{tmp}} = \{x_1^{\text{tmp}}, \ldots, x_j^{\text{tmp}}\}$,
  - $T^r = \{ \langle q^m, p^r, e, f^r, q \rangle \mid q \xrightarrow{e}_{\mathcal{O}} q' \wedge \text{verdict}^{\mathcal{O}}(q') = \bot \}$, where $f^r = [x_1 := x_1^{\text{tmp}}; \ldots; x_j := x_j^{\text{tmp}}]$ if $X^{\text{tmp}} = \{x_1^{\text{tmp}}, \ldots, x_j^{\text{tmp}}\}$.

Example 6 (Building an enforcement monitor). Figure 7 depicts the enforcement monitor in BIP generated from the runtime oracle in Fig. 4. From the initial state, the enforcement monitor synchronizes with the system by receiving the value of $comp_1.x$ through port p^m. Then, it either recovers (when $comp_1.x = 0$), or continues otherwise. In case of continue (resp. recovery), variable $comp_1.x$ is backed up (resp. recovered).

6.5 Integration - Spin recovery

We define the connection between the instrumented atomic components $\pi(\Gamma(\{B_1^{\text{rec}}, \ldots, B_n^{\text{rec}}\}))$ and enforcement monitor $\mathcal{E}$. We connect the p^m ports of the instrumented components with port p^m of $\mathcal{E}$ (γ_m). All the ports of that connector should be trigger to make all interactions possible.

Because of maximal progress, all the enabled p^m ports of the instrumented components will be synchronized with port p^m of $\mathcal{E}$. The update function of that connector transfers the updated values of the monitored variables from the instrumented atomic components to $\mathcal{E}$.

Then, we connect all continue ports (p^c) of instrumented components, with a connector with trigger ports and connected hierarchically to port p^c of $\mathcal{E}$. The ports of the hierarchical connector are synchron so that the synchronization between ports p^c of instrumented components requires port p^c of $\mathcal{E}$ to be enabled. This is necessary because the instrumented components will be ready to execute both the continue and the recovery ports based on the decision taken by $\mathcal{E}$. Similarly, we connect the recovery ports.

Finally, the priority model is augmented by giving more priority to the interactions defined by the monitored, continue, and recovery connections. Modifying the priority model ensures that, after the execution of an interaction synchronizing some instrumented transition, $\mathcal{E}$ notifies first the system to recover or continue before involving other interactions synchronizing instrumented transitions.

Note that, when some of the ports p^m of the instrumented atomic components are enabled, the port p^m of $\mathcal{E}$ is also enabled. However, the instrumented atomic components could be in a state where none of their p^m ports are enabled. To prevent $\mathcal{E}$ from moving without synchronizing with the components, the port p^m of $\mathcal{E}$ is synchron.

Definition 21 (Integration - Spin Recovery). The composite component is $\pi^{\text{rec}}(\Gamma^{\text{rec}}(B_1^{\text{rec}}, \ldots, B_n^{\text{rec}}, \mathcal{E}))$, where:

- $\Gamma^{\text{rec}} = \Gamma \cup \{\gamma^m, \gamma^{c_1}, \gamma^{c_2}, \gamma^{r_1}, \gamma^{r_2}\}$, where:
  - $\gamma^m = \langle P_{\gamma^m}, t_{\gamma^m}, \texttt{true}, F_{\gamma^m}\rangle$, where:
    - $P_{\gamma^m} = \{\langle B_i.p^m, \text{mon\_vars}(B_i)\rangle\}_{B_i \in \text{rec\_comp}} \cup \{\mathcal{E}.p^m\}$, $t_{\gamma^m}(\mathcal{E}.p^m) = \texttt{false}$ and $\forall p \in P_{\gamma^m} \setminus \{\mathcal{E}.p^m\} : t_{\gamma^m}(p) = \texttt{true}$,
    - F_{γ^m}, the update function, is the identity data transfer from the variables in the ports of the interacting components to the corresponding variables in the oracle port.
  - $\gamma^{c_1} = \langle P_{\gamma^{c_1}}, t_{\gamma^{c_1}}, \texttt{true}, [\,]\rangle$, $\gamma^{c_2} = \langle P_{\gamma^{c_2}}, t_{\gamma^{c_2}}, \texttt{true}, [\,]\rangle$, where:
    - $P_{\gamma^{c_1}} = \{\langle B_i.p^c, \emptyset\rangle\}_{B_i \in \text{rec\_comp}}$ and $\forall p \in P_{\gamma^{c_1}} : t_{\gamma^{c_1}}(p) = \texttt{true}$,
    - $P_{\gamma^{c_2}} = \{\gamma^{c_1}.\mathit{export}, \mathcal{E}.p^c\}$ and $t_{\gamma^{c_2}}(\gamma^{c_1}.\mathit{export}) = t_{\gamma^{c_2}}(\mathcal{E}.p^c) = \texttt{false}$.
  - $\gamma^{r_1} = \langle P_{\gamma^{r_1}}, t_{\gamma^{r_1}}, \texttt{true}, [\,]\rangle, \gamma^{r_2} = \langle P_{\gamma^{r_2}}, t_{\gamma^{r_2}}, \texttt{true}, [\,]\rangle$, where:
    - $P_{\gamma^{r_1}} = \{\langle B_i.p^r, \emptyset\rangle\}_{B_i \in \text{rec\_comp}}$ and $\forall p \in P_{\gamma^{r_1}} : t_{\gamma^{r_1}}(B_i.p^r) = \texttt{true}$,
    - $P_{\gamma^{r_2}} = \{\gamma^{r_1}.\mathit{export}, \mathcal{E}.p^r\}$ and $t_{\gamma^{r_2}}(\gamma^{r_1}.\mathit{export}) = t_{\gamma^{r_2}}(\mathcal{E}.p^r) = \texttt{false}$,
- $\pi^{\text{rec}} = \pi \cup \{\langle a, a'\rangle \mid a \in \cup_{\gamma \in \text{rec\_i}} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma^m, \gamma^{c_1}, \gamma^{c_2}, \gamma^{r_1}, \gamma^{r_2})\}$.

An example of integration with spin recovery is provided in the following sub-section.

Remark 4 (Inefficiency of spin recovery). The instrumented system defined with spin recovery may be inefficient in some cases. For instance, when $\mathcal{E}$ notifies the system to recover, the system may execute again one of the previously-executed "bad" interactions.

Remark 5 (Spin recovery may introduce livelocks). If the system reaches a state, where no further transition is possible, it will enter in a livelock as all transitions will be tried and rolled back indefinitely.

The next section specifically addresses the issues mentioned in Remarks 4 and 5 by introducing an alternative way of connecting components and an enforcement monitor.

6.6 Integration - With Disabler

The system resulting from the integration with spin recovery may exhibit undesirable behaviors at runtime. For instance, when $\mathcal{E}$ notifies the system to recover, the system may execute again one of the previously executed bad interactions. To solve this issue, we create a disabler component that comes as an optimization for the monitored system. The idea is to keep disabled the interactions that led to a property violation (aka bad interactions) and that we have recovered from, until a interaction leading to a state where the property holds (aka good interaction) is found. Note: the system should contain at least one possible good interaction, which can possibly be taken after recovering, if no good interaction exists then the system would reach a deadlock state after the system has exhausted all available interactions. For this purpose, we assume that all connectors of the input BIP system contain only synchron ports, hence each connector represents only one interaction. In the following, we use the terms connector and interaction interchangeably.

