# Runtime Verification of Temporal Patterns for Dynamic Reconfigurations of Components

Julien Dormoy, Olga Kouchnarenko, Arnaud Lanoix

# Runtime Verification of Temporal Patterns for Dynamic Reconfigurations of Components

Julien Dormoy[1], Olga Kouchnarenko[1], and Arnaud Lanoix[2]

[1] University of Franche-Comté, Besançon, France
{jdormoy,okouchnarenko}@lifc.univ-fcomte.fr
[2] Nantes University, Nantes, France
arnaud.lanoix@univ-nantes.fr

**Abstract.** Dynamic reconfigurations increase the availability and the reliability of component-based systems by allowing their architectures to evolve at runtime. Recently we have proposed a temporal pattern logic, called FTPL, to characterize the correct reconfigurations of component-based systems under some temporal and architectural constraints.
As component-based architectures evolve at runtime, there is a need to check these FTPL constraints on the fly, even if only a partial information is expected. Firstly, given a generic component-based model, we review FTPL from a runtime verification point of view. To this end we introduce a new four-valued logic, called RV-FTPL (Runtime Verification for FTPL), characterizing the "potential" (un)satisfiability of the architectural constraints in addition to the basic FTPL semantics. Potential true and potential false values are chosen whenever an observed behaviour has not yet lead to a violation or satisfiability of the property under consideration. Secondly, we present a prototype developed to check at runtime the satisfiability of RV-FTPL formulas when reconfiguring a Fractal component-based system. The feasability of a runtime property enforcement is also shown. It consists in supervising on the fly the reconfiguration execution against desired RV-FTPL properties. The main contributions are illustrated on the example of a HTTP server architecture.

## 1 Introduction

This paper deals with the formal specification and verification of dynamic reconfigurations of component-based systems at runtime. Dynamic reconfigurations increase the availability and the reliability of those systems by allowing their architectures to evolve at runtime.

Dynamic reconfiguration of distributed applications is an active research topic [1,2,21] motivated by practical distributed applications like, e.g., those in Fractal [10] or OSGi[3]. In many recent works, the idea of using temporal logics to manage applications at runtime has been explored [6,18,8,14].

[3] http://www.osgi.org

In [14], we have proposed a temporal pattern logic, called FTPL, to characterize the correct reconfigurations of component-based systems under some temporal and architectural constraints **(1)**. We have also explained in [19], how to reuse a generic formal model to check the component-based model consistency through reconfigurations, and to ensure that dynamic reconfigurations satisfy architectural and integrity constraints, invariants, and also temporal constraints over (re)configuration sequences **(2)**.

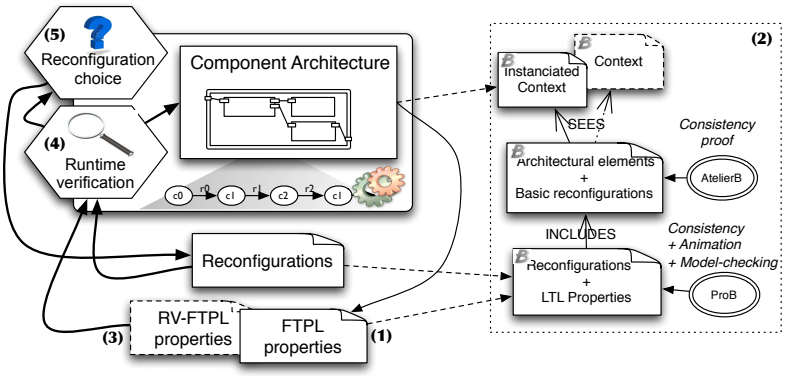


**Fig. 1.** Principle and contributions

As component-based architectures evolve at runtime, there is a need to evaluate the FTPL constraints on the fly, even if only a partial information can be expected. Indeed, an FTPL property often cannot be evaluated to true or false during the system execution. In addition, the reconfigurations change the validity of FTPL constraints by modifying the component architecture. In this paper, given a generic component-based model, we review FTPL from a runtime verification point of view **(3)**. To this end we introduce a new four-valued logic, called RV-FTPL (Runtime Verification for FTPL), characterizing the "potential" (un)satisfiability of the architectural constraints in addition to the basic FTPL semantics. Like in RV-LTL [8], potential true and potential false values are chosen whenever an observed behaviour has not yet lead to a violation or acceptance of the property under consideration.

We then integrate the runtime verification of temporal patterns into the Fractal component model [10]. More precisely, we describe a prototype developed to check at runtime—by reusing the FPath and FScript [12] tool supports—the satisfiability of RV-FTPL formulas. This verification is performed when reconfigurating a component-based system **(4)**. More, the feasability of a runtime property enforcement is also shown. It consists in supervising at runtime the reconfiguration execution in order to ensure that the RV-FTPL property of interest is fulfilled **(5)**: our 4-valued logic can help in guiding the reconfiguration process,

namely in choosing the next reconfiguration operations to be applied. The main contributions are illustrated on the example of a HTTP server architecture.

The remainder of the paper is organised as follows. After introducing a motivating example in Sect. 2, we briefly recall, in Sects. 3 and 4, the considered architectural (re-)configuration model and the FTPL syntax and semantics. We then define in Sect. 5 the runtime verification of FTPL (RV-FTPL) refining FTPL semantics with potential true and potential false values. Section 6 describes a prototype implementing the RV-FTPL verification, and its integration into the Fractal framework. Section 7 explains how to enforce, at runtime, Fractal component system reconfigurations against desired RV-FTPL properties. Finally, Section 8 concludes before discussing related work.

## 2 Motivating Example

To motivate and to illustrate our approach, let us consider an example of an HTTP server from [11]. The architecture of this server is displayed in Fig. 2.

