Runtime Enforcement for Component-Based Systems
Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, Mohamad Jaber

To cite this version:
Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, Mohamad Jaber. Runtime Enforcement for Component-Based Systems. Symposium on Applied Computing - Software Verification and Testing, Apr 2015, Salamanque, Spain. 10.1145/2695664.2695879. hal-01248353

HAL Id: hal-01248353
https://inria.hal.science/hal-01248353

Submitted on 2 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright
Runtime Enforcement for Component-Based Systems

Hadil Charafeddine
American University of Beirut
hnc01@aub.edu.lb

Khalil El-Harake
American University of Beirut
kme07@aub.edu.lb

Yliès Falcone
Université Grenoble-Alpes
Ylies.Falcone@imag.fr

Mohamad Jaber
American University of Beirut
mj54@aub.edu.lb

ABSTRACT
We propose a theoretical runtime enforcement framework for component-based systems (CBS) 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, ii) safety properties are 1-step enforceable. Given an abstract enforcement monitor for some 1-step enforceable property, we formally instrument (at relevant locations) a system 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.

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. Interestingly, these techniques are complementary to each other. Both types of techniques 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.

We aim to marry software synthesis and dynamic analysis. While runtime verification complements model-checking, we propose runtime enforcement (cf. [21, 14, 19]) 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 without specifying how systems should be instrumented.

We target component-based systems (CBSs) expressed in the BIP [5, 2]. BIP allows to build complex systems by coordinating the behavior of a set of atomic components. Behavior is described with Labelled Transition Systems extended with data and C functions. Coordination between components is done with interactions and priorities between them. This layered architecture confers a strong expressiveness to BIP [6]. Moreover, BIP has a rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors of its atomic components.

This paper shows how to easily integrate correctness properties into a CBS (thus allowing separation of concerns). More specifically, the contributions of this paper are as follows:
- to propose runtime enforcement for CBSs to avoid runtime errors;
- to introduce a new runtime enforcement paradigm: previous enforcement approaches introduced monitors that can store “bad events” in their memory without the possibility of canceling these events (rolling the system back) to explore alternative executions (see Sec. 5 for related work): the runtime enforcement paradigm proposed in this paper prevents the occurrence of misbehaviors;
- to instrument CBSs to observe and minimally alter their behavior;
- to propose 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.

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 propose an efficient and complete instrumentation technique: the monitor receives all events of interest of the property without degrading the performance of the system.

Preliminaries and Notation. For two domains of elements $E$ and $F$, we note $[E \to F]$ (resp. $[E \to F]$) 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 $\{e \in E | f \in F\}$ or $\{e \in E | f \in F\}$ when clear from context. For two functions $v, v' \in [X \to Y]$ and $v' \in [X' \to Y']$, the substitution function $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as: $v/v'(x) =$
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.

On the right two connectors are depicted: Rendezvous (only the maximal interaction \{s, r_1, r_2, r_3\} is possible), Rendezvous Broadcast (all interactions containing trigger port s are possible).

**DEFINITION 4** (CONNECTOR). A connector \( \gamma \) is a tuple \( \langle \gamma, t, G, F \rangle \), where:
- \( \mathcal{P}_s = \{ p_i \mid p_i \in \mathcal{P}, \mathcal{I} \} \) s.t. \( \forall i \in \mathcal{I} : \mathcal{P}_s \cap \mathcal{B}_i = \{ p_i \} \).
- \( t \in \mathcal{P}_s \rightarrow \{ \text{true}, \text{false} \} \) s.t. \( t(p) = \text{true} \) if \( p \) is trigger (and false otherwise).
- \( G \) is an expression over variables in \( \cup_{i \in \mathcal{I}} \mathcal{P}_i \cdot \text{vars} \) (the guard),
- \( F \) is an update function defined over variables in \( \cup_{i \in \mathcal{I}} \mathcal{P}_i \cdot \text{vars} \).

**DEFINITION 5** (INTERACTION). A set of ports \( \alpha = \{ p_j \}_{j \in J} \subseteq \mathcal{P}_s \) for some \( J \subseteq I \) is an interaction of \( \gamma \) if either there exists \( j \in J \) s.t. \( p_j \) is trigger, or for all \( j \in J, p_j \) is synchron and \( \{ p_j \}_{j \in J} = \mathcal{P}_s \).