For each interaction ($a \in \text{rec\_i}$) connected to an instrumented transition, we associate a transition in the disabler. This transition will be labeled with a port connected to the interaction that corresponds to that transition. That is, to execute that interaction, the port of the corresponding transition of the disabler should be ready as well. Upon executing that interaction the id representing the interaction is sent to the disabler. We also create a continue port p^c and a recoverable port p^r that will be synchronized with $\mathcal{E}$ in case of continue and recovery, respectively. The disabler synchronizes with $\mathcal{E}$ on the recovery and continue ports. On a recovery, $\mathcal{E}$ synchronizes with the instrumented components and with the disabler. The disabler will set the guard of the corresponding last received id to `false`. Consequently, after recovery, the last executed interaction cannot not be taken again. On continue, $\mathcal{E}$ informs the disabler that it should enable all its ports, by re-setting their corresponding guards to true, and now all interactions become valid. For each recoverable interaction, i.e., $a \in \text{rec\_i}$ we assign a positive integer for it: $\text{index} : \text{rec\_i} \to [0, |\,\text{rec\_i}\,| - 1]$.

Definition 22 (Disabler construction). Given the set of recoverable interactions rec_i we construct the disabler $\mathcal{D} = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T}\rangle$, where:

- $P = \{p^r\} \cup \{p^c\} \cup \{\langle p^\gamma, \emptyset\rangle \mid \gamma \in \text{rec\_i}\}$,
- $L = \{l\}$,
- $X = \{enab, id\}$, where $enab$ is an array of Boolean variables initialized to `true` and its size is equal to $|\,\text{rec\_i}\,|$,
- $T = T^r \cup T^c \cup T^{\text{inter}}$, where:
  - $T^r = \{\tau^r\}$, where $\tau^r = \langle l, \texttt{true}, p^r, \big[enab[id] := \texttt{false}\big], l\rangle$,
  - $T^c = \{\tau^c\}$, where $\tau^c = \langle l, \texttt{true}, p^c, \big[enab[0] := \texttt{true}; \ldots; enab[|\,\text{rec\_i}\,| - 1] := \texttt{true}\big], l\rangle$,
  - $T^{\text{inter}} = \big\{\, \langle l, enab[\text{index}(\gamma)], p^\gamma, [id := \text{index}(\gamma)], l\rangle \mid \gamma \in \text{rec\_i}\,\big\}$.

Example 7 (Disabler). Figure 8 provides an example of disabler (component $\mathcal{D}$). We have $\text{rec\_i} = \{a_0, a_1\}$ (a_0 and a_1 contain ports that are attached to instrumented transitions). The disabler contains transitions that correspond to a_0 and a_1. Those transitions are labeled with ports p^{a_0}, p^{a_1} which are connected to interactions a_0 and a_1. Moreover, the disabler contains an array of Boolean variables of size 2. The transitions that correspond to rec_i are guarded with the elements of the array accordingly. In case of recovery, e.g., after executing a_0 (resp. a_1), the corresponding Boolean variable is set to `false`, and hence, interaction a_0 (resp. a_1) is disabled. In case of continue, all the elements of the array are set to `true`.

As in Definition 21, we connect the instrumented system with $\mathcal{E}$, but we also connect the instrumented interactions to their corresponding ports of the disabler. Moreover, we connect the continue port (resp. the recovery port) of $\mathcal{E}$ with the continue port (resp. the recovery port) of the disabler. As in Definition 21, we augment the priority model.

Definition 23 (Integration - with Disabler). Given a BIP enforcement monitor $\mathcal{E}$ and $\pi(\Gamma(\{B_1^{\text{rec}}, \ldots, B_n^{\text{rec}}\}))$ a composite component obtained as described above, that is, $B_i^{\text{rec}} = \text{inj}(\text{inst}(B_1^m))$, and disabler $\mathcal{D}$, we build the composite component $\pi^{\text{rec}}(\Gamma^{\text{rec}}(B_1^{\text{rec}}, \ldots, B_n^{\text{rec}}, \mathcal{E}, \mathcal{D}))$, where:

- $\Gamma^{\text{rec}} = (\Gamma \setminus \text{rec\_i}) \cup \Gamma^{\text{rec\_i}} \cup \{\gamma^m, \gamma^{c_1}, \gamma^{c_2}, \gamma^{r_1}, \gamma^{r_2}\}$,
  - $\Gamma^{\text{rec\_i}} = \{\gamma^{\text{rec\_i}} = (P_{\gamma^{\text{rec\_i}}}, t_{\gamma^{\text{rec\_i}}}, G_\gamma, F_\gamma) \mid \gamma = (P_\gamma, t_\gamma, G_\gamma, F_\gamma) \in \text{rec\_i}\}$ where $P_{\gamma^{\text{rec\_i}}} = P_\gamma \cup \{p^\gamma \mid p^\gamma \in \gamma \in \text{rec\_i}\}$ and $\forall p \in P_{\gamma^{\text{rec\_i}}} : t_{\gamma^{\text{rec\_i}}}(p) = \texttt{false}$,
  - $\gamma^m = (P_{\gamma^m}, t_{\gamma^m}, \texttt{true}, F_{\gamma^m})$, with
    - $P_{\gamma^m} = \{\langle B_i.p^m, \text{mon\_vars}(B_i)\rangle\}_{B_i \in \text{rec\_comp}} \cup \{\mathcal{E}.p^m\}$,
    - $t_{\gamma^m}(\mathcal{E}.p^m) = \texttt{false}$, and $\forall p \in P_{\gamma^m} \setminus \{\mathcal{E}.p^m\} : t_{\gamma^m}(p) = \texttt{true}$,
    - F_{γ^m}, the update function, is the identity data transfer from the variables in the ports of the interacting components B_i ($i \in [1, n]$) to the corresponding variables in the oracle port,
  - $\gamma^{c_1} = (P_{\gamma^{c_1}}, t_{\gamma^{c_1}}, \texttt{true}, [\,])$, $\gamma^{c_2} = (P_{\gamma^{c_2}}, t_{\gamma^{c_2}}, \texttt{true}, [\,])$, with
    - $P_{\gamma^{c_1}} = \{\langle B_i.p^c, \emptyset\rangle\}_{B_i \in \text{rec\_comp}}$ and $\forall p \in P_{\gamma^{c_1}} : t_{\gamma^{c_1}}(p) = \texttt{true}$,
    - $P_{\gamma^{c_2}} = \{\gamma^{c_1}.export, \mathcal{E}.p^c, \mathcal{D}.p^c\}$ and $\forall p \in P_{\gamma^{c_1}} : t_{\gamma^{c_2}}(p) = \texttt{false}$.
  - $\gamma^{r_1} = (P_{\gamma^{r_1}}, t_{\gamma^{r_1}}, \texttt{true}, [\,])$, $\gamma^{r_2} = (P_{\gamma^{r_2}}, t_{\gamma^{r_2}}, \texttt{true}, [\,])$, with
    - $P_{\gamma^{r_1}} = \{\langle B_i.p^r, \emptyset\rangle\}_{B_i \in \text{rec\_comp}}$ and $\forall p \in P_{\gamma^{r_1}} : t_{\gamma^{r_1}}(p) = \texttt{true}$,
    - $P_{\gamma^{r_2}} = \{\gamma^{r_1}.export, \mathcal{E}.p^r, \mathcal{D}.p^r\}$ and $\forall p \in P_{\gamma^{r_2}} : t_{\gamma^{r_2}}(p) = \texttt{false}$,
- $\pi^{\text{rec}} = \pi \cup \{\langle a, a'\rangle \mid a \in \cup_{\gamma \in \text{rec\_i}} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma^m) \cup \mathcal{I}(\gamma^{c_1}) \cup \mathcal{I}(\gamma^{c_2}) \cup \mathcal{I}(\gamma^{r_1}) \cup \mathcal{I}(\gamma^{r_2})\}$.