The **RequestReceiver** component reads HTTP requests from the network and transmits them to the **RequestHandler** component. In order to keep the response time as short as possible, **RequestHandler** can either use a cache (with the component **CacheHandler**) or directly transmit the request to the **RequestDispatcher** component. The number of requests (load) and the percentage of similar requests (deviation) are two parameters defined for the **RequestHandler** component:

- The **CacheHandler** component is used only if the number of similar HTTP requests is high.
- The memorySize for the **CacheHandler** component must depend on the overall load of the server.
- The validityDuration of data in the cache must also depend on the overall load of the server.
- The number of used file servers (like the **FileServer1** and **FileServer2** components) used by **RequestDispatcher** depends on the overall load of the server.

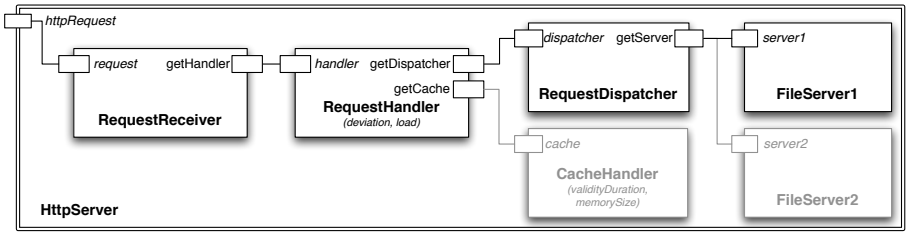


**Fig. 2.** HTTP Server architecture

We consider that the HTTP server can be reconfigured during the execution by the following reconfiguration operations:

1. AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the **CacheHandler** component when the deviation value increased/decreased around 50;
2. AddFileServer and removeFileServer which are respectively used to add and remove the **FileServer2** component;
3. MemorySizeUp and MemorySizeDown which are respectively used to increase and to decrease the MemorySize value;
4. DurationValidityUp and DurationValidityDown to respectively increase and decrease the ValidityDuration value.

As an illustration, we specify the AddCacheHandler reconfiguration expressed in the FScript language [12]. When the deviation value exceeds 50, the reconfiguration consists in instantiating a **CacheHandler** component. Then, the component is integrated into the architecture, and the binding with the required interface of **RequestHandler** is established. Finally, the component **CacheHandler** is started.

```
1 action AddCacheHandler(root)
2   newCache = new("CacheHandler");
3   add($root, $newCache);
4   bind($root/child::RequestHandler/interface::getcache, $newCache/
        interface::cache);
5   start($newCache);
```

## 3 Architectural (Re-)Configuration Model

This section recalls the generic model for component-based architectures given in [14] and inspired by the model in [20,21] for Fractal. Both models are graphs allowing one to represent component-based architectures and reconfiguration operations and to reason about them.

Component-based models must provide mechanisms for systems to be dynamically adapted—through their reconfigurations—to their environments during their lifetime. These dynamic reconfigurations may happen because of architectural modifications specified in primitive operations. Notice that reconfigurations are not the only manner to make an architecture evolve. The normal running of different components also changes the architecture by modifying parameter values or stopping components, for instance.

### 3.1 Component-based architectures

In general, the system configuration is the specific definition of the elements that define or prescribe what a system is composed of. The architectural elements we consider (components, interfaces and parameters) are the core entities of a component-based system and relations over them to express various links between these basic architectural elements. We consider a graph-based representation illustrated by Fig. 3.

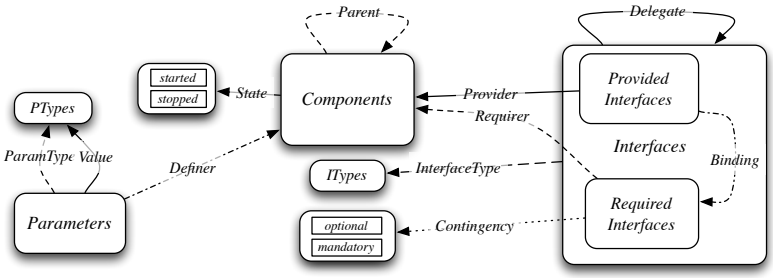


**Fig. 3.** Architectural elements and relations between them

In our model, a configuration $c$ is a tuple $\langle Elem, Rel \rangle$ where $Elem$ is a set of architectural elements, and $Rel \subseteq Elem \times Elem$ is a relation over architectural elements.

The architectural elements of $Elem$ are the core entities of a component-based system:

- $Components$ is a non-empty set of the core entities, i.e components;
- $RequiredInterfaces$ and $ProvidedInterfaces$ are defined to be subsets of $Interfaces$. Their union is disjunctive;
- $Parameters$ is a set of component parameters.

The architectural relation $Rel$ then expresses various links between the previously mentioned architectural elements.

- $InterfaceType$ is a total function that associates a type with each required and provided interface;
- $Provider$ is a total surjective function which gives the component having at least a provided interface, whereas $Requirer$ is only a total function;
- $Contingency$ is a total function which indicates for each required interface if it is *mandatory* or *optional*;
- $Definer$ is a total function which gives the component of a considered parameter;
- $Parent$ is a partial function linking sub-components to the corresponding composite component. Composite components have no parameter, and a sub-component must not be a composite including its parent component, and so on;
- $Binding$ is a partial function which connects together a provided interface and a required one: a provided interface can be linked to only one required interface, whereas a required interface can be the target of more than one provided interface. Moreover, two linked interfaces do not belong to the same component, but their corresponding instantiated components are sub-components of the same composite component. The considered interfaces must have the same interface type, and they have not yet been involved in a delegation;

– *Delegate* expresses delegation links. It is a partial bijection which associates a provided (resp. required) interface of a sub-component with a provided (resp. required) interface of its parent. Both interfaces must have the same type, and they have not yet been involved in a binding;
– *State* is a total function which associates a value from $\{started, stopped\}$ with each instantiated component: a component can be *started* only if all its mandatory required interfaces are bound or delegated;
– Last, $Value$ is a total function which gives the current value of a considered parameter.

Complete and formal definitions can be found in [19].