An interaction \( \alpha \) has a guard and two functions \( G_\alpha, F_\alpha \) obtained by projecting \( G \) and \( F \) on the variables of the ports involved in \( \alpha \). We denote by \( \mathcal{I}(\gamma) \) the set of interactions of \( \gamma \) and \( \mathcal{I}(\gamma_1 \cup \ldots \cup \gamma_n) \) by \( \mathcal{I}(\gamma_1, \ldots, \gamma_n) \). Synchronization through an interaction involves two steps: evaluating \( G_\alpha \), and applying update function \( F_\alpha \).

**DEFINITION 6** (COMPONENT). A composite component consists of a set of atomic components \( \{ B_i \}_{i \in I} \) and a set of connectors \( \Gamma \). The connection of the components in \( \{ B_i \}_{i \in I} \) using set of connectors \( \Gamma \) is denoted \( \Gamma \langle \{ B_i \}_{i \in I} \rangle \).

The composite component defined from atomic components \( \{ B_i \}_{i \in I} \) and a set of connectors \( \Gamma \) is noted \( \Gamma \langle \{ B_1, \ldots, B_n \} \rangle \).

**DEFINITION 7** (SEMANTICS OF COMPOSITE COMPONENTS). A state \( q \) of composite component \( \Gamma \langle \{ B_1, \ldots, B_n \} \rangle \) is an n-tuple \( \langle q_{1_1}, \ldots, q_{n_1} \rangle \) where \( q_i = \langle l_i, \nu_i, p_i \rangle \) is a state of \( B_i \). The semantics of \( \Gamma \langle \{ B_1, \ldots, B_n \} \rangle \) is an LTS \( (Q, A, \rightarrow, \gamma) \), where:
- \( Q = B_1 \cdot Q \times \ldots \times B_n \cdot Q \),
- \( A = \cup_{i \in I} \{ a \in \mathcal{I}(\gamma) \} \) is the set of all possible interactions,
- \( \rightarrow \) is the least set of transitions satisfying the following rule:

\[
\forall q \in \gamma : \mathcal{G}(\gamma) \quad \exists \mathcal{E} \in \mathcal{I}(\gamma) : a = \{ p_j \}_{j \in J} \quad \mathcal{G}(\nu(x)) \\
\forall i \in \mathcal{I} : q_i \rightarrow^{(q_i, \nu(x))} q_i' \quad \forall i \in \mathcal{I} \setminus J : q_i = q_i' \\
\rightarrow = \mathcal{G}(\gamma)
\]

\( X \) is the set of variables attached to the ports of \( \alpha \), \( \nu \) is the global valuation, and \( F_\alpha \) is the restriction of \( F \) to the variables of \( \alpha \).

Interaction \( \alpha \) can be fired, whenever all its ports are enabled and its guard \( \langle \mathcal{G}(\nu(x)) \rangle \) holds. Then, involved components evolve according to \( \alpha \) 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). Adding priority model \( \pi \) over \( \Gamma \langle \{ B_1, \ldots, B_n \} \rangle \) defines a new composite component \( \pi \Gamma \langle \{ B_1, \ldots, B_n \} \rangle \) noted \( \pi \gamma \) and whose behavior is defined by \( (Q, A, \rightarrow, \pi) \), where \( \rightarrow = \pi \gamma \) is the least set of transitions satisfying the following rule:

\[
\forall q \in \gamma : \mathcal{G}(\gamma) \quad \exists \mathcal{E} \in \mathcal{I}(\gamma) : a = \{ p_j \}_{j \in J} \quad \mathcal{G}(\nu(x)) \\
\forall i \in \mathcal{I} : q_i \rightarrow^{(q_i, \nu(x))} q_i' \quad \forall i \in \mathcal{I} \setminus J : q_i = q_i' \\
\rightarrow = \mathcal{G}(\gamma)
\]
\[
q \xrightarrow{a} q' \quad \neg(\exists a' \in A, \exists \alpha'' \in Q : a < a' \land q \xrightarrow{a'} q'') \\
q \xrightarrow{\alpha} q'
\]

Interaction \(a\) is enabled whenever \(a\) is maximal according to \(\pi\). In BIP, maximal progress is expressed at the level of connectors.

**Definition 9 (Maximal Progress).** Given a connector \(\gamma\) and a priority model \(\pi\), we have: \(\forall a, a' \in I(\gamma) : (a \subset a') \land (a' \not\subset a) \implies \exists a < a'\).

