HAL
open science

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-01248353v1

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
CMPS, Beirut, Lebanon
hnc01@aub.edu.lb

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

Yliès Falcone
Université Grenoble-Alpes
LIG, Grenoble, France
Ylies.Falcone@imag.fr

Mohamad Jaber
American University of Beirut
CMPS, Beirut, Lebanon
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 Behavior, Interaction and Priority (BIP) framework [6, 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 executionsof 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 while not degrading the performance of the system.

Preliminaries and Notation. 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 $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as: $v/v'(x) =$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SAC-SVT 15, April 13 - 17 2015, Salamanca, Spain
ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695879.

$v'(x)$ if $x \in X'$ and $v(x)$ otherwise. A Labelled Transition System (LTS) L is a 3-tuple $\langle \mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans} \rangle$ where Lab (resp. Sta) is the set of labels (resp. states) and $\mathrm{Trans} \subseteq \mathrm{Sta} \times \mathrm{Lab} \times \mathrm{Sta}$ is the transition relation. The *runs* of LTS L, noted $\mathrm{runs}(L)$, are the finite sequences over Sta that can be obtained by starting from the initial state and following the transition relation.

2. BEHAVIOR INTERACTION PRIORITY

2.1 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 $\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$, 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. $l \xrightarrow{p}_T l'$ (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 $\mathrm{var}(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$. Given an atomic component B, $B.ports$ denotes the set of ports of B, $B.locs$ denotes its set of locations, etc. Figure 1 shows an atomic component.

DEFINITION 3 (SEMANTICS OF ATOMIC COMPONENTS). *The semantics of atomic component $\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$ 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 $v' = f_\tau(v/v_{p'})$.

2.2 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 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.

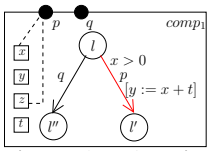
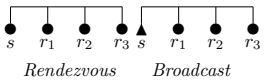


Figure 1: Atomic component

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



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

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:*

$$\frac{\begin{array}{c}\exists \gamma \in \Gamma : \gamma = \langle P_\gamma, t, G, F \rangle \quad \exists a \in \mathcal{I}(\gamma) : a = \{p_i\}_{i \in I} \quad G_a(v(X)) \\ \forall i \in I : \ q_i \xrightarrow{p_i(v_i)}_i q'_i \wedge v_i = F_{a_i}(v(X)) \qquad \forall i \notin I : \ q_i = q'_i\end{array}}{\langle q_1, \ldots, q_n \rangle \xrightarrow{a} \langle q'_1, \ldots, q'_n \rangle}$$

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). *Adding priority model π over $\Gamma(\{B_1, \ldots, B_n\})$ defines a new composite component $\pi\big(\Gamma(\{B_1, \ldots, B_n\})\big)$ noted $\pi(C)$ and whose behavior is defined by $\langle Q, A, \longrightarrow_\pi \rangle$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:*

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

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

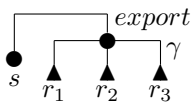
DEFINITION 9 (MAXIMAL PROGRESS). *Given a connector γ and a priority model π, we have: $\forall a, a' \in \mathcal{I}(\gamma)$: $(a \subset a') \wedge (a' \prec a \notin \pi) \implies a \prec a'$.*

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

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

Hierarchical connectors [6]. 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., $\{sr_1, sr_2, sr_3, sr_1r_2, sr_1r_3, sr_2r_3, sr_1r_2r_3\}$.



3. ABSTRACT RUNTIME ENFORCEMENT FOR CBS

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

Preliminaries. 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 truth-domain $\mathbb{B}_4$ 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 σ against Π [16] is defined as:

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

We consider *safety* properties over Σ noted $Safety(\Sigma)$. Note, for a safety property Π, $\mathrm{codom}([\![\cdot]\!]^{\Pi}_{\mathbb{B}_4}) \subseteq \{\bot, \top_c, \top\}$.