*Example 1.* Figure 4 gives a graph-based representation of the example from Sect. 2. In this figure, the architectural elements are depicted as boxes and circles, whereas architectural relations are represented by arrows.

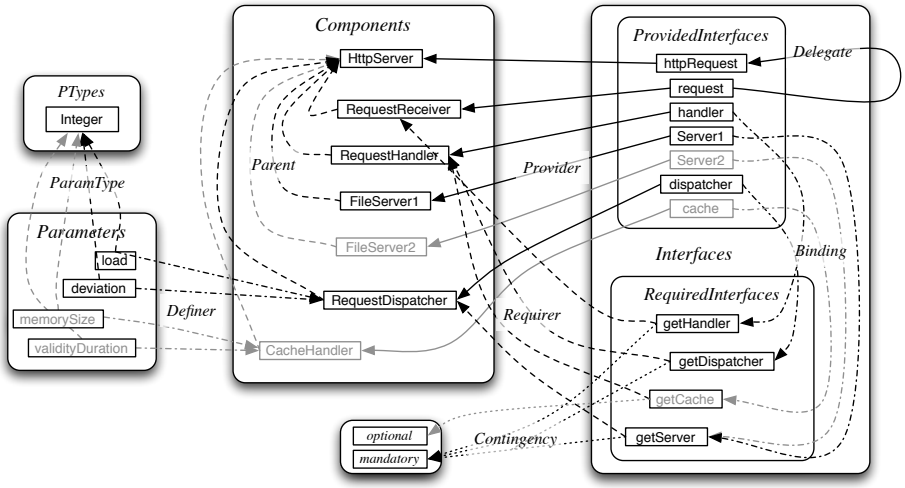


**Fig. 4.** Graph-based representation of the HTTP Server example

### 3.2 Dynamicity of Component Architectures

To support system evolution, some component models provide mechanisms to dynamically reconfigure the component-based architecture, during their execution. These dynamic reconfigurations are then based on architectural modifications, among the following primitive operations:

– instantiation/destruction of components;
– addition/removal of components;

– binding/unbinding of component interfaces;
– starting/stopping components;
– setting parameter values of components;

or combinations of them. A component architecture may also evolve by modifying parameter values or stopping components, like in the example.

Considering the component-based architecture model recalled in Sect. 3.1, a reconfiguration action is modelled by a graph transformation operation adding or removing nodes and/or arcs in the graph of the configuration. An evolution operation *op* transforms a configuration $c = \langle Elem, Rel \rangle$ into another one $c' = \langle Elem', Rel' \rangle$. It is represented by a transition from $c$ to $c'$, noticed $c \stackrel{op}{\rightarrow} c'$. Among the evolution operations (running operations and reconfigurations), we particularly focus on the reconfiguration ones, which are either the above-mentioned primitive architectural operations or their compositions. The remaining running operations are all represented by a generic operation, called the *run* operation; it is also the case for sequences of running operations.

The evolution of a component architecture is defined by the transition system $\langle \mathcal{C}, \mathcal{R}_{run}, \rightarrow \rangle$ where:

– $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ is a set of configurations;
– $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ is a finite set of evolution operations;
– $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ is the reconfiguration relation.

Given the model $M = \langle \mathcal{C}, \mathcal{R}_{run}, \rightarrow \rangle$, an evolution path (or a path for short) $\sigma$ of $M$ is a (possibly infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0. \exists r_i \in \mathcal{R}_{run}. c_i \stackrel{r_i}{\rightarrow} c_{i+1} \in \rightarrow))$.

We use $\sigma(i)$ to denote the $i$-th configuration of a path $\sigma$. The notation $\sigma_i$ denotes the suffix path $\sigma(i), \sigma(i+1), \ldots$, and $\sigma_i^j$ denotes the segment path $\sigma(i), \sigma(i+1), \sigma(i+2), ..., \sigma(j-1), \sigma(j)$. The segment path is infinite in length when the last state of the segment is repeated infinitely often. We write $\Sigma$ to denote the set of evolution paths, and $\Sigma^f$ $(\subseteq \Sigma)$ for the set of finite paths.

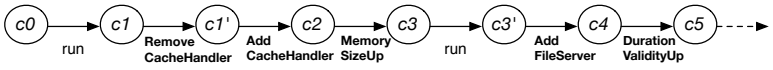


**Fig. 5.** Part of an evolution path of the HTTP server example

*Example 2.* A possible evolution path of the HTTP server is given in Fig. 5. In this path,

– $c_0$ is a configuration of the HTTP server without the **CacheHandler** nor **FileServer2** components;
– $c_1$ is obtained from $c_0$: the load value was changed following the running of the **RequestHandler** component;

- $c'_1$ is the same configuration as $c_1$: Without the **CacheHandler** component, the RemoveCacheHandler reconfiguration cannot terminate, it is then rollbacked without any modification;
- $c_2$ is obtained from the configuration $c_1$ by adding **CacheHandler**, following the AddCacheHandler reconfiguration operation;
- $c_3$ is the configuration $c_2$ in which the memorySize value was increased;
- $c'_3$ is the same configuration as $c_3$: The result of the running is not observable;
- $c_4$ is obtained from $c_3$ by adding the **FileServer2** component;
- $c_5$ is like the configuration $c_6$ but the durationValidity value was increased.

## 4 FTPL

In this section, we recall the syntax of the linear temporal logic for dynamic reconfigurations introduced in [14] and called FTPL. It allows characterizing the correct behaviour of reconfiguration-based systems by using architectural invariants and linear temporal logic patterns. FTPL has been inspired by proposals in [15], and their temporal extensions for JML [24,9,17].