Finally, systems are composed of a component and an initial state.

**Definition 10 (System).** A system \(\langle B, \text{Init} \rangle\) consists of a component \(B\) and an initial state \(\text{Init} \in B_1 \times B_2 \times \ldots \times B_n, L\).

Hierarchical connectors [6]. Given a connector \(\gamma\) we denote by \(\gamma.\text{export}\) the exported port of connector \(\gamma\), 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 \(\gamma\) are possible, i.e., \(\{s_1, s_2, s_3, s_1r_2, s_1r_3, s_2r_3, s_1r_2s_3\}\).

3. ABSTRACT RUNTIME ENFORCEMENT FOR CBS

We introduce an abstract runtime enforcement framework specific to CBS that i) considers the specific instrumentation of CBS components involved in CBS interactions, and ii) hierarchizes enforceable properties.

**Preliminaries.** A property \(\Pi\) over \(\Sigma\) is a subset of \(\Sigma^*\). If a sequence \(\sigma\) belongs to \(\Pi\), we note it \(\Pi(\sigma)\). To evaluate sequences of system events against properties, we shall use truth-domain \(B_k\) containing truth-values true \((\top)\), false \((\bot)\), currently true \((\top_c)\), and currently false \((\bot_c)\) [3, 16]. Given a sequence \(\sigma \in \Sigma\) and \(\Pi \subseteq \Sigma^*\), the evaluation of \(\sigma\) against \(\Pi\) [16] is defined as:

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

We consider safety properties over \(\Sigma\) noted \(\text{Safety}(\Sigma)\). Note, for a safety property \(\Pi\), \(\text{codom}(\Pi)_{\Sigma}^B \subseteq \{\top, \top_c, \bot, \bot_c\}\).

3.1 Specifying Properties of CBS

Following [18], 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, 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 \(\mathcal{AP}\) be the set of atomic propositions defined with the following grammar (where \(s \in \{\leq, =\}\)):

- **Atom**: \(\text{cnpt}_1\text{.var}_1 \cdot \text{cnpt}_2\text{.var}_2 \cdot \text{cnpt}_3\text{.var}_3 \cdot \text{cnpt}_4\text{.var}_4\)
- **c threatening**: \(s \in \cup_{s \in \{1, 2\}} B_1, \text{vars}\)
- **a_val**: \(v \in \text{Data}\)
- **a_loc**: \(s \in \cup_{s \in \{1, 2\}} B_1, \text{locs}\)
- **a_port**: \(p \in \cup_{s \in \{1, 2\}} B_1, \text{ports}\)

We use \(\text{Prop} : \Sigma \rightarrow 2^{\mathcal{AP}}\) for the set of atomic propositions used in an event \(e \in \pi(C)\). For \(ap \in \text{Prop}(e)\), \(\text{used}(ap)\) is the sequence of pairs formed by the components and the variables (or locations or ports) used to define \(ap\). Expression \(\text{used}(ap)\) is defined using a pattern-matching:

- \(\text{used}(ap) = \text{match}(ap)\) with
  - \(\text{cnpt}_1\text{.var}_1 \cdot \text{cnpt}_2\text{.var}_2 \rightarrow (\text{cnpt}_1\text{.var}_1, \text{cnpt}_2\text{.var}_2)\)
  - \(\text{cnpt}\text{.var} \cdot \text{val} \rightarrow (\text{cnpt}, \text{var})\)
  - \(\text{cnpt}\text{.loc} \rightarrow (\text{cnpt}\text{.loc})\)
  - \(\text{cnpt}\text{.port} = \text{a}\text{.port} \rightarrow (\text{cnpt}\text{.port})\)

3.2 Enforceable Properties

\(k\)-step tolerance and stutter-invariance delineate enforceable properties. ¹

**Definition 11 (k-step Enforceability).** \(\Pi\) is \(k\)-step-enforceable, if \(\Phi(\sigma) \Rightarrow \Pi(\sigma)\) for all sequences \(\sigma : \Pi(\sigma) \Rightarrow \Pi(\sigma)\) for which there exists a sequence \(\sigma'\) (without the factor) that evaluates to \(\top, \top_c\), and all sequences \(\sigma' \cdot \sigma_p\) obtained by appending a prefix \(\sigma_p\) of \(\sigma\) to \(\sigma'\) evaluate to \(\top, \top_c\). The constant \(k\) additionally represents the maximal “roll-back distance”, i.e., the number of observational steps, an enforcement monitor can revert the system.