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 AP be the set of atomic propositions defined with the following grammar (where $* \in \{=, \leq\}$):

Atom	::=	$\text{cpnt}_1.\text{var}_1 * \text{cpnt}_2.\text{var}_2 \mid \text{cpnt.var} * \text{a_val}$
		$\mid \text{cpnt.loc} = \text{a_loc} \mid \text{cpnt.port} = \text{a_port}$
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	::=	$p \in \cup_{i\in[1,n]} B_i.ports$

We use $Prop : \Sigma \to 2^{AP}$ for the set of atomic propositions used in an event $e \in \pi(C)$. For $ap \in Prop(e)$, $\text{used}(ap)$ 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*:

```
used(ap)  =  match(ap) with
          |  cpnt1.var1 * cpnt2.var2 → (cpnt1, var1) · (cpnt2, var2)
          |  cpnt.var * val → (cpnt, var)
          |  cpnt.loc = a_loc → (cpnt, loc)
          |  cpnt.port = a_port → (cpnt, port)
```

3.2 Enforceable Properties

k-step tolerance and stutter-invariance delineate enforceable properties. [1]

k-step tolerance and enforceability. k-step tolerance models the maximal number of steps for which the system can deviate from the property and can still roll back. This might be due to the criticality of the system or the controlability endowed to our enforcement monitors on the system. Moreover, when a monitor intervenes, it should not destroy any (future) correct behavior and should determine that a deviation is definitive. In other words, on any execution sequence, if the last events made the property unsatisfied, then after some steps, on receiving an event the monitor should determine that there is no possible future behavior s.t. the execution again becomes correct. It is thus legitimate for the monitor to intervene.

DEFINITION 11 (k-STEP ENFORCEABILITY). *Π is k-step-enforceable, if $\max\{|\sigma| \mid \exists \sigma' \in \Sigma^* : [\![\sigma']\!]^{\Pi}_{\mathbb{B}_4} = \top_c \wedge \forall \sigma_p \preceq \sigma : [\![\sigma' \cdot \sigma_p]\!]^{\Pi}_{\mathbb{B}_4} = \bot_c\} < k$.*

The set of k-step enforceable properties is noted $Enf(k, \Sigma)$. $\Pi \subseteq \Sigma^*$ is k-step-enforceable, if the length of its maximal factor σ for which there exists a sequence σ' (without the factor) that evaluates to $\top_c$ and all sequences $\sigma' \cdot \sigma_p$ obtained by appending a prefix σ_p of σ to σ' evaluate to $\bot_c$. 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 a 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: $Safety(\Sigma) \subseteq Enf(1, \Sigma)$.*

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 to 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 change.

DEFINITION 12 (STUTTER-INVARIANCE [23]). *Two sequences of events $\sigma, \sigma' \in \Sigma^*$ are stutter-equivalent if there exist $a_0, \ldots, a_k \in \Sigma$ 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 $\Pi \subseteq \Sigma^*$ 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 $\sigma' \notin \Pi$).*

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

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 $\mathcal{O}$ is a tuple $\langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \mathbb{B}_4, \text{verdict}^{\mathcal{O}} \rangle$. 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 $\longrightarrow_{\mathcal{O}}: \Theta^{\mathcal{O}} \times \Sigma \to \Theta^{\mathcal{O}}$ is the transition function. In the following we abbreviate $\longrightarrow_{\mathcal{O}} (\theta, a) = \theta'$ by $\theta \xrightarrow{a}_{\mathcal{O}} \theta'$. Function $\text{verdict}^{\mathcal{O}} : \Theta^{\mathcal{O}} \to \mathbb{B}_4$ 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 [16].[3] 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. 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 $e_1^* \cdot e_2^*$, where e_1 (resp. e_2) denotes that variable x in component $comp_1$ is strictly positive (resp. strictly negative).

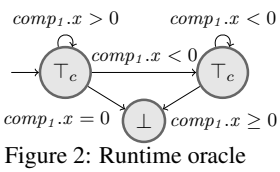


Figure 2: Runtime oracle

Enforcement Monitor. An enforcement monitor (EM) is a finite-state machine that transforms 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 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 $\mathcal{O} = \langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \mathbb{B}_4, \text{verdict}^{\mathcal{O}} \rangle$ is a tuple $\mathcal{E} = \langle \Theta^{\mathcal{E}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma \cup \overline{\Sigma}, \longrightarrow_{\mathcal{E}} \rangle$ where:*