Example 8 (Integration - With Disabler). Figure 8 shows the supervised system where an enforcement monitor $\mathcal{E}$ and disabler $\mathcal{D}$ have been integrated. In case of spin recovery, we do not include $\mathcal{D}$ and its connections. In this example, we assume that the monitored variables are modified only when executing interactions a_0 and a_1. Consequently, component B_3 remains unchanged. Notice that the expressiveness and modularity of BIP design allows us to add and remove $\mathcal{D}$ without modifying the behaviors of components.

Remark 6. If the system reaches a state, where no further transition is possible, it will enter in a deadlock as all transitions will be tried, rolled back, and disabled successively by the disabler.

Remark 7. When added, the disabler $\mathcal{D}$ might disable an interaction that violates the property and the scheduler would select the next one in terms of priority. For example, consider a composite component with two interactions a_0 and a_1 such that a_0 has more priority than a_1. If a_0 is always enabled, then a_1 could not be enabled according to BIP semantics. However, in the supervised system, if a_0 leads to a bad state, $\mathcal{D}$ will disable that interaction. Consequently, interaction a_1 becomes enabled. This can be seen as a powerful primitive to enforce the correctness of a system by allowing low priority interactions. However, if the property should be enforced while preserving the priority model, then on recovery, $\mathcal{D}$ must disable all interactions with less priority than the last executed one.

6.7 On the Correctness and Behavior of the Supervised System

Correctness of our approach relies on our instrumentation technique and stems from the facts that we consider safety properties and that, as it was similarly expressed at the abstract level, our enforcement monitors roll-back the system by one step as soon as the system emits an event that violates the property.

More formally, the correctness of our transformations can be expressed by mapping the concepts in this section to the concepts of our abstract runtime enforcement framework (Proposition 1). Consider S the supervised system resulting from the previous transformations and the safety property over the alphabet used to synthesize the runtime oracle used as input to our transformations. Instrumenting atomic components of a BIP system and integrating it with the enforcement monitor results in an LTS that is as obtained with the composition operator defined in Definition 16.

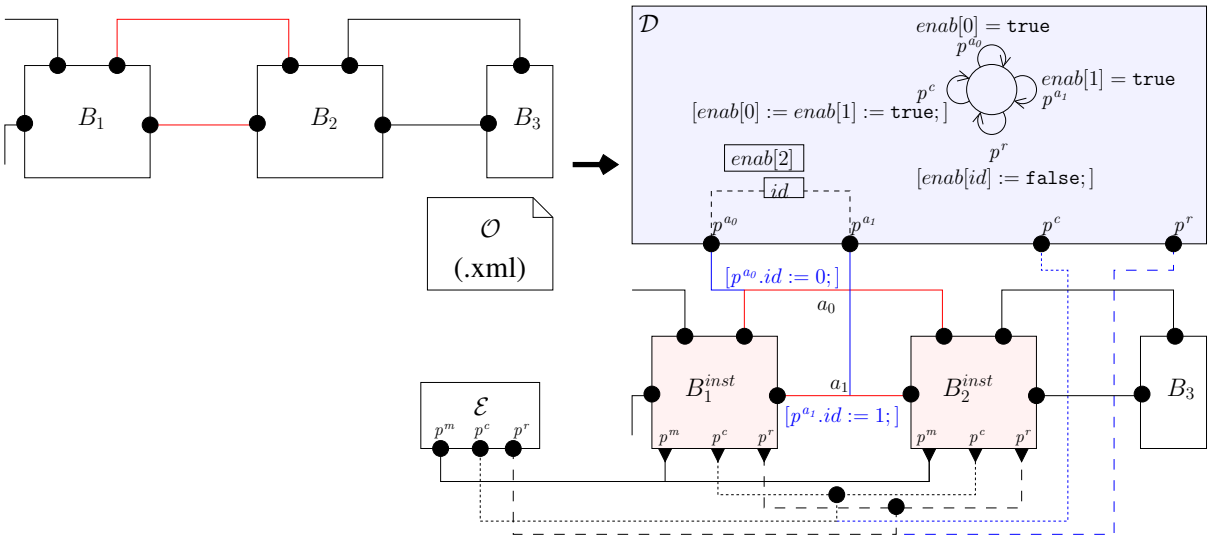


Fig. 8. Supervised system [with/without] disabler (spin recovery – without disabler).

Proposition 4 (Correctness of the transformations). *Consider the mapping described above, Proposition 1 holds for Π, the abstracted LTS and the abstract enforcement monitor. Moreover, the supervised BIP system weakly simulates the restriction of the initial BIP system to its traces satisfying Π.*

Proof. The proof of this proposition is in Appendix A.3 (p. 22).

7 Implementation and Evaluation

In this section, we present RE-BIP, our implementation of the transformations presented in Sec. 6 and its evaluation when enforcing properties on some BIP systems.

7.1 RE-BIP: a Toolset for Runtime Enforcement of BIP Systems

RE-BIP[6] is a Java implementation (8,000 LOC) of the transformations described in Sec. 6, and, is part of the BIP distribution. RE-BIP takes as input a BIP system and an abstract oracle (an XML file) and then outputs a new BIP system whose behavior is supervised at runtime (see Fig. 9). RE-BIP uses the following modules (see Fig. 9):

- *Analysis* module: from the runtime oracle of the property, it collects the variables that should be monitored;
- *Instrumentation* module: according to the analysis, it instruments some of the atomic components;
- *Enforcement Monitor Creation* module: from the runtime oracle (given as an XML file), it generates the corresponding enforcement monitor in BIP;
- *Integration* module: according to the user's input, it creates the supervised system with or without the disabler.

[6] `http://ujf-aub.bitbucket.org/re-bip/`

7.2 Enforcing Deadlock-Freedom on Dining Philosophers

Modeling the dining philosophers problem in BIP. Figure 10 shows a model of the dining philosophers problem in BIP.

Figure 10(a) represents the behavior of a philosopher. From location $init$, a philosopher can take the right fork (port get_r) and moves to location r. From location r, a philosopher can take the left fork (port get_l) and moves to location rl. From location rl, a philosopher can release the two forks (port $release$ and moves to the initial location ($init$).

Figure 10(b) represents the behavior of a fork. From location $init$, a fork can be taken by either the right or left philosopher (i.e., port get) and moves to location $busy$. From location $busy$, the fork can be released by the philosopher that took it (i.e., port $release$) and moves to location $init$.

Figure 10(c) shows an instance of the dining philosophers problem in BIP with two forks and two philosophers. The system is obtained by connecting the get port of each fork with either the get_l or get_r port of a philosopher. Forks and philosophers are connected in such a way that the forks are between the philosophers.

Clearly, at runtime the system may deadlock. As one can observe, enforcing deadlock freedom at design-time is intricate and would require significant modifications to the system. Moreover, it is unlikely that such modifications scale-up well with the number of philosophers and forks.