**Proposition 3.1 (Enforceable Properties).** There exists an hierarchy of enforceable properties where: (1) \(\forall k, k' \in \mathbb{N} : k \leq k' \implies Enf(k, \Sigma) \subseteq Enf(k', \Sigma)\), and (2) for regular properties, \(k\)-step enforceability is decidable.

In the following, we consider monitors that can memorize one state of the system and thus restore it up to one observational step.²

**Proposition 3.2.** All safety properties are 1-step-enforceable as per Definition 11: \(\text{Safety}(\Sigma) \subseteq Enf(1, \Sigma)\).

Safety properties are prefix-closed languages. Thus, when a monitor detects a deviation on one event, it is legitimate to intervene because all deviations from the normal behavior are definitive. Stutter-invariance. Stutter-invariance [23] 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 instrument all synchronized transitions (through a port/interaction). This is a consequence of BIP semantics. Note, even if an instrumented transition does not

1. Contrarily to other runtime enforcement frameworks such as [21, 20], we do not consider specifications over infinite sequences but finite sequences. It avoids dealing with enforceability issues due to the semantics of the specification formalism (over infinite sequences, see [17] for a detailed explanation). In that case, for monolithic systems, all properties are enforceable.
2. The complexity of the instrumentation depends on the number of steps one wants to be able to roll-back the system (see Sec. 4). Considering more than one steps is left for future work.
interfere with 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. The evaluation of the property should not changed.

**Definition 12** (Stutter-invariance [23]). Two sequences of events \(σ, σ′ \in Σ^∗\) are stutter-equivalent if there exist \(a_0, \ldots, a_k \in Σ\) for some \(k\) s.t. \(σ\) and \(σ′\) belong to the set defined by regular expression \(a_0^* \cdot a_1^* \cdots a_k^∗\). A property \(Π \subseteq Σ^∗\) is stutter-invariant, if for any stutter-equivalent sequences \(σ, σ′ \in Σ^∗\), we have \((σ \in Π \wedge σ′ \in Π)\) or \((σ \notin Π \wedge σ′ \notin Π)\).

Based on Proposition 3.2, we finally consider the set of stutter-invariant safety properties as the enforceable properties on CBSs.

### 3.3 Abstract Runtime Enforcement for CBS

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

**Definition 13** (Runtime oracle [17]). An oracle \(O\) is a tuple \((Θ^O, θ^O_{\text{init}}, Σ, →O, B_4, \text{verdict}^O)\). The finite set \(Θ^O\) denotes the control states and \(θ^O_{\text{init}} \in Θ^O\) is the initial state. The complete function \(→O: Θ^O \times Σ \rightarrow Θ^O\) is the transition function. In the following we abbreviate \(→O(θ, a) = θ′\) by \(θ → O a θ′\).

Function \(\text{verdict}^O: Θ^O \rightarrow B_4\) is an output function, producing verdicts (i.e., truth-values) in \(B_4\) from control states.

Runtime oracles are independent from any formalism used to generate them and are able to check any linear-time property [16].

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 performs an appraisal on the sequence read so far. For the formal semantics of the oracle and a formal definition of sequence checking, we refer to [16]. Figure 2 shows an example of a runtime oracle that observes \(e_1, e_2\), where \(e_1\) (resp. \(e_2\)) denotes that variable \(x\) in component \(\text{comp}_1\) is strictly positive (resp. strictly negative).

**Enforcement Monitor.** An enforcement monitor (EM) is a finite-state machine that forms a sequence of events from the program to one that evaluates on “good verdicts” of the oracle. 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. 4. Compared to the enforcement monitors proposed in the literature, 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 14** (Enforcement Monitor). The EM associated to runtime oracle \(O = (Θ^O, θ^O_{\text{init}}, Σ, →O, B_4, \text{verdict}^O)\) is a tuple \(Ε = (Θ^E, θ^E_{\text{init}}, Σ \cup \text{Trans}, →E)\) where:
- \(Θ^E \subseteq Θ^O \cup Σ\) with \(Θ^O = \{θ_e \mid e \in Σ \wedge θ \in Θ^O\}\) s.t. \(Θ^E\) is reachable from \(θ^O_{\text{init}}\) with \(→E\).
- \(Σ = \{σ \mid σ \in Σ\}\) is the set of cancellation events,
- \(→E\) is the transition function defined as \(→E = \{θ (θ, e, θ′) \in Σ \wedge \text{verdict}^O(θ') \in \{\top, \bot\}\} \cup \{θ (θ, e, θ'), (θ, π, θ) \mid \exists θ' \in Θ : (θ, e, θ') \in →E \wedge \text{verdict}^E(θ') = \bot\} \) follows the structure of a runtime oracle. For each transition \((θ, e, θ')\) leading to a “bad” location \(θ'\) (verdict\(^O\)(θ’) = ⊥), we add a transition \((θ, e, θ_c)\) leading to a fresh intermediate state \(θ_c\) and a transition \((θ_c, π, θ)\) back to state \(θ\) labelled by the corresponding cancellation event.