- $\Theta^{\mathcal{E}} \subseteq \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}$ *with* $\overline{\Theta}^{\mathcal{O}} = \{\theta_e \mid e \in \Sigma \wedge \theta \in \Theta^{\mathcal{O}}\}$ *s.t.* $\Theta^{\mathcal{E}}$ *is reachable from* $\theta^{\mathcal{O}}_{\text{init}}$ *with* $\longrightarrow_{\mathcal{E}}$,
- $\overline{\Sigma} = \{\overline{e} \mid e \in \Sigma\}$ *is the set of* cancellation events,
- $\longrightarrow_{\mathcal{E}}$ *is the transition function defined as* $\longrightarrow_{\mathcal{E}} = \{\langle \theta, e, \theta' \rangle \in \longrightarrow_{\mathcal{O}} \mid \text{verdict}^{\mathcal{O}}(\theta') \in \{\top, \top_c\}\} \cup \{\langle \theta, e, \theta_e \rangle, \langle \theta_e, \overline{e}, \theta \rangle \mid \exists \theta' \in \Theta : \langle \theta, e, \theta' \rangle \in \longrightarrow_{\mathcal{O}} \wedge \text{verdict}^{\mathcal{O}}(\theta') = \bot\}$.

[3]The runtime oracle to be 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.

An EM follows the structure of a runtime oracle. For each transition $\langle \theta, e, \theta' \rangle$ leading to a "bad" location θ' ($\text{verdict}^{\mathcal{O}}(\theta') = \bot$), we add a transition $\langle \theta, e, \theta_e \rangle$ leading to a fresh intermediate state θ_e and a transition $\langle \theta_e, \overline{e}, \theta \rangle$ 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 $\text{inst} : \text{Sta} \to \Sigma$ 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 $L = \langle \text{Lab}, \text{Sta}, \text{Trans} \rangle$, and an enforcement monitor $\mathcal{E} = \left\langle \Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma \cup \overline{\Sigma}, \longrightarrow_{\mathcal{E}} \right\rangle$ for a safety property where states in $\Theta^{\mathcal{O}}$ are associated to currently good and good verdicts, the composition is the LTS $\left\langle \text{Lab}, \text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}), \text{Mon} \right\rangle$, noted $L \otimes_{\text{inst}} EM$, where the transition relation $\text{Mon} \subseteq \text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}}) \times \text{Lab} \times \text{Sta} \times (\Theta^{\mathcal{O}} \cup \overline{\Theta}^{\mathcal{O}})$ is defined by the two following semantics rules (where $e = \text{inst}(q, la)$):*

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

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

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

At runtime, an enforcement monitor executes in a lockstep manner with the system. When the system emits an event that is of interest for the enforcement monitor (i.e., an event $\text{inst}(q, la) \notin \Sigma$), 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 $\overline{e}$, 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 $\text{inst} : \text{Sta} \to \Sigma$. Any execution of the composition seen through the instrumentation function 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 $\Pi \in Safety(\Sigma)$, its enforcement monitor EM, and a system LTS L, we have:*
$\forall \sigma \in \text{runs}(L \otimes_{\text{inst}} EM) : \text{inst}(\sigma) \in \Pi \wedge \sigma \in \text{runs}(L)$.

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 $\mathcal{O} = \langle \Theta^{\mathcal{O}}, \theta^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \mathbb{B}_4, \text{verdict}^{\mathcal{O}} \rangle$ for some (enforceable) property into a BIP system $\left(\pi(\Gamma(\{B_1, \ldots, B_n\})), \langle l_0^1, \ldots, l_0^n \rangle\right)$.

Some transformations are defined w.r.t. an atomic component $B = \langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$.

4.1 Analysis and Extraction of Information

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

- $\text{mon_vars}(B_i)$ is the set of variables used in the property related to component B_i, formally $\text{mon_vars}(B_i) \stackrel{\text{def}}{=} \{B_i.x \mid \exists e \in \Sigma, \exists ap \in Prop(e) : (B_i, x) \in \text{used}(ap)\}$,
- occur is the set of all monitored variables, formally $\text{occur} \stackrel{\text{def}}{=} \bigcup_{i \in [1,n]} \text{mon_vars}(B_i)$.