Enforcing deadlock-freedom with RE-BIP. Enforcing deadlock-freedom with our approach just requires to specify it as an input property. A system, as presented in the previous paragraph, enters a deadlock state if all philosophers are in location r. Note, deadlock-freedom is 1-step enforceable

In our experiments, we varied the numbers of philosophers and forks and compared the execution times before and after enforcing deadlock-freedom, with two possible integration of the monitor (with and without disabler). The BIP system without monitor (which may deadlock) is referred to as

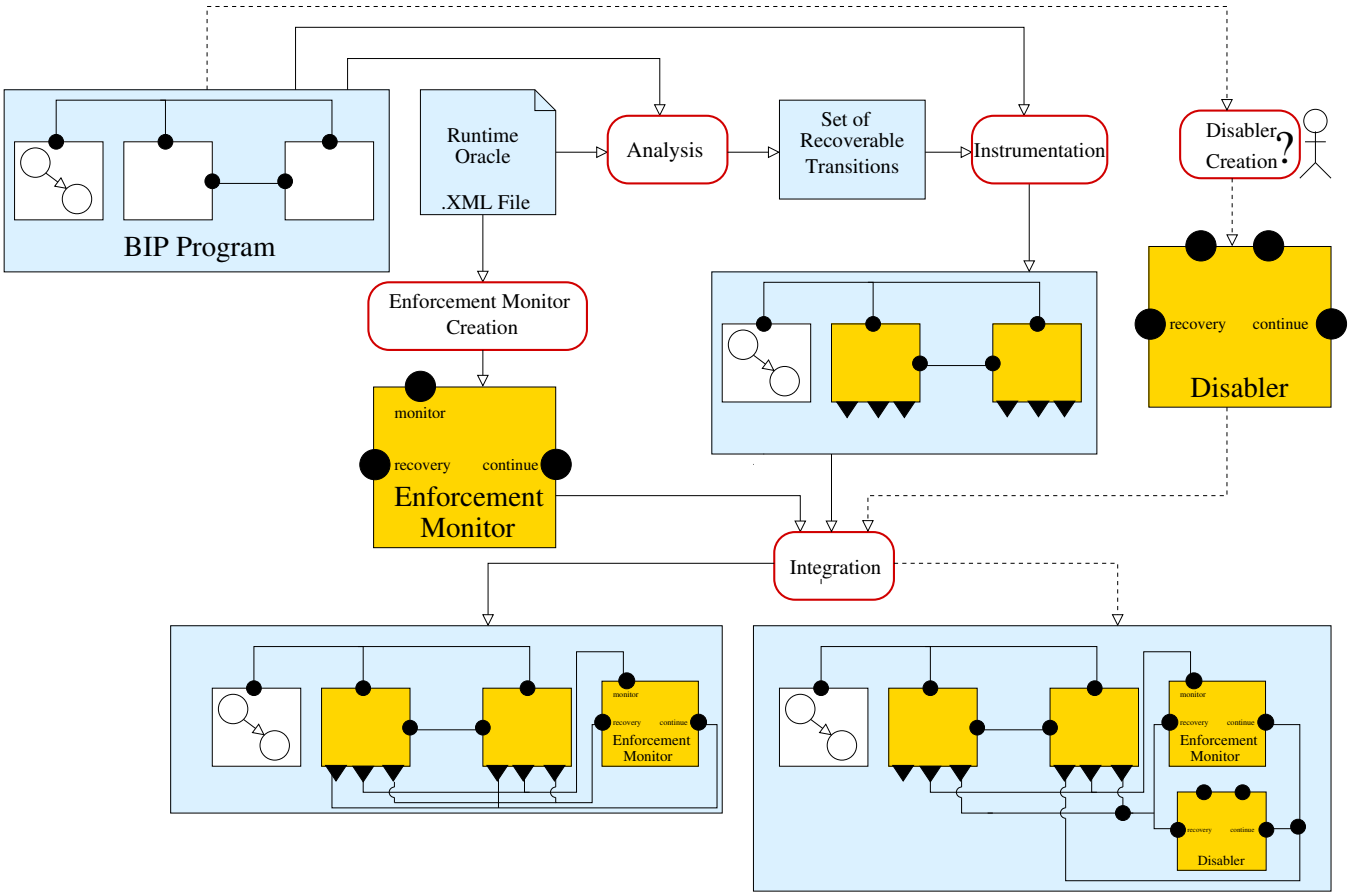
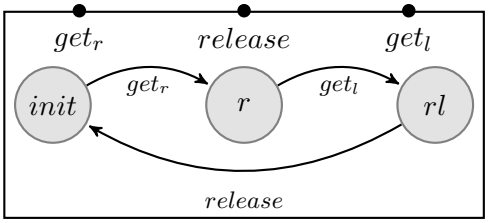
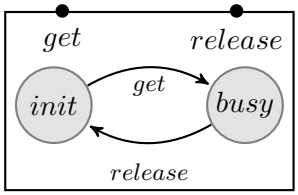


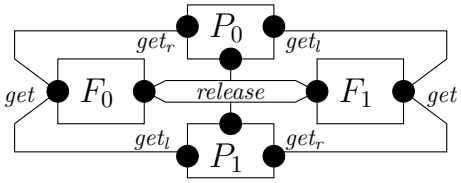
Fig. 9. Toolset for runtime enforcement (RE-BIP).



(a) Philosopher



(b) Fork



(c) Dining philosophers in BIP

Fig. 10. Dining philosophers with possible deadlock.

the initial system, whereas the system with enforcement monitor is referred to as the supervised system. We ran the initial and supervised systems several times (to obtain average values), in such a way that each execution contains $10,000$ fork cycles (i.e., $10,000$ acquisitions and releases).

In Fig. 11, we present some experimental results when using RE-BIP to enforce deadlock-freedom on instances of the dining philosophers problem. The x-axis represents the number of philosophers (and number of forks). The y-axis represents execution time (in seconds). Our results show that the supervised system introduces a reasonable overhead (e.g., 4% in case of 900 philosophers with disabler). In this example, enabling the disabler i) does not introduce deadlocks (there is always at least one good interaction after recovery because a philosopher with a fork on its right can take the fork on its left), and ii) reduces significantly the overhead (from 35% without disabler to 4% with disabler) by disabling the last executed interaction which leads to a deadlock state.

7.3 Enforcing the Correct Placement of Robots

Figure 12 shows a robotic system modeled in BIP. The system contains three robots (referred to as R_1, R_2, and R_3) placed on a map of size $n \times n$. A robot can move up, down, left, and right. Each robot R_i is synchronized with a local controller

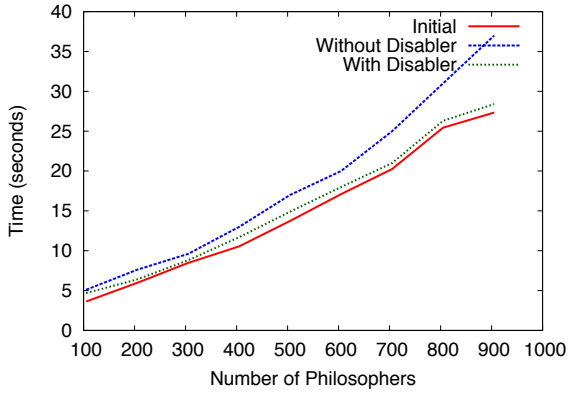


Fig. 11. Performance of enforcement monitors on dining philosophers.