**Composition of a system with a monitor.** We define the composition of a system with an enforcement monitor. The composition assumes some instrumentation function \(\text{inst} : Σ \rightarrow Σ\) that generates the event in \(Σ\) that holds according to the information contained in a state and the event fired by the LTS.

**Definition 15** (Composing a system with a monitor). Given a system LTS \(L = (\text{Lab}, \text{Sta}, \text{Trans})\), and an enforcement monitor \(Ε = (Θ^O, θ^O_{\text{init}}, Σ \cup Σ′, →E)\) for a safety property where states in \(Θ^O\) are associated to currently good and good verdicts, the composition is the LTS \((\text{Lab}, \text{Sta} × (Θ^O \cup Σ′), \text{Mon})\), noted \(L \circ \text{inst} EM\), where the transition relation \(Mon \subseteq \text{Sta} \times (Θ^O \cup Σ′) \times \text{Lab} \times \text{Sta} × (Θ^O \cup Σ′)\) is defined by the two following semantics rules (where \(e = \text{inst}(q, la)\)):

\[
q \xrightarrow{\text{lab}} Mon (q', θ, la) \quad \text{and} \quad q \xrightarrow{\text{lab}} Trans (q', θ, e) \quad θ, la \in Σ\]

At runtime, an enforcement monitor executes in a lockstep manner with the system. When the system emits an event that is of interest for the enforcement monitor (i.e., an event \(\text{inst}(q, la)\) \(\notin Σ\)), the enforcement monitors lets the system execute without intervening (first semantics rule). When the system emits an event that leads to a currently-good or good location, the enforcement monitor simply follows the system (second semantics rule). When the system emits an event that leads to a bad location according to the oracle, the enforcement monitor executes a cancellation event. In the third semantics rule, state \(q'\) is called an unstable state: it is a state where the system never actually stays in because the enforcement monitor inserts immediately a cancellation event. During an execution when the enforcement monitor executes event \(π\), it 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.

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 \(\text{inst} : \text{Sta} \rightarrow Σ\). Any execution of the composition seen through the instrumentation function \(\text{inst}\) does not deviate from the property. Note that the initial LTS deviates by at most 1 event before being corrected.

**Proposition 3.3.** Given \(Π \in \text{Safety}(Σ)\), its enforcement monitor \(EM\), and a system LTS \(L\), we have:

∀σ ∈ runs\((L \circ \text{inst} EM)\) : \(\text{inst}(σ) \in Π \wedge σ \in \text{runs}(L)\).

The proposition states that the composed system produces only correct executions that belong to the initial LTS.

### 4. RUNTIME ENFORCEMENT FOR BIP

We integrate a runtime oracle \(O = (Θ^O, θ^O_{\text{init}}, Σ, →O, B_4, \text{verdict}^O)\) for some (enforceable) property into a BIP system \((π(Γ\{B_1, \ldots, B_n\}), \{b_0, \ldots, b_n\})\).
Some transformations are defined w.r.t. an atomic component \( B = \langle P, I, T, X, \{g_r\}_{r \in CT}, \{f_r\}_{r \in CT} \rangle \).

### 4.1 Analysis and Extraction of Information

For a property expressed over \( \Sigma(\Gamma\{(B_1, \ldots, B_n)\}): \)
- \( \text{mon}_{\text{vars}}(B_1) \) is the set of variables used in the property related to component \( B_1 \), formally \( \text{mon}_{\text{vars}}(B_1) = \{B_1, x \mid \exists c \in \Sigma, \exists a \in \text{Prop}(\cdot \, : \, (B_1, x) \in \text{used}(ap)) \} \),
- \( \text{occur} \) is the set of all monitored variables, formally \( \text{occur} \overset{\text{def}}{=} \bigcup_{i \in [1 \ldots n]} \text{mon}_{\text{vars}}(B_i) \).