Let us first recall the FTPL syntax as presented in [14]. A *configuration property*, denoted with $conf$, is a first order logic formula over sets and relational operations on the primitive sets and over relations defined in Sect. 3.1. A trace property, denoted with *trace*, is a temporal constraint on (a part of) the execution of the dynamic reconfiguration model. Further, for a reconfiguration operation *ope*, its ending is considered as an event.

$$
\begin{array}{rl}
event ::= & ope \textbf{ terminates} \\
| & ope \textbf{ exceptional} \\
| & ope \textbf{ normal} \\
trace ::= & \textbf{always } conf \\
| & \textbf{eventually } conf \\
| & trace_1 \wedge trace_2 \\
| & trace_1 \vee trace_2 \\
temp ::= & \textbf{after } event\ temp \\
| & \textbf{before } event\ trace \\
| & trace \textbf{ until } event
\end{array}
$$

The *trace properties* specify the constraints to ensure on a sequence of reconfigurations. We mainly specify the **always** and **eventually** constraints which respectively describe that a property has to be satisfied by every configuration of the sequence for the former, or by at least one configuration of the sequence for the latter.

Every *temporal property* concerns a part of the execution trace on which the property should hold: it is specified with special keywords, like e.g., **after**, **before** or **until** a particular event has happened.

The set of FTPL formulae is denoted with $FTPL$. The complete and detailed semantics can be found in [14].

*Example 3.* Let us now illustrate the FTPL language on the example of the HTTP server from Sect. 2. Notice that the reconfiguration AddCacheHandler (resp. RemoveCacheHandler) adds (resp. removes) **CacheHandler** when the deviation value is greater (resp. less) than 50:

$$
\begin{array}{rll}
 & & \textbf{after } \text{RemoveCacheHandler } \textbf{terminates} \\
\textbf{Property 1} & : & (\textbf{eventually } deviation{>}50 \\
 & & \textbf{until } \text{AddCacheHandler } \textbf{terminates})
\end{array}
$$

The previous property specifies that the deviation value eventually becomes greater than 50 between the two considered reconfigurations.

## 5 Runtime Verification for FTPL: RV-FTPL

As component-based architectures evolve at runtime, there is a need to check the FTPL constraints on the fly, even if only a partial information is expected. Indeed, an FTPL property often cannot be evaluated to true or false during the system execution, as only the history of the system is available and no specification of its future evolution exists. In addition, as architectural reconfigurations change the component architecture, they also change the values of FTPL constraints.

In this paper we review the FTPL semantics from a runtime verification point of view. To this end we introduce a new four-valued logic, called RV-FTPL (Runtime Verification for FTPL), characterizing the "potential" (un)satisfiability of the architectural constraints in addition to the basic FTPL semantics. Intuitively, potential true and potential false values are chosen whenever an observed behaviour has not yet lead to a violation or acceptance of the property under consideration.

Let $S$ be a set and $R$ a relation over $S \times S$. $R$ is a pre-ordering iff it is reflexive and transitive, and a partial ordering iff it is anti-symmetric in addition. For a partial ordering $R$, the pair $(S, R)$ is called a partially ordered set; it is sometimes denoted $S$ when the ordering is clear. A lattice is a partially ordered set $(S, R)$ where for each $x, y \in S$, there exists (i) a unique greatest lower bound, and (ii) a unique least upper bound. A lattice is finite iff $S$ is finite. Every finite lattice has a well-defined unique least element, often called the minimum, and a well-defined greatest element, often called the maximum.

More specifically, let $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$ be a set where $\bot, \top$ stand resp. for *false* and *true* values where as $\bot^p, \top^p$ stand resp. for *potential false* and *potential true* values. We consider $\mathbb{B}_4$ together with the truth non-strict ordering relation $\sqsubseteq$ satisfying $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$. On $\mathbb{B}_4$ we define the unary operation $\neg$ as $\neg\bot = \top$, $\neg\top = \bot$, $\neg\bot^p = \top^p$, and $\neg\top^p = \bot^p$, and we define two binary operations $\sqcap$ and $\sqcup$ as the minimum, respectively the maximum, interpreted with respect to $\sqsubseteq$. Thus, $(\mathbb{B}_4, \sqsubseteq)$ is a finite *de Morgan* lattice but not a Boolean lattice.

Before defining the RV-FTPL semantics, let us recall that a configuration property $conf \in FTPL$ is valid on a configuration $c = \langle Elem, Rel \rangle$ when the evaluation of $conf$ on the configuration $c = \langle Elem, Rel \rangle$ is true, written $[c \models conf] = \top$; otherwise, the property $conf$ is not valid on $c$, written $[c \models conf] = \bot$.

**Definition 1 (RV-FTPL Semantics).** *Let $\sigma_0^n \in \Sigma^f$ be a finite execution path of the length $n+1$. Given an FTPL property, its value on $\sigma_0^n$ is given by the interpretation function $[\_ \models \_]_{rv} : \Sigma^f \times FTPL \to \mathbb{B}_4$ defined as follows:*

1. *For the configuration properties and events:*

$$
\begin{array}{|ll|}
\hline
[\sigma_0^n(i) \models conf]_{rv} & = \begin{cases} \top \; if \; [\sigma_0^n(i) \models conf] = \top \\ \bot \; otherwise \end{cases} \\
[\sigma_0^n(i) \models ope \; \textbf{normal}]_{rv} & = \begin{cases} \top \; if \; 0 < i \leqslant n \; \wedge \sigma_0^n(i-1) \neq \sigma_0^n(i) \\ \qquad \wedge \sigma_0^n(i-1) \overset{ope}{\rightarrow} \sigma_0^n(i) \in \rightarrow \\ \bot \; otherwise \end{cases} \\
[\sigma_0^n(i) \models ope \; \textbf{exceptional}]_{rv} & = \begin{cases} \top \; if \; 0 < i \leqslant n \; \wedge \sigma_0^n(i-1) = \sigma_0^n(i) \\ \qquad \wedge \sigma_0^n(i-1) \overset{ope}{\rightarrow} \sigma_0^n(i) \in \rightarrow \\ \bot \; otherwise \end{cases} \\
[\sigma_0^n(i) \models ope \; \textbf{terminates}]_{rv} & = \begin{cases} \top \; if \; ope \; \textbf{normal} \; \vee \; ope \; \textbf{exceptional} \\ \bot \; otherwise \end{cases} \\
\hline
\end{array}
$$