C_i to start and stop it. When a robot starts, it randomly makes $1,000$ moves. The system contains also a global controller C that synchronizes with local controllers to count the number of active robots. In this model, collisions between robots are possible. To avoid collisions, the system must satisfy the following invariant stating that any two (distinct) robots should have at least one different coordinate:

$$\forall i, j \in [1,3] : i \neq j \implies R_i.x \neq R_j.x \vee R_i.y \neq R_j.y.$$

Enforcing the above invariant at design-time would require modifying the behaviors of robots as well as the architecture by adding new interactions. However, using our method, we can take the system off-the-shelf and create a monitor that emits a verdict $\bot$ in case of collision of two robots, otherwise its verdict is $\top_c$, and, the system is automatically instrumented to avoid collision between robots. This permits a separation of concerns between the main functionalities of the system and additional behaviors (e.g., avoid collision, avoid ambush coordinates, limit the number of active robots, etc.). Table 2 shows the execution times (in seconds) to perform 2×10^5 correct (i.e., no collision) steps. We generated four configurations (Supervised, Supervised-d, Supervised-o, and Supervised-o-d) of the supervised system. We use -o (resp. -d) to denote that the system is optimized (i.e., instrumenting only the minimal set of transitions (resp. to denote that the disabler is integrated in the system). For each configuration, we ran the system on maps of different sizes ($n = 2, 5$, and 100). Obviously, the number of collisions and rollbacks decreases with the size of the map. For example, if we consider configuration Supervised-o and the map of size $n = 2$ we obtain $400,280$ rollbacks and the execution time to perform 2×10^5 correct steps is 224 seconds. In this case, enabling the disabler (i.e., Supervised-o-d configuration) reduces the number of rollbacks and hence the execution time (177 seconds). Clearly, the optimized configurations outperform the non-optimized ones. For maps of sizes 5 and 100 the disabler slightly reduces the number of collisions since the probability to take again the same step that has lead to a collision is reduced. Thus, in that case, enabling the disabler does not improve the execution time but adds a reasonable overhead because of the interactions with the disabler.

8 Related Work

8.1 Model Repair

Recent efforts (e.g., [18]) aim at adapting model-checking abstraction techniques to model repair. Our approach fundamentally differs from model repair in two main respects. First, our approach operates at runtime: we do not statically modify systems as our properties are expressive enough so that model-checking is undecidable or does not scale. Second, our objective is to minimally alter the initial behavior of the system. Correct executions are preserved and yield observationaly-equivalent executions.

8.2 Theories of Fault-tolerance

Given a fault-tolerant program that complies to some safety and liveness properties upon the occurrence of faults, the authors of [11] decompose the program into detectors and correctors preserving and ensuring the satisfaction of safety and liveness properties, respectively. The approach assumes a fault-model as input (i.e., a labeling of all system transitions as normal, faulty, and recovery), and then characterizes the conditions for a system to converge to a normal behavior. The considered systems are non-masking, i.e., i) faults are recovered within a finite number of recovery actions, and ii) and the system always progresses. Both the later and our approach target BIP systems. However, [11] takes as input fault-tolerant programs and assumes fault-tolerance being encoded inside the target program. In [2], the system is seen as a collection of guarded commands. In [11], fault detection and recovery span across multiple components. Both approaches fall short in meeting the modularity requirement of CBSs. Indeed, programs in [2] do not have their own state-space. The fault models considered in [11] assumes fault detection and recovery to concern several components with inter-dependent interactions.

8.3 Supervisory Control

The runtime enforcement approach proposed in this paper has similarities with supervisory control. In a supervisory control approach, a model of the supervised system is used to synthesize a controller. At runtime, the controller executes in parallel with the supervised system. Should the system try to perform an illegal action, the controller disables it, and the system is stopped. Several approaches have applied supervisory control to component-based systems, e.g., [13,30] for integrating synchronous reconfiguration managers in Fractal systems, [29] for enforcing coherency strategies of autonomic managers administrating components. The focus of [13,30] is rather on adaptive systems where controllers operate on the configuration of components that evolve dynamically, while the focus of this paper is rather on statically defined components and safety-critical properties which involve the internal

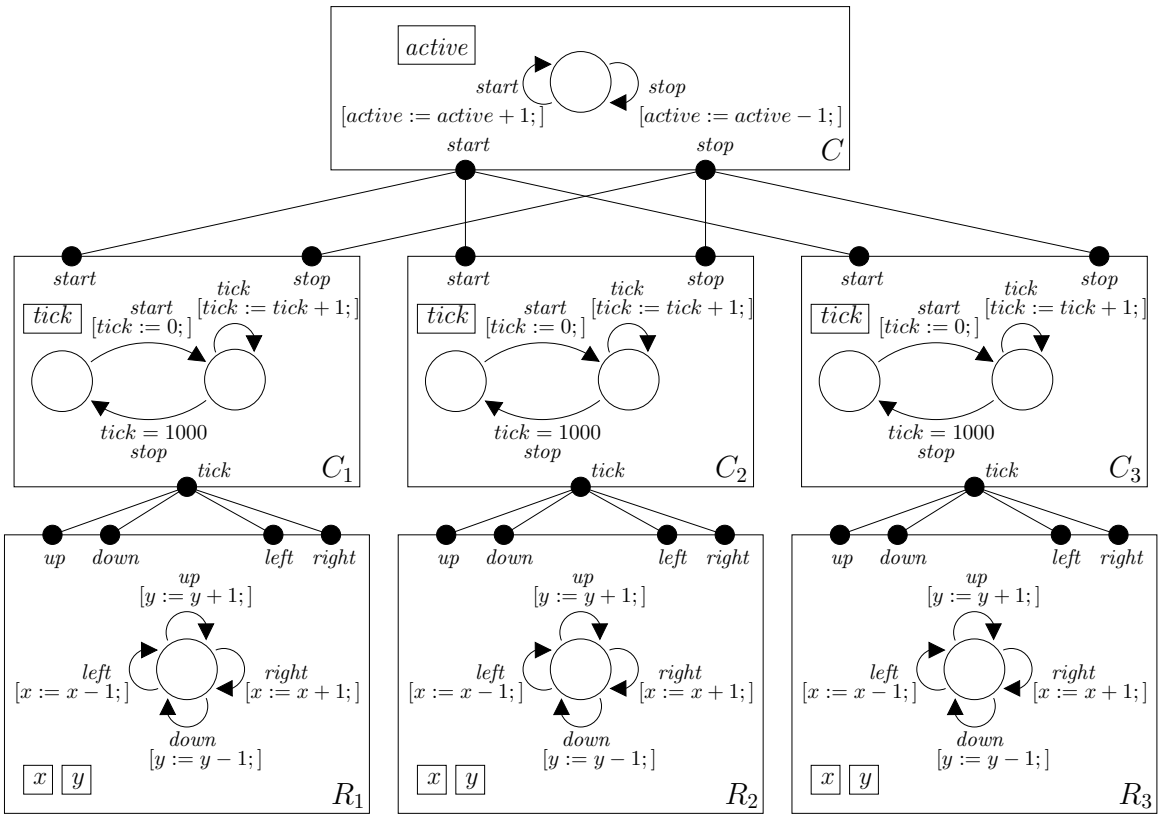


Fig. 12. Robotic application in BIP.

Table 2. Execution times (seconds) and numbers of rollbacks of the supervised robots.

Configuration / Size (n)	Supervised		Supervised-d		Supervised-o		Supervised-o-d	
	#Rollback	Time	#Rollback	Time	#Rollback	Time	#Rollback	Time
2	399998	345	267001	282	400280	224	266549	177
5	18039	129	16007	128	18022	82	15630	83
100	68	122	53	120	35	76	50	78

states of components. The focus of [29] is on controlling coordination properties (e.g., mutual exclusion) expressed on the states of autonomic managers of components, while the focus of this paper is on properties expressed on the internal states and interactions between components; and our monitors hence consider the effect of interactions when enforcing properties.