For instance the property described by the runtime oracle in Figure 2, we have \( \text{mon}_{\text{vars}}(\text{comp1}) = \{\text{comp1}.x\} \).

### 4.2 Instrumenting Transitions

We should only instrument the transitions that may modify some monitored variables. We denote by \( \text{select}_{\text{trans}}(B) \) 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 location or to a port of a component \( B \), e.g., if \( B.\text{loc} = l_0 \) appears in the property, then all transitions of \( B \) should be instrumented.

For the component in Figure 1, \( \text{select}_{\text{trans}}(\text{comp1}) = \{(l, p, x > 0, [y := x + l], l')\} \) since variable \( x \) is attached to port \( p \) and \( \text{comp1}.x \in \text{mon}_{\text{vars}}(\text{comp1}) \).

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.

### 4.3 Instrumenting Atomic Components

Let \( T^{\text{inst}} = \text{rec}_{\text{trans}} \cap B.\text{trans} \) be the set of transitions that should be instrumented in \( B \) (noted \( T^{\text{inst}} \) when clear from context).

We create new temporary/recovery variables used to store the values of the variables that could be modified 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: \( \text{rec}_{\text{trans}}(T^{\text{inst}}) \overset{\text{def}}{=} \bigcup_\tau \tau.\text{vars} \cup \text{mon}_{\text{vars}}(\tau.\text{func}) \).

The enforcement monitor needs to observe the location or the port being executed, we create two new variables \( \text{port} \) and \( \text{loc} \) that store the name of the next location and the name of the port being executed.

We create three new ports: (1) \( p^m \) is used to send the values of monitored variables to the monitor; (2) \( p^r \) is used to receive a \( \text{continue} \) notification from the monitor; (3) \( p^c \) is used to receive a \( \text{recovery} \) notification from the monitor.

Finally, we split each of its instrumented transitions in \( T^{\text{inst}} \) according to Definition 16, and we create new locations accordingly.

### Definition 17 (Instrumenting an Atomic Component)

**Instrumentation function \( \text{inst} \)** transforms an input atomic component: \( \text{inst}(B) \) is: (1) \( B \) if \( B \not\in \text{rec}_{\text{comp}} \); (2) \( \{p_{\text{inst}}, I_{\text{inst}}, T_{\text{inst}}, X_{\text{inst}}, \{g_r\}_{r \in p_{\text{inst}}}, \{f_r\}_{r \in I_{\text{inst}}} \} \), otherwise, where:

\[
X_{\text{inst}} = X \cup \{v \mid B_1.v \in \text{mon}_{\text{vars}}(B_1) \} \cup \{x_{\text{tmp}} \mid x \in \text{rec}_{\text{vars}}(T^{\text{inst}})\}
\]

\( p_{\text{inst}} = P \cup \{p^m, \text{mon}_{\text{vars}}(B_1) \}, \{p^r, \emptyset\}, \{p^c, \emptyset\} \),

\( I_{\text{inst}} = L \cup \{l^m_{\text{inst}} \mid \tau \in T^{\text{inst}}\} \cup \{l^c_{\text{inst}} \mid \tau \in T^{\text{inst}}\} \),

\( T_{\text{inst}} = \langle T^{\text{inst}} \rangle \cup \bigcup_{r \in r_{\text{inst}}} \text{inst}_{\text{trans}}(\tau) \).

#### Example 1 (Instrumenting a Transition).

Figure 3 shows how the red transition in Figure 1 is instrumented. On recovery, we restore all the variables that are modified when executing that 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 \( \tau \in \text{select}_{\text{trans}}(B) \) implies the instrumentation of all transitions synchronizing with \( \tau \) through an interaction. We define \( \text{rec}_{\text{trans}} \) to be the set of all transitions that should be instrumented. We also define \( \text{rec}_{\text{comp}} \) to be the set of components that contain at least one instrumented transition, and \( \text{rec}_{\text{inter}} \) to be the set of connectors synchronizing on at least one instrumented transition. Formally:

\[
\text{rec}_{\text{trans}} = \bigcup_{\tau \in \text{inst}_{\text{trans}}(B)} \text{select}_{\text{trans}}(B_1) \cup \text{rec}_{\text{trans}} = \bigcup_\tau \tau.\text{trans} \cup \{\tau \mid \exists \gamma \in \Gamma, \exists k \in \text{rec}_{\text{trans}} : \{\gamma.\text{port}, \tau.\text{port} \} \subseteq P_k \};
\]

\[
\text{rec}_{\text{comp}} = \{B \mid B.\text{trans} \cap \text{rec}_{\text{trans}} \neq \emptyset\};
\]