2. *For the trace properties:*

$$
\begin{array}{|ll|}
\hline
[\sigma_0^n \models \textbf{always} \; conf]_{rv} & = \begin{cases} \bot \quad if \; \exists i.(0 \leqslant i \leqslant n \; \wedge [\sigma_0^n(i) \models conf]_{rv} = \bot) \\ \top^p \; otherwise \end{cases} \\
[\sigma_0^n \models \textbf{eventually} \; conf]_{rv} & = \begin{cases} \top \quad if \; \exists i.(0 \leqslant i \leqslant n \; \wedge [\sigma_0^n(i) \models conf]_{rv} = \top) \\ \bot^p \; otherwise \end{cases} \\
[\sigma_0^n \models trace_1 \wedge trace_2]_{rv} & = [\sigma_0^n \models trace_1]_{rv} \sqcap [\sigma_0^n \models trace_2]_{rv} \\
[\sigma_0^n \models trace_1 \vee trace_2]_{rv} & = [\sigma_0^n \models trace_1]_{rv} \sqcup [\sigma_0^n \models trace_2]_{rv} \\
\hline
\end{array}
$$

3. *For the temporal properties:*

$$
\begin{array}{|ll|}
\hline
[\sigma_0^n \models \textbf{after} \; event \; temp]_{rv} & = \begin{cases} \top^p \; if \; \forall i.(0 \leqslant i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \Rightarrow [\sigma_i^n \models temp]_{rv} = \top) \vee \forall i.(0 < i \leqslant n \\ \qquad \Rightarrow [\sigma_0^n(i) \models event]_{rv} = \bot) \\ \bot \quad if \; \exists i.(0 \leqslant i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \wedge [\sigma_i^n \models temp]_{rv} = \bot) \\ \bot^p \; if \; \exists i.(0 \leqslant i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \wedge [\sigma_i^n \models temp]_{rv} = \bot^p) \end{cases} \\
[\sigma_0^n \models \textbf{before} \; event \; trace]_{rv} & = \begin{cases} \top^p \; if \; \forall i.(0 < i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \Rightarrow [\sigma_0^{i-1} \models trace]_{rv} \in \{\top, \top^p\}) \vee \\ \qquad \forall i.(0 < i \leqslant n \Rightarrow [\sigma_0^n(i) \models event]_{rv} = \bot) \\ \bot \quad if \; \exists i.(0 < i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \wedge [\sigma_0^{i-1} \models trace]_{rv} \in \{\bot, \bot^p\}) \end{cases} \\
[\sigma_0^n \models trace \; \textbf{until} \; event]_{rv} & = \begin{cases} \top^p \; if \; \forall i.(0 < i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \Rightarrow [\sigma_0^{i-1} \models trace]_{rv} \in \{\top, \top^p\}) \\ \bot \quad if \; ([\sigma_0^n \models trace]_{rv} = \bot) \vee \\ \qquad (\exists i.(0 < i \leqslant n \; \wedge [\sigma_0^n(i) \models event]_{rv} = \top \\ \qquad \wedge [\sigma_0^{i-1} \models trace]_{rv} = \bot^p) \\ \bot^p \; if \; \forall i.(0 < i \leqslant n \; \Rightarrow [\sigma_0^n(i) \models event]_{rv} = \bot \end{cases} \\
\hline
\end{array}
$$

Let us now comment and illustrate the above definition. The goal of our work is to be able to detect when the FTPL properties become false. So, for configuration properties and events, the interpretation does only depend on the fact that considered configurations actually belong to the path $\sigma_0^n$. For events, the basic FTPL semantics is reflected in the interpretation function.

For trace properties the intuition is as follows.

- The **always** *conf* property is not satisfied on $\sigma_0^n$ if there is a configuration of $\sigma_0^n$ which does not satisfy *conf*. For the other cases, the property is evaluated to be "potentially true". Indeed, if the execution terminated in $\sigma_0^n$, the property would be satisfied.

– The **eventually** *conf* property is satisfied on $\sigma_0^n$ if at least one configuration of $\sigma_0^n$ satisfies *conf*. In the other cases, the property is evaluated to be "potentially false". Indeed, if the execution terminated in $\sigma_0^n$, the property would be violated.

*Example 4.* Figure 6 displays an evolution path of the HTTP example. The next array illustrates the evaluation of two trace properties on each configuration, depending on the chosen either FTPL or RV-FTPL semantics:

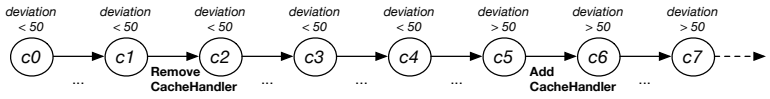


**Fig. 6.** Part of an evolution path of the HTTP server example

| | | $c_0$ | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| **always** *deviation* $< 50$ | FTPL | ? | ? | ? | ? | ? | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| | RV-FTPL | $\top^p$ | $\top^p$ | $\top^p$ | $\top^p$ | $\top^p$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| **eventually** *deviation* $> 50$ | FTPL | ? | ? | ? | ? | ? | $\top$ | $\top$ | $\top$ | $\top$ |
| | RV-FTPL | $\bot^p$ | $\bot^p$ | $\bot^p$ | $\bot^p$ | $\bot^p$ | $\top$ | $\top$ | $\top$ | $\top$ |

Considering the FTPL semantics, we cannot conclude about the interpretation of the considered properties, until we reach the configuration $c_5$. On the contrary, in RV-FTPL we say at the beginning that the **always** property is expected to be true in the future, until we reach $c_5$ where it is false.