More generally, comparing supervisory control and our runtime enforcement approach, both controllers and our enforcement monitors let an undesired action happen before acting. However, enforcement monitors allow to roll the system back, reverting the effect of undesired actions, and allowing to explore alternative executions; hence providing a longer (correct) usage of the system. Note, while we did not illustrate it in this paper, our runtime enforcement monitors, can of course also be used to terminate the execution of the system upon the detection of an error. Moreover, supervisory control *needs* a model of the system under scrutiny, while our runtime enforcement approach could also be used with gray-box components where, for instance, only the interactions between component would be known in the model. Note, considering gray box components would entail important consequences on our framework. First, an obvious consequence is that the enforced properties can refer to atomic propositions that may be assessed at runtime; e.g., if only interactions are known, the property should not refer to component locations. Moreover, regarding the observation of the system to assess the property, instrumentation should rather be defined on interactions, and in BIP systems, not all variables are exported through the ports of interactions. Furthermore, one could not state any guarantee on the internal state of the components, and hence the simulation relation would not hold anymore.

8.4 Supervisory Approaches to Fault-tolerance

Some techniques are based on supervisory-control theory and controller synthesis [16]. Similarly to our approach, the objectives are to synthesize a mechanism that is maximally per-

missive and ensures fault-tolerance by disabling the controllable transitions that would either make the system diverge from the expected behavior or prevent it from reaching the expected behavior. In supervisory approaches the fault is due to a system action (cf. [43]). Faults are uncontrollable events and after their occurrence, the controller recovers the system within a finite number of steps. Moreover, the non-faulty part of the system needs to be both available and distinguishable from the system. Such approaches fall in the scope of our framework where monitors can enforce the non-occurrence of a particular action. Moreover, as BIP systems usually contain data, guards and assignments, it is generally not possible to statically compute the faulty behavior in the system.

8.5 Runtime Enforcement for Monolithic Systems

Enforceable properties. Several sets of enforceable properties were defined with their associated enforcement monitors [42, 33, 28]. Restrictions to the set of enforceable specifications stem from the fact that the considered specifications are over infinite executions sequences and enforcement monitors have to take consistent decisions for possible infinite continuations of their observation. As shown in [25], when considering specifications over finite sequences, all properties become enforceable. In this paper, we consider specifications over finite sequences but point out restrictions arising from the features of CBSs.

Enforcement paradigm. The runtime enforcement paradigm defined in this paper improves previous ones. Indeed, upon the detection of bad behaviors, in previous paradigms events are "accumulated" in a memory until a future event makes the property satisfied (in case of progress properties) or to halt the execution (in case of safety properties). The enforcement paradigm defined in this paper avoids the occurrence of faults by reverting the effect of events that lead to a deviation from the desired behaviors, restoring the system in a state before the fault occurrence.

8.6 Dynamic Techniques for CBSs

The approach in [20] defines FTPL, a customization of Linear Temporal Logic to specify the correctness of component reconfigurations in Fractal. Then, the approach in [21] verifies at runtime the correctness of architectures.

Independently, the approach in [26] introduces runtime verification for BIP systems with monitors for checking the conformance of the runtime behavior against linear-time properties. All these approaches allowed only the *detection of errors and not their correction* using recovery. As [26] is only concerned with (the simpler problem) of runtime verification, it considers all properties as monitorable. In this paper, we introduce a notion of enforceable properties specific to CBSs and parameterized by a number of tolerance steps. While the purpose of the transformations in [26] is to introduce a verification monitor and transmit snapshots of the system to it, the transformations in Sec. 6 additionally grant the (enforcement) monitor with primitives to backup the system state and control it. As seen in Sec. 6, to preserve the system consistency on a roll-back, not only the parts of the system involved with the property are instrumented but also the "connected" parts. Finally, the approach in this paper provides stronger correctness guarantees.

The approach in [14] proposes a protocol to enforce the correct reconfigurations of a component-based systems. Similarly to our approach, [14] shows how to enforces properties on CBSs. However, this approach fundamentally differs from ours because, similarly to [32], [14] targets architectural invariants (for instance "All started components have all their mandatory imports wired." as mentioned in [14] while we target behavioral correctness properties. Moreover, components in [14] are seen as black boxes, their semantics is not used for enforcement purposes. Moreover, the interaction model considers only binary-unidirectional connections. On the contrary, our approach considers and uses the internal behavior of components and their interactions. Our approach targets behavioral properties about the internal states of components. Note, the architecture of BIP component is statically defined and cannot evolve at runtime.

9 Conclusions and Future Work

9.1 Conclusions

This paper introduces an abstract runtime enforcement framework for component-based systems and presents an instantiation of this framework for components described in the Behavior Interaction Priority framework. Our approach considers an input system whose behavior may deviate from a desired specification.

We identify the set of *stutter-invariant safety properties* as enforceable on component-based systems. Restrictions on the set of enforceable specifications come from i) the number of steps the system is allowed to deviate from the specification (before being corrected) and ii) the constraints imposed by instrumentation. We introduce a series of formal transformations of a (non-monitored) system to integrate an enforcement monitor, using the oracle of the specification as input.

Although several verification tools (e.g., DFinder [7, 8]) exist to verify the correctness of a model w.r.t. specifications, a re-design of the model is required if the model does not satisfy the specifications. Moreover, those tools fail if the local computations on components are not visible (for instance when calling function from compiled libraries), which is the case in most of the complex models. However, enforcing properties with our approach just requires to specify them as an input. Moreover, our method works even if local computations on components are not visible.

As a result, runtime enforcement provides an interesting complementary validation method as the validity of the specification is generally either undecidable or leads to an intractable state-explosion problem. Experimental results

demonstrate the usefulness of the approach. Both benchmarks show that introducing runtime enforcement monitors with our method introduces a reasonable overhead while enforcing the good behavior of the monitored systems. Our benchmark on robots additionally shows that the adequacy of introducing a disabler (to complement an enforcement monitor) augments with the likelihood of error in the system (smaller maps in our experiments). When the system produces fewer errors (larger maps in our experiments), a disabler imposes an additional (reasonable) overhead on the system.

9.2 Perspectives

An assumption of this paper is that events are formed with state-based information (using e.g., the current location, values of variables). If the desired property refers to events involving event-based information (e.g., execution of a function, exchange of messages), adequate cancellation of events have to be also provided.

Another interesting problem is to consider more expressive properties (i.e., non-safety) such as k-step enforceable properties (with $k > 1$) to allow transactional behavior. It will entail to find an alternative instrumentation technique and avoid hard-coding the connections between the initial system and the monitor. We will consider more dynamic connections between components using the (recent) dynamic version of BIP [15], combined with a memorization mechanism to store the state-history of components.

Another avenue for future work related to expressiveness is to consider timed properties/automata [1] (where the physical time elapsing between events influences the satisfaction of the underlying property) and inspire from existing approaches to runtime enforcement of timed properties for monolithic systems [35,37,36], possibly with uncontrollable events [3,39].

We also plan to specialize the approach presented in this paper to security-oriented properties. For instance, we will inspire from runtime enforcement of opacity properties for monolithic systems [27] and apply it to secureBIP [41], a secured version of the BIP framework.

Moreover, we will work towards the decentralization of the enforcement monitor to allow them to take decisions alone. The expected benefit is to reduce communication in the system. For this purpose, we shall inspire from [6,23,19] which considers the problem of decentralizing verification monitors in monolithic systems, and also from [12] which distributes a centralized scheduler of components for a given distributed architecture.