\( \text{rec}_{\text{inter}} = \{a \mid \exists \gamma \in \Gamma, \exists \tau \in \text{rec}_{\text{trans}} : \tau.\text{port} \subseteq P_k \} \).

#### Example 2 (Instrumenting an Atomic Component).

Figure 3 shows the instrumentation of the atomic component in Figure 1. Note that only the transition in red is instrumented. Also, the variables attached to port \( p^m \) (i.e., only \( \text{comp1}.x \) in this example) are those extracted from the oracle (see Figure 2), i.e., monitored variables of that component. Moreover, the function of the recovery transition (i.e., labelled with with \( p^r \)) recovers the variables that could be modified, i.e., \( x, y, z \) since variables \( x \) and \( z \) are attached to port \( p \) and \( r \) is assigned on the transition.

Next, we consider an instrumented atomic component \( B_{\text{inst}} = \text{inst}(B) \). After instrumenting an atomic component, we must also

---

4Variables created by the transformations have fresh name w.r.t. existing variables of the input system.
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 18 (Backup injection).** We define the backup injection function \( inj \) that transforms an input (already instrumented) atomic component s.t. \( \text{inj}(B_{\text{mon}}) = B'_{\text{mon}} = \langle P_{\text{mon}}, L_{\text{mon}}, T_{\text{mon}}, X_{\text{mon}}, \{g_r\} \cup \{f_r\} \rangle \), where:

\[
T_{\text{rec}} = \{ (l, g, p, f; \{ x_{1}^{\text{imp}} \mapsto x_{1}; \ldots; x_{j}^{\text{imp}} \mapsto x_{j} \}) | \tau \} = \text{rec\_vars}(\{r^1 \in B_{\text{mon}} \mid r^1 \text{.arc} = \ell' \text{ and } \forall r^1 \text{.port} \in P' \})
\]

Next, we consider \( B_{\text{rec}} = \text{inj}(B_{\text{mon}}) \) with injected backup.

**Example 3 (Backup injection).** Figure 4 shows backup injection (in blue). Variables \( x \) and \( z \) are backed up on transitions entering \( l_0 \) because they are modified on two outgoing transitions.

### 4.4 Creating an Enforcement Monitor in BIP from an Oracle

We transform an oracle \( \mathcal{O} \) into a BIP enforcement monitor \( \mathcal{E} \) that mimics the behavior of the enforcement monitor associated to \( \mathcal{O} \). \( \mathcal{E} \) contains a copy of monitored variables and a backup of them. When the system executes an interaction that synchronizes at least one instrumented transition, it interacts with \( \mathcal{E} \) through port \( p^m \) and sends the new values of monitored variables. \( \mathcal{E} \) supports the system to continue or to recover, accordingly. In case of recovery (resp. continue), the monitored variables should be recovered (resp. backup).

**Definition 19 (Building an Enforcement Monitor).** From oracle \( \mathcal{O} \) we define enforcement monitor \( \mathcal{E} = \langle P, L, T, X, \{ g_r \} \cup \{ f_r \} \rangle \) as an atomic component:

- \( X = \{ x \in X_{\text{mon}}^{\text{imp}} \mid x \in \text{occur} \} \)
- \( P = \{ (p^m, \text{occur}), (p^r, \emptyset), (p^p, \emptyset) \} \)
- \( L = L^\top \cup L^w \) with \( L^\top = \{(q, q) \mid q \in \Theta^\top \land \text{verdict}^\top(q) \in \{\top, \bot\}\) and \( L^w = \{ q^m \mid q \in L^\top \} \)
- \( T = T^\top \cup T^w \cup T^c \)
- \( T^c = \{ (q^m, p^r, e, f^c, q') \mid q \rightarrow q' \land \text{verdict}^c(q') = \top \}
\]

where:

\[
\begin{align*}
T^m &= \{ (q^m, \text{true}, \varnothing), (q^m, q') \mid q \in L^\top \}; \\
T^* &= \{ (q^m, p^r, e, f^p, q') \mid q \rightarrow q' \land \text{verdict}^p(q') = \top \}
\end{align*}
\]