The intuition of the definition of temporal properties is as follows:

– The value of the **after** *event temp* property is potentially true either if the *event* event does not occur in all considered configurations, or if the occurrence of the *event* event on a configuration implies that the *temp* temporal property is evaluated to true on the suffix of the path starting at this configuration. The **after** *event temp* property is evaluated to false if there is a configuration $\sigma_0^n(i)$ of $\sigma_0^n$ where the *event* event happens and *temp* is evaluated to false on the suffix $\sigma_i^n$. The **after** *event temp* property is evaluated to potentially false if there is a configuration $\sigma_0^n(i)$ of $\sigma_0^n$ where the *event* event occurs, and *temp* is evaluated to potentially false on the suffix $\sigma_i^n$.
– The value of the **before** *event trace* property is potentially true if either the *event* event does not occur in all considered configurations, or if *trace* is evaluated either to true or to potentially true on the prefix of the path where the *event* event occurs. The **before** *event trace* property is evaluated to false if there is a configuration $\sigma_0^n(i)$ of $\sigma_0^n$ where *event* happens, and *trace* is evaluated either to false or to potentially false on the path ending at $\sigma_0^n(i)$, non including this configuration.

– The value of the *trace* **until** *event* property is potentially true if the *trace* property is evaluated either to true or to potentially true on the prefix of the path where there is a configuration satisfying *event*, the prefix being without that configuration. The *trace* **until** *event* property is evaluated to false either if there is a configuration $\sigma_0^n(i)$ of $\sigma_0^n$ where *event* happens, and if *trace* is either false or potentially false on the path ending at $\sigma_0^n(i)$ but non-including it; or if $\sigma_0^n$ does not satisfy the *trace* property when *event* does not happen on $\sigma_0^n$. The property is potentially false if the *event* event does not occur in all considered configurations.

*Example 5.* Let us again consider the path in Fig. 6 and the FTPL property 1

**after** RemoveCacheHandler **terminates**
(**eventually** *deviation*>50
**until** AddCacheHandler **terminates**)

explained in Example 3. The following array displays the value of the considered property interpreted respectively in FTPL and in RV-FTPL:

| | | $c_0$ | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| **Property 1** | FTPL | ? | ? | ? | ? | ? | ? | $\top$ | $\top$ | $\top$ |
| | RV-FTPL | $\top^p$ | $\top^p$ | $\bot^p$ | $\bot^p$ | $\bot^p$ | $\bot^p$ | $\top^p$ | $\top^p$ | $\top^p$ |

From the FTPL semantics point of view, we cannot conclude about the validity of the property until we reach the configuration $c_6$. Using the RV-FTPL semantics, the property interpretation is potential true before the reconfiguration RemoveCacheHandler is executed. Then, the property value becomes potential false until the deviation becomes greater than 50 on $c_5$; as a consequence the property value becomes potentially true because of partial information.

## 6 Using RV-FTPL Properties to Check Reconfigurations

The proposals of the paper have been applied to the Fractal component model. Thsi section presents the prototype we have been developing to check at runtime the satisfiability of RV-FTPL formulas on Fractal component-based systems. To this end, it exploits and adapts the FPath and FScript [12] tool supports for Fractal to evaluate the desired RV-FTPL formulas after each reconfiguration operation.

### 6.1 Overview of Fractal, FPath and FScript

The Fractal model is a hierarchical and reflective component model intended to implement, deploy and manage software systems [10]. A Fractal component is both a design and a runtime entity that consists of a unit of encapsulation, composition and configuration. A component is wrapped in a membrane which can show and control a casually connected representation of its encapsulated

content. This content is either directly an implementation in case of a primitive component, or sub-components for composite components.

In order to control the internal structure of a component at runtime, the Fractal model also defines standard interfaces named *controllers*. In addition, the Fractal model can be extended thanks to new controllers which allow the user to integrate new features.

FPath [12] is a domain-specific language inspired by the XPath language that provides a notation and introspection mechanisms to navigate inside Fractal architectures. FPath expressions use the properties of components (e.g. the value of a component attribute or the state of a component) or architectural relations between components (e.g. the subcomponents of a composite component) to express queries about Fractal architectures.

FScript [12] is a language that allows the definition of reconfigurations of Fractal architectures. FScript integrates FPath seamlessly in its syntax, FPath queries being used to select the elements to reconfigure. To ensure the reliability of its reconfigurations, FScript considers them as transactions and integrates a back-end that implements this semantics on top of the Fractal model.

### 6.2 Integrating RV-FTPL Property Verification into Fractal

To check RV-FTPL properties at runtime, we have implemented two Fractal controllers which observe the Fractal component model: our first controller, called the *reconfiguration controller*, permits capturing reconfiguration invocations, whereas the second controller, called the *RV-FTPL controller*, handles RV-FTPL formulas.

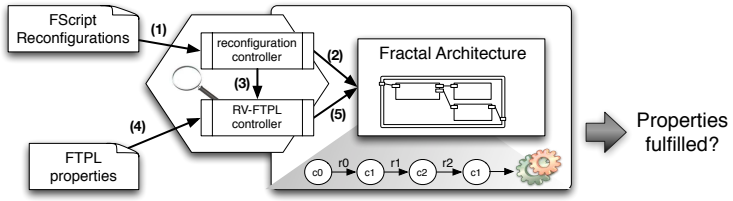


**Fig. 7.** RV-FTPL runtime verification principle

Figure 7 explains how both controllers are used to evaluate properties of interest. When a reconfiguration is invoked **(1)**, the reconfiguration controller executes the reconfiguration **(2)**—specified in a FScript file—on the considered component-based architecture. It then invokes the RV-FTPL controller **(3)** to evaluate the RV-FTPL properties from a file **(4)** where those properties are specified. The RV-FTPL controller uses the instantiated component model **(5)** and executes queries over it: to post up the property evaluation result to the user, the RV-FTPL controller parses the property of interest and uses a visitor to evaluate it on the current configuration using FPath. In the case of the future

patterns containing the **after** keyword, the visitor waits for the reconfiguration event before evaluating the temporal part of the property. On the contrary, for the past patterns, i.e., the RV-FTPL properties without the **after** keyword, the trace part of the property is evaluated before the reconfiguration event appears. This avoids us from saving all the previous configurations needed to evaluate the property once the event appears.