Finally, we shall consider optimization techniques to further reduce the performance impact on the initial system. For this purpose, we consider using various static analysis on both the specification and the system to reduce the needed instrumentation.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *ICDCS 98: Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 436–443, 1998.
3. David A. Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zalinescu. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1):3, 2013.
4. Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.
5. Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
6. Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In *FM 2012: Proceedings of 18th International Symposium on Formal Methods*, pages 85–100, 2012.
7. Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.
8. Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2: Towards efficient correctness of incremental design. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2011.
9. Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.
10. Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR - Concurrency Theory, Proceedings of the 19th International Conference*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.
11. Borzoo Bonakdarpour, Marius Bozga, and Gregor Gößler. A theory of fault recovery for component-based models. In *SSS 2012: Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 7596 of *LNCS*, pages 314–328. Springer, 2012.
12. Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.
13. Tayeb Bouhadiba, Quentin Sabah, Gwenaël Delaval, and Éric Rutten. Synchronous control of reconfiguration in Fractal component-based systems: a case study. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 309–318. ACM, 2011.
14. Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In David Notkin,

Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 13–22. IEEE / ACM, 2013.

15. Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using Dy-BIP. In *SC: Proceedings of Conference on High Performance Computing Networking, Storage and Analysis*, volume 7306 of *LNCS*, pages 1–16. Springer, 2012.
16. Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag, Secaucus, NJ, USA, 2006.
17. Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, and Mohamad Jaber. Runtime enforcement for component-based systems. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1789–1796. ACM, 2015.
18. George Chatzieleftheriou, Borzoo Bonakdarpour, Scott A. Smolka, and Panagiotis Katsaros. Abstract model repair. In *NFM*, volume 7226 of *LNCS*, pages 341–355. Springer, 2012.
19. Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.
20. Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. Using temporal logic for dynamic reconfigurations of components. In *FACS 2010: Proceedings of the 7th International Symposium on Formal Aspects of Component Software*, volume 6921 of *LNCS*, pages 200–217. Springer, 2010.
21. Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. Runtime verification of temporal patterns for dynamic reconfigurations of components. In *FACS 2011: Proceedings of 8th International Symposium on the Formal Aspects of Component Software. Revised Selected Papers*, volume 7253 of *LNCS*, pages 115–132. Springer, 2011.
22. Yliès Falcone. You should better enforce than verify. In *RV*, volume 6418 of *LNCS*, pages 89–105. Springer, 2010.
23. Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In *Proceedings of Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014.*, volume 8461 of *LNCS*, pages 66–83, 2014.
24. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *RV 2009: Proceedings of the 9th International Workshop on Runtime Verification. Selected Papers*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.
25. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
26. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the BIP framework with formally proved sound and complete instrumentation. *SOSYM*, 2013.
27. Yliès Falcone and Hervé Marchand. Enforcement and validation (at runtime) of various notions of opacity. *Discrete Event Dynamic Systems*, 25(4):531–570, 2015.
28. Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.
29. Soguy Mak Karé Gueye, Noel De Palma, and Éric Rutten. Component-based autonomic managers for coordination control. In Rocco De Nicola and Christine Julien, editors, *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7890 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2013.
30. Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Éric Rutten, Jean-Philippe Diguet, and Guy Gogniat. Modeling and synthesis of a dynamic and partial reconfiguration controller. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 703–706. IEEE, 2012.
31. Klaus Havelund and Allen Goldberg. Verify your runs. In *VSTTE 2005: Proceedings of the First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments. Revised Selected Papers and Discussions*, pages 374–383, 2008.
32. Olga Kouchnarenko and Jean-François Weber. Adapting component-based systems at runtime via policies with temporal patterns. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, volume 8348 of *Lecture Notes in Computer Science*, pages 234–253. Springer, 2013.
33. Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, January 2009.
34. Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC 90:Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 377–410, 1990.
35. Ilaria Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electron. Notes Theor. Comput. Sci.*, 186:101–120, July 2007.
36. Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, and Hervé Marchand. Runtime enforcement of parametric timed properties with practical applications. In Jean-Jacques Lesage, Jean-Marc Faure, José E. R. Cury, and Bengt Lennartson, editors, *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014.*, pages 420–427. International Federation of Automatic Control, 2014.
37. Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena-Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014.
38. Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Proceedings of the 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.
39. Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, and Hervé Marchand. Enforcement of (timed) properties with uncontrollable events. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th Interna-*

tional Colloquium Cali, Colombia, October 29-31, 2015, Proceedings, volume 9399 of *Lecture Notes in Computer Science*, pages 542–560. Springer, 2015.
40. Runtime Verification. http://www.runtime-verification.org, 2001-2015.
41. Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven information flow security for component-based systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, volume 8415 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.
42. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.
43. Qin Wen, Ratnesh Kumar, Jing Huang, and Haifeng Liu. A framework for fault-tolerant control of discrete event systems. *IEEE Trans. Automat. Contr.*, 53(8):1839–1849, 2008.
44. Thomas Wilke. Classifying discrete temporal properties. In *STACS*, volume 1563 of *LNCS*, pages 32–46. Springer, 1999.

A On the Correctness and Behavior of the Supervised System

At an abstract level, the correctness of runtime enforcement of a property φ on a BIP system S stems from two facts regarding the behavior of the synthesized BIP enforcement monitor:

- the enforcement monitor correctly observes S (see Sec. A.1); and
- the enforcement monitor intervenes on S only when φ is violated and then restores S to the previous correct state (see Sec. A.3), otherwise it lets S execute normally.

Moreover, the observation and intervention of the monitor are done in such a way that the executions of S are preserved.

More precisely, from an input BIP system and a monitor, using the transformations in Sec. 6, we synthesize a system which runtime semantics is the composition between the initial system and the monitor, as described in Definition 16. Hence, the weak bisimulation and trace inclusion properties described at the abstract level in Sec. 5 apply to the transformed BIP system.

A.1 Correctness of the Observation [26]

Correctly observing the system behavior relies on our instrumentation technique and follows the same correctness arguments as in [26]. We do not reiterate the proof but briefly recall the main arguments. First, in the instrumented system, the value of variables are the same as in the original system. Instrumentation only modifies the system to transfer the values of variables to the monitor. Second, the chosen priority model ensures the consistency of the events fed to the monitor: events are sent each time the system performs a transition relevant to the property, and the order of events faithfully reflects the execution.

A.2 Proof of Proposition 3 (p. 10)

Proof. Let us consider the smallest relation $R \subseteq (\mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})) \times \mathrm{Sta}$ defined as follows: $((q_0, \theta^{\mathcal{O}}_{\mathrm{init}}), q_0) \in R$ and:

(a) $(r, s) \in R \implies (r', s') \in R$, whenever $s \xrightarrow{la}_{\mathrm{Trans}} s' \wedge r \xrightarrow{la}_{\mathrm{Mon}} r'$, or
(b) $(r, s) \in R \implies (r', s) \in R$, whenever $r \xrightarrow{\epsilon}_{\mathrm{Mon}} r'$.

The proof is a direct consequence of Definition 16 and the chosen definition of R. The three conditions of Proposition 3 hold for relation R.