Then, we connect all continue ports \( \langle p^r \rangle \) of instrumented components, with a connector with trigger ports and connected hierarchically to port \( p^r \) of \( \mathcal{E} \). The ports of the hierarchical connector are synchron so that the synchronization between ports \( p^r \) of instrumented components requires port \( p^r \) of \( \mathcal{E} \) to be enabled. This is necessary because the instrumented components will be ready to execute both the continue and the recoverable ports based on the decision taken by \( \mathcal{E} \). Similarly, we connect the recoverable 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, \( \mathcal{E} \) notes all 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 20 (Integration - Spin Recovery).** The composite component is \( \pi_{\text{mon}}(\mathcal{E} ; \mathcal{B}_{\text{mon}}^{\prime} \ldots, \mathcal{B}_{\text{mon}}^{\prime n}) \) where:

1. \( \Gamma_{\text{mon}} = \Gamma \cup \{ \gamma^m_{\text{rec}} = \gamma^m_1, \ldots; \gamma^m_{\text{rec}} = \gamma^m_n \} \)
2. \( \gamma^m_{\text{rec}} = \langle P^m, t_{\text{rec}}^{\text{true}}, \mathcal{E}_m \rangle \)
3. \( t_{\text{rec}}^{\text{true}}(\mathcal{E}, p^m) = \text{false} \land \forall p \in P_{\text{mon}} \setminus \{ \mathcal{E}, p^m \} : t_{\text{rec}}^{\text{true}}(p) = \text{true} \)
4. \( \mathcal{E}_{\text{rec}} \), 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.

- \( \gamma^c_1 = \langle \gamma^c_1, t_{\gamma^c_1}^{\text{true}}, [], \rangle \)
- \( \gamma^c_2 = \langle \gamma^c_2, t_{\gamma^c_2}^{\text{true}}, [] \rangle \)
- \( \gamma^p_1 = \langle \gamma^p_1, \mathcal{E}, p^m \rangle \)
- \( \gamma^p_2 = \langle \gamma^p_2, \gamma^p_1, \mathcal{E}, p^m \rangle \)
work for fault recovery in CBSs [7] which assumes a fault-model yield observationaly equivalent executions. Moreover, our objective is to minimally alter the checking abstraction techniques to model repair. Our approach Recent efforts (e.g., [11]) aim at adapting model- repair. Model repair. Supervisory approaches to fault-tolerance. Some techniques are based on supervisory-control theory and controller synthesis [10]. Similar objectives are to synthesize a mechanism that is maximally permissive 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. [22]). 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 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.

Runtime enforcement for monolithic systems [21, 20, 19]. Several sets of enforceable properties were proposed with their associated enforcement monitors. Restrictions to the set of enforceable specifications stem from the fact that the considered specifications are over infinite executions sequences. As shown in [17], 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. Moreover, the runtime enforcement paradigm proposed in this paper improves previous ones. Indeed, upon the detection of bad behaviors, previous paradigms proposed to “accumulate events” 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 proposed in this paper, studied now for safety properties but stated generally for any property, prevents and 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. Dynamic techniques for CBSs. [12] proposes FTPL, a customization of Linear Temporal Logic to specify the correctness of component reconfigurations in Fractal. Then, [13] proposes to runtime verify the correctness of architectures. Independently, [18] proposes runtime verification for BIP systems with monitors for 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 [18] 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 CBS and parametrized by a notion of number of tolerance steps. While the purpose of the transformations in [18] is to introduce a monitor and transmit snapshots of the system to it, the transformations proposed in this paper additionally grant the monitor with primitives to backup the system state and control it. As seen in Sec. 4, 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, our approach provides stronger correctness guarantees.

6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

This paper introduces runtime enforcement for component-based systems described in the BIP 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 propose a series of formal transformations of a (non-monitored) system to integrate an enforcement monitor, using the oracle of the specification as input. 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.

6.2 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 [9], combined with a memoryization mechanism to store the state-history of components.

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 [4, 15] which considers the problem of decentralizing verification monitors in monolithic systems, and also from [8] 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.

7. REFERENCES