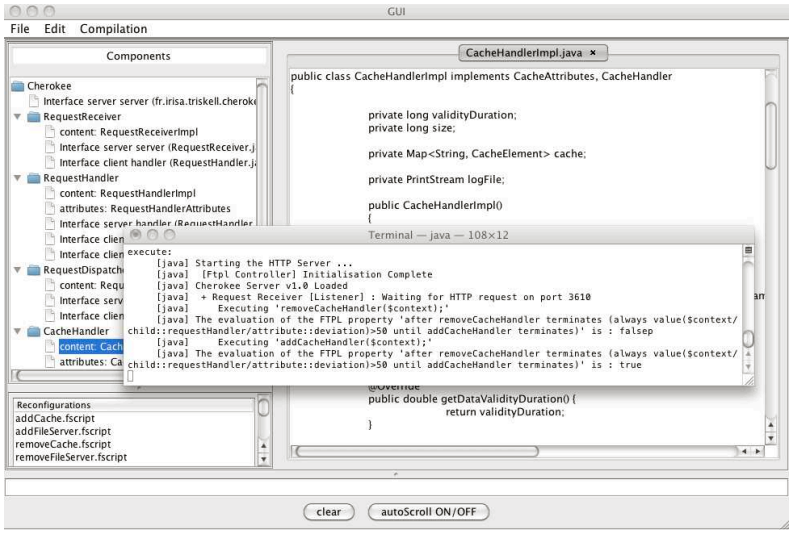


**Fig. 8.** Running prototype

The above verification procedure has been integrated into the EVA4Fractal tool previously described in [13]. Figure 8 shows our prototype in action: a Fractal implementation of the HTTP server example is running and the FTPL property

**after** RemoveCacheHandler **terminates**
(**eventually** *deviation*>50
**until** AddCacheHandler **terminates**)

is evaluated at runtime after each reconfiguration execution. The reader can notice that after the execution of the reconfiguration RemoveCacheHandler, the value of the property is potential false. If the value of deviation raises above 50, when the reconfiguration AddCacheHandler is applied, the property value becomes true.

## 7 Using RV-FTPL Properties to Enforce Reconfigurations

As explained in Sect. 1, one of the main motivations of the present work is to use the RV-FTPL property evaluation to control the execution of reconfigurations.

Actually, for some kind of systems like critical systems or embedded systems, the behaviour where the property evaluation becomes false might be not acceptable. To this end, we can use potential true or potential false values to enforce the reconfigurations.

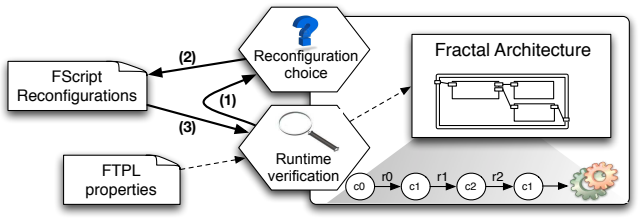


**Fig. 9.** RV-FTPL runtime enforcement principle

In this section, we show the capability of our monitor to enforce the component-based system reconfigurations by using the interpretation of desired properties. The principle is illustrated in Fig. 9. While interpreting RV-FTPL properties **(1)**, the potential true or potential false values can be used to guide the choice of the next reconfiguration operation **(2)** which will be applied to the component architecture **(3)**. Let us give an intuition about our approach:

1. Let us consider the RV-FTPL property 3 valued $\bot^p$ on the current architectural configuration $c_5$ from the path given in Fig. 6;
2. We are looking for enabling the reconfiguration operations that make the component-architecture evolve to a new architectural configuration where the RV-FTPL property will be enforced;
3. The reconfiguration manager chooses the reconfiguration AddCacheHandler to be applied;
4. The property will be enforced: it is valued to $\top^p$ on the new configuration $c_6$.

In Fractal an obvious manner to implement the *reconfiguration choice* procedure is to reuse the transaction mechanism of FScript [12], allowing the system to rollback to a consistent state when a reconfiguration operation failed. We propose to exploit this mechanism to evaluate the RV-FTPL property on the possible target configurations, until a reconfiguration operation where the system benefits enforcement in the best possible way, is found.

We display in Fig. 10 the execution scenario using this mechanism. For each FScript reconfiguration, a transaction is started and the considered reconfiguration operation is executed. Then, the RV-FTPL property is evaluated on the reached configuration. If the interpretation value is true, there is no need to consider remaining reconfiguration operations, so the transaction is committed and the execution goes on. For other interpretation values, the transaction is rollbacked and the results of the reconfiguration valuation are recorded. When

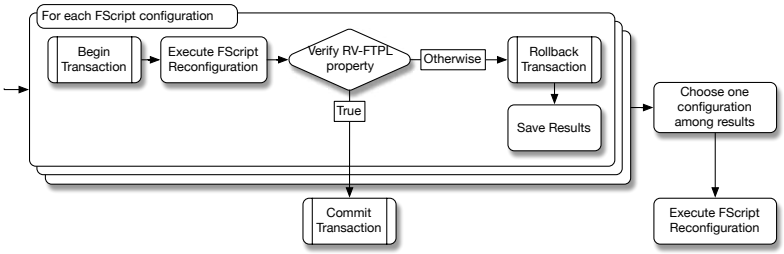


**Fig. 10.** Enforcement scenario

all the enable reconfigurations are explored, the recorded results are used to choose the most appropriate reconfiguration operation which is then applied to the system. To help this choice, adaptation policies [13,11] defined by the user, or distributed controllers [18] for knowledge-based priority properties, or runtime enforcement monitors [16] built automatically for several enforcable properties, can be used.