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

4.2 Instrumenting Transitions

We should only instrument the transitions that may modify some monitored variables. We denote by $\text{select_trans}(B)$ 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 location or to a port of a component B (e.g., if $B.loc = l_0$ appears in the property), then all transitions of B should be instrumented. For the component in Figure 1, $\text{select_trans}(comp1) = \{(l, p, x > 0, [y := x + t], l')\}$ since variable x is attached to port p and $comp1.x \in \text{mon_vars}(comp1)$.

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, p^c are special, their purpose is detailed in Sec. 4.3.

DEFINITION 16 (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 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 $\tau \in \text{select_trans}(B_i)$ 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_inter 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) $\text{rec_inter} \stackrel{\text{def}}{=} \{a \in \Gamma \mid \exists \tau \in \text{rec_trans} : \tau.port \in P_\gamma\}$.

4.3 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 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: $\text{rec_vars}(T') \stackrel{\text{def}}{=} \bigcup_{\tau \in T'} \tau.port.vars \cup var(\tau.func)$. If the enforcement monitor needs to observe the location or the port being executed, we create two new variables[4] *port* and *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^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 16, and we create new locations accordingly.

DEFINITION 17 (INSTRUMENTING AN ATOMIC COMPONENT). *Instrumentation function* inst *transforms an input atomic component:* $\text{inst}(B)$ *is: (1) B if $B \notin \text{rec_comp}$; (2) $\langle P^{\text{inst}}, L^{\text{inst}}, T^{\text{inst}}, X^{\text{inst}}, \{g_\tau\}_{\tau \in T^{\text{inst}}}, \{f_\tau\}_{\tau \in T^{\text{inst}}} \rangle$, otherwise. 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,*
- *$P^{\text{inst}} = P \cup \{\langle p^m, \text{mon_vars}(B_i) \rangle, \langle p^c, \emptyset \rangle, \langle p^r, \emptyset \rangle\}$,*
- *$L^{\text{inst}} = L \cup \{l_\tau^m \mid \tau \in T^r\} \cup \{l_\tau^r \mid \tau \in T^r\}$,*
- *$T^{\text{inst}} = (T \setminus T^r) \cup (\bigcup_{\tau \in T^r} \text{inst_trans}(\tau))$.*

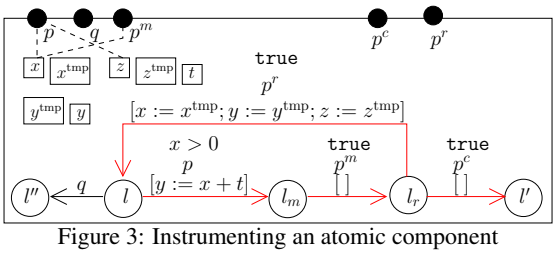


Figure 3: Instrumenting an atomic component

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 $comp1.x$ 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, and z since variables x and z are attached to port p and y 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

[4] Variables 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.* $\mathrm{inj}(B^{\mathrm{inst}}) = B^{\mathrm{rec}} = \langle P^{\mathrm{inst}}, L^{\mathrm{inst}}, T^{\mathrm{rec}}, X^{\mathrm{inst}}, \{g_\tau\}_{\tau \in T^{\mathrm{rec}}}, \{f_\tau\}_{\tau \in T^{\mathrm{rec}}} \rangle$, *where:*

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

Next, we consider $B^{\mathrm{rec}} = \mathrm{inj}(B^{\mathrm{inst}})$ 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. $\mathcal{E}$ notifies the system to continue or to recover, accordingly. In case of recovery (resp. continue), the monitored variables should be recovered (resp. backup) .

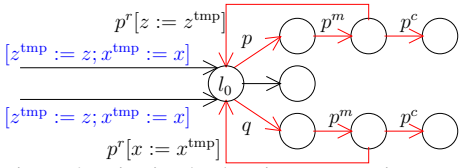


Figure 4: Injecting backup into an atomic component.