- Condition *1.* holds because q_0 and $\theta^{\mathcal{O}}_{\mathrm{init}}$ are the initial states of respectively $\langle \mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans} \rangle$ and the monitor.
- Let us consider $(r, s) \in R, la \in \mathrm{Lab}$, and $r' \in \mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})$ such that $r \xrightarrow{la}_{\mathrm{Mon}} r'$. Let r and r' be $\langle q, \theta \rangle$ and $\langle q', \theta' \rangle$ for some $q, q' \in \mathrm{Sta}$ and $\theta, \theta' \in \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}$. According to the semantics rules in Definition 16, $r \xrightarrow{la}_{\mathrm{Mon}} r'$ is possible only at two conditions. The first case is when rule (1) applies, that is when L moves from q to q' by la and $\theta = \theta'$ (i.e., the instrumentation function does not produce an event), for some $la \in \mathrm{Lab}$. According to (a), we have $(\langle q', \theta \rangle, q') \in R$. Similarly, when rule (2) applies, we can find $(\langle q', \theta' \rangle, q') \in R$ where θ' is such that $\exists \theta_c \in \Theta^{\mathcal{O}}, \exists e \in \Sigma : \theta \xrightarrow{e}_{\mathcal{E}} \theta_c \wedge \theta_c \xrightarrow{com}_{\mathcal{E}} \theta'$, with $\theta' \in \Theta^{\mathcal{O}}$. Hence, condition *2.* holds.
- Let us consider $(r, s) \in R$ and $r' \in \mathrm{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})$ such that $r \xrightarrow{\epsilon}_{\mathrm{Mon}} r'$. Let r and r' be $\langle q, \theta \rangle$ and $\langle q', \theta' \rangle$ for some $q, q' \in \mathrm{Sta}$ and $\theta, \theta' \in \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}$. According to the semantics rules in Definition 16, $r \xrightarrow{\epsilon}_{\mathrm{Mon}} r'$ is possible only if $q \xrightarrow{la}_{\mathrm{Trans}} q' \wedge \theta \xrightarrow{e}_{\mathcal{E}} \theta_e \wedge \theta_e \in \overline{\Theta}^{\mathcal{O}} \wedge \theta_e \xrightarrow{\bar{e}}_{\mathcal{E}} \theta$, for some $la \in \mathrm{Lab}, e = \mathrm{inst}(la, q'), \theta_e \in \overline{\Theta}^{\mathcal{O}}$, and we have $r = r'$. Hence, condition *3.* holds.

A.3 Correctness of the Intervention

In the following, we focus on the correctness of the behavior of enforcement monitors. The correctness stems from the fact that we consider safety properties and that, as it was similarly expressed at an abstract level in Proposition 1, enforcement monitors roll-back the system by one step as soon as the system emits an event that violates the property.

Intuitively, the correctness proof of the transformations consists in showing that the supervised BIP system behaves in the same way as the composition of an abstract enforcement monitor with the LTS of the initial system. That is, the behavior of the supervised systems follows the semantics rules in Definition 16.

A.3.1 Preliminaries: Partitioning Interactions

Recall that a trace of length l of a BIP system $\langle B, Init \rangle$ whose runtime semantics is $\pi(C) = \langle Q, A, \longrightarrow_\pi \rangle$ is the sequence of alternating states/configurations and interactions $q^0 \cdot a_0 \cdot$

$q^1 \cdot a_1 \cdots a_{l-1} \cdot q^l$ such that: $q^0 = Init$, and, $\forall i \in [0, l-1] : q^i \in Q \wedge \exists a_i \in A : q^i \xrightarrow{a_i}_\pi q^{i+1}$.

According to the transformations defined in Sec. 6, a trace $q^0 \cdot a_0 \cdot q^1 \cdot a_1 \cdots a_{l-1} \cdot q^l$ of the monitored system C^{rec} satisfies the following property.

Lemma 1. *If $\mathcal{E}.p^m \in a^i$, then all other ports involved in a^i are p^m ports.*

Proof. The lemma holds by construction, according to Definitions 21 (p. 13) and 23 (p. 14).

Similar lemmas hold for $\mathcal{E}.p^c$ and $\mathcal{E}.p^r$. A consequence of Lemma 1 is that the interactions of the supervised system can be grouped into four categories: the initial, recovery, continue, and monitor interactions. Consequently, we denote by α_m (resp. α_c, α_r) any interaction involving $\mathcal{E}.p^m$ (resp. $\mathcal{E}.p^c$, $\mathcal{E}.p^r$).

Lemma 2. *Let us consider $i \in [1, m]$ s.t. $q^i \cdot a^i \cdot q^{i+1}$, then $\mathcal{E}.p^m \in a^i$ iff $q^{i+1} \cdot a^{i+1} \cdot q^{i+2}$ where $\{\mathcal{E}.p^r, \mathcal{E}.p^c\} \cap a^{i+1} \neq \emptyset$.*

Proof. First, according to Definitions 21 and 23 (p. 13 and 14), interactions α^c and α^r have more priority than the interactions of the initial BIP system. Second, according to Definition 17 (p. 11), any instrumented transition of an atomic component consists of two transitions (one for recovery and one for continue) just after a transition for interacting with the monitor (i.e., labeled with port p^m).

Lemma 2 states that, after an interaction with the monitor (a^i is an α_m interaction), only two kinds of interactions can happen: either a recovery or a continue interaction (i.e., a_{i+1} is an α_r or an α_c interaction).

A.3.2 Proof of Proposition 4 (p. 15)

Let us consider a trace $q^0 \cdot a_0 \cdot q^1 \cdot a_1 \cdots a_{l-1} \cdot q^l$ of the supervised system and the next step of the system after this trace which consists in performing an interaction a. We distinguish two cases according to whether a is connected to an instrumented transition (i.e., $a \in \text{rec\_i}$) or not.

1. If $a \notin \text{rec\_i}$, then the execution of a does not modify the variables that affect the satisfiability of the property. This stems from the following facts. First, according to Definitions 21 and 23 (p. 13 and 14 respectively), interaction α^m has more priority than the interactions of the initial BIP system. Second, according to Definition 17 (p. 11), any instrumented transition of an atomic component consists of its previous transition followed by a transition to interact with the monitor (i.e., labeled with port p^m). After instrumentation, such interaction and the following state are mapped to ϵ. Consequently the first rule in Definition 16 (p. 9) applies.
2. If $a \in \text{rec\_i}$, then a is followed by the execution of an α^m interaction (i.e., an interaction with the enforcement monitor). After instrumentation, such interaction and the following state (where the values of variables are sent through port $\mathcal{E}.p^m$ of the enforcement monitor) are mapped to an event $e \in \Sigma$ in Definition 16. Recall that, according to Lemma 2, an α_m interaction is followed by either an α_c or an α_r interaction. We distinguish two subcases:
   - The first sub-case is when a involves transitions that do not modify the variables of the property but at least one of these transitions has a port that is in an interaction modifying some variables of the property. Hence, e corresponds to the last emitted event in the trace. Because of stutter-invariance, the system keeps satisfying the property. Consequently, α_m is followed by an α_c interaction. This situation corresponds to rule number 2 in Definition 16.
   - The second sub-case is when a involves transitions that modify some variables of the property. We distinguish two subsub-cases.
     - The first subsub-case is when e brings the monitor to a good (with verdict $\top$) or currently good state (with verdict $\top_c$). Then, the system executes interaction α^c that moves the system to a next good state that is the same as in the original system. This situation is similar to the previous first sub-case and also corresponds to rule number 2 in Definition 16.
     - The second subsub-case is when e brings the monitor to a bad state (with verdict $\bot$). Then, the system executes interaction α^r that restores the values of the variables and brings the system back to its (correct) previous state. The execution of a recovery transition corresponds to $\overline{e}$ in rule number 3 in Definition 16.