DEFINITION 19 (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:*

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

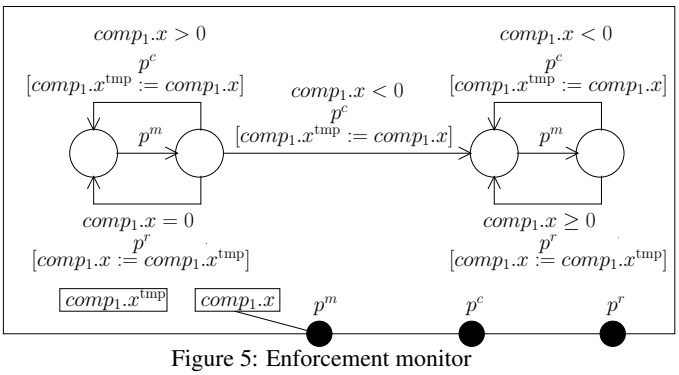


Figure 5: Enforcement monitor

EXAMPLE 4 (BUILDING AN ENFORCEMENT MONITOR). *Figure 5 depicts the enforcement monitor in BIP generated from the runtime oracle in Figure 2. 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).*

4.5 Integration

We define the connection between the instrumented atomic components $\pi(\Gamma(\{B_1^{\mathrm{rec}}, \ldots, B_n^{\mathrm{rec}}\}))$ 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 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, $\mathcal{E}$ 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 20 (INTEGRATION - SPIN RECOVERY). *The composite component is* $\pi^{\mathrm{rec}}(\Gamma^{\mathrm{rec}}(B_1^{\mathrm{rec}}, \ldots, B_n^{\mathrm{rec}}, \mathcal{E}))$, *where:*

- $\Gamma^{\mathrm{rec}} = \Gamma \cup \{\gamma^m, \gamma^{c_1}, \gamma^{c_2}, \gamma^{r_1}, \gamma^{r_2}\}$, *where:*
  - $\gamma^m = \langle P_{\gamma^m}, t_{\gamma^m}, \mathtt{true}, F_{\gamma^m} \rangle$, *where:*
    - $P_{\gamma^m} = \{\langle B_i.p^m, \mathrm{mon_vars}(B_i) \rangle\}_{B_i \in \mathrm{rec_comp}} \cup \{\mathcal{E}.p^m\}$, $t_{\gamma^m}(\mathcal{E}.p^m) = \mathtt{false}$ *and* $\forall p \in P_{\gamma^m} \setminus \{\mathcal{E}.p^m\} : t_{\gamma^m}(p) = \mathtt{true}$,
    - 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.*
  - $\gamma^{c_1} = \langle P_{\gamma^{c_1}}, t_{\gamma^{c_1}}, \mathtt{true}, [\,] \rangle$, $\gamma^{c_2} = \langle P_{\gamma^{c_2}}, t_{\gamma^{c_2}}, \mathtt{true}, [\,] \rangle$, *where:*
    - $P_{\gamma^{c_1}} = \{\langle B_i.p^c, \emptyset \rangle\}_{B_i \in \mathrm{rec_comp}}$ *and* $\forall p \in P_{\gamma^{c_1}} : t_{\gamma^{c_1}}(p) = \mathtt{true}$,
    - $P_{\gamma^{c_2}} = \{\gamma^{c_1}.export, \mathcal{E}.p^c\}$ *and* $t_{\gamma^{c_2}}(\gamma^{c_1}.export) = t_{\gamma^{c_2}}(\mathcal{E}.p^c) = \mathtt{false}$.
  - $\gamma^{r_1} = \langle P_{\gamma^{r_1}}, t_{\gamma^{r_1}}, \mathtt{true}, [\,] \rangle, \gamma^{r_2} = \langle P_{\gamma^{r_2}}, t_{\gamma^{r_2}}, \mathtt{true}, [\,] \rangle$, *where:*

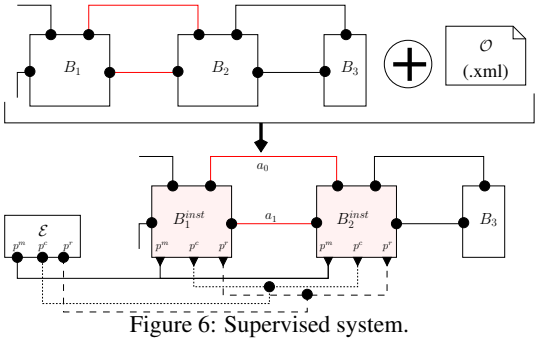


Figure 6: Supervised system.

- $P_{\gamma^{r_1}} = \{\langle B_i.p^r, \emptyset\rangle\}_{B_i \in \text{rec_comp}}$ *and* $\forall p \in P_{\gamma^{r_1}}$: $t_{\gamma^{r_1}}(B_i.p^r) = \texttt{true}$,
- $P_{\gamma^{r_2}} = \{\gamma^{r_1}.export, \mathcal{E}.p^r\}$ *and* $t_{\gamma^{r_2}}(\gamma^{r_1}.export) = t_{\gamma^{r_2}}(\mathcal{E}.p^r) = \texttt{false}$,

- $\pi^{\text{rec}} = \pi \cup \{\langle a, a'\rangle \mid a \in \cup_{\gamma \in \text{rec_inter}} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma^m, \gamma^{c_1}, \gamma^{c_2}, \gamma^{r_1}, \gamma^{r_2})\}$.

Note that, the instrumented system defined above 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.

EXAMPLE 5 (INTEGRATION). *Figure 6 shows the supervised system with $\mathcal{E}$. We assume that the monitored variables are modified only when executing interactions a_0 and a_1. Consequently, component B_3 remains unchanged.*

4.6 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 3.3). 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 15.

PROPOSITION 4.1. *Consider the mapping described above, Proposition 3.3 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.*

5. RELATED WORK

Model repair. Recent efforts (e.g., [11]) aim at adapting model-checking abstraction techniques to model repair. Our approach fundamentally differs from model repair. 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. Moreover, our objective is to minimally alter the initial behavior of the system. Correct executions are preserved and yield observationaly equivalent executions.

Theories of fault-tolerance [1]. Close to our approach is a framework for fault recovery in CBSs [7] which assumes a fault-model as input (i.e., a labelling 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, [7] takes as input fault-tolerant programs and assumes fault-tolerance being encoded inside the target program. In [1], the system is seen as a collection of guarded commands. In [7], fault detection and recovery span across multiple components. Both approaches fall short in meeting the modularity requirement of CBSs. Indeed, programs in [1] do not have their own state-space. The fault models considered in [7] assumes fault detection and recovery to concern several components with inter-dependent interactions.

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 memorization 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

[1] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *ICDCS*, pages 436–443, 1998.

[2] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

[3] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[4] A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proceedings of FM 2012: Formal Methods - 18th International Symposium*, pages 85–100, 2012.

[5] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.

[6] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[7] B. Bonakdarpour, M. Bozga, and G. Gößler. A theory of fault recovery for component-based models. In *SSS*, volume 7596 of *LNCS*, pages 314–328. Springer, 2012.

[8] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

[9] M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling dynamic architectures using Dy-BIP. In *SC*, volume 7306 of *LNCS*, pages 1–16. Springer, 2012.

[10] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[11] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros. Abstract model repair. In *NFM*, volume 7226 of *LNCS*, pages 341–355. Springer, 2012.

[12] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In *FACS 2010*, volume 6921 of *LNCS*, pages 200–217. Springer, 2010.

[13] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Runtime verification of temporal patterns for dynamic reconfigurations of components. In *FACS*, volume 7253 of *LNCS*, pages 115–132. Springer, 2011.

[14] Y. Falcone. You should better enforce than verify. In *RV*, volume 6418 of *LNCS*, pages 89–105. Springer, 2010.

[15] Y. Falcone, T. Cornebize, and J. 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.

[16] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *RV*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.

[17] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[18] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems in the BIP framework with formally proved sound and complete instrumentation. *SOSYM*, 2013. To appear. Available online.

[19] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *FMSD*, 38(3):223–262, 2011.

[20] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, Jan. 2009.

[21] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[22] Q. Wen, R. Kumar, J. Huang, and H. Liu. A framework for fault-tolerant control of discrete event systems. *IEEE Trans. Automat. Contr.*, 53(8):1839–1849, 2008.

[23] T. Wilke. Classifying discrete temporal properties. In *STACS*, volume 1563 of *LNCS*, pages 32–46. Springer, 1999.