If for every reconfiguration operation the property of interest is violated, the execution should be either stopped or continued with special recovery operations, and the user should be informed. This reaction clearly depends on the system features (safety critical systems, embedded systems, etc.). Again, adaptation policies can be used to handle events associated with the property violation on the one hand, and to specify special recovery reconfiguration operations, on the other hand.

## 8 Conclusion

As component-based architectures evolve at runtime, this paper pays particular attention to checking—on the fly—temporal and architectural constraints expressed with a linear time temporal logic over (re)configuration sequences, FTPL [14]. Unfortunately, an FTPL property often cannot be evaluated to true or false during the system execution. Indeed, only a partial information about the system evolution is available: only a (finite) history of the system state is known, and no specification about its future evolutions exists. To remedy this problem, we have reviewed the FTPL semantics from a runtime verification point of view. Inspired by proposals in [8], we have introduced a new four-valued logic, called RV-FTPL, characterizing the "potential" (un)satisfiability of the architectural FTPL constraints in addition to the basic FTPL semantics.

The paper has also reported on the prototype we have been developing to verify and enforce RV-FTPL properties. Given a Fractal component-based system [10] and some desired temporal and archtectural FTPL contraints, to make it possible the system to reconfigure, the prototype interprets RV-FTPL formulas at runtime. The feasability of a runtime property enforcement has also been discussed: the proposed 4-valued logic not only captures information absence,

but also helps the monitor in guiding the reconfiguration process, namely in choosing the next reconfiguration operations to be applied.

**Related work.**

In the context of dynamic reconfigurations, ArchJava [3] gives means to reconfigure Java architectures, and the ArchJava language guarantees communication integrity at runtime. Barringer and al. give a temporal logic based framework to reason about the evolution of systems [5]. In [4], a temporal logic is proposed to specify and verify properties on graph transformation systems.

In the Fractal-based framework, the work in [21] has defined integrity constraints on a graph-based representation of Fractal, to specify the reliability of component-based systems. Unlike [21], our model lays down only general architectural constraints, thus providing an operational semantics to other component-based systems, to their refinements and property preservation issues. On the integrity constraints side, the FTPL logic allows specifying architectural constraints more complex than architectural invariants in [12]. Let us remark that architectural invariants as presented in [12] can be handled within the FTPL framework by using **always** *cp*, where *cp* represents the considered architectural invariant.

Among other applications, our proposals aim at a monitoring of component-based systems. In [6], Basin and al. have shown the feasibility of monitoring temporal safety properties (and, more recently, security properties) using a runtime monitoring approach for metric First-order temporal logic (MFOTL). In [23,22], monitors are used to check some policies at runtime, and to enforce the program to evolve correctly by applying reconfigurations. A similar approach based on a three-valued variant of LTL has been proposed in [7]. Contrary to those works, we focus on temporal and architectural constraints to make it possible *component-based* systems to reconfigure at runtime.

In [8], a three-valued and a four-valued LTL are studied from a logic point of view. In [16], the authors have studied the class of enforceable properties from the point of view of the well-known temporal property hierarchies. The automatic monitor generation for enforceable properties has also been proposed. In this direction, it would be interesting and important to characterize the FTPL temporal patterns wrt. the class of enforceable properties. For non-enforceable temporal patterns, we intend to exploit event-based adaptation policies to make the system behave and reconfigure according to a given recovery policy when the desired property is violated.

## References


1. M. Aguilar Cornejo, H. Garavel, R. Mateescu, and N. De Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. Research Report RR-4222, INRIA, 2001.
2. N. Aguirre and T. Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. *Automated Software Engineering*, 2002.
3. J. Aldric. Using types to enforce architectural structure. In *WICSA'08*, pages 23–34, 2008.

4. P. Baldan, A. Corradini, B. König, and A. Lluch Lafuente. A temporal graph logic for verification of graph transformation systems. In *WADT'06*, pages 1–20. Springer-Verlag, 2007.
5. H. Barringer, D. M. Gabbay, and D. E. Rydeheard. From runtime verification to evolvable systems. In *RV*, volume 4839 of *LNCS*, pages 97–110. Springer, 2007.
6. D. A. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *IARCS, FSTTCS 2008, India*, volume 2 of *LIPIcs*, pages 49–60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
7. A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *ASWEC'iso06. IEEE*, 2006.
8. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation (JLC)*, 2010.
9. F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical report RR-5331, INRIA, 2004.
10. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.
11. F. Chauvel, O. Barais, I. Borne, and J.-M. Jézéquel. Composition of qualitative adaptation policies. In *ASE 2008*, pages 455–458. IEEE, 2008. Short paper.
12. P.-C. David, Th. Ledoux, M. Léger, and Th. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009.
13. J. Dormoy and O. Kouchnarenko. Event-based Adaptation Policies for Fractal Components. In *AICCSA 2010*, pages 1–8. IEEE, May 2010.
14. J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In *FACS 2010*, volume 6921 of *LNCS*. Springer-Verlag, 2010. to appear.
15. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
16. Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.
17. A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with java modelling language. *IET Software*, 2008.
18. S. Graf, D. Peled, and S. Quinton. Achieving distributed control through model checking. In *CAV 2010*, volume 6174 of *LNCS*, pages 396–409. Springer, 2010.
19. A. Lanoix, J. Dormoy, and O. Kouchnarenko. Combining proof and model-checking to validate reconfigurable architectures. In *FESCA 2011*, ENTCS, 2011.
20. M. Léger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composant*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2009.
21. M. Léger, Th. Ledoux, and Th. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In *CBSE 2010*, volume 6092 of *LNCS*, pages 74–92, 2010.
22. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM TISSEC*, 12:19:1–19:41, January 2009.
23. F. B. Schneider. Enforceable security policies. *ACM TISSEC*, 3:30–50, 2000.
24. K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *AMAST 2002*, volume 2422 of *LNCS*, pages 334–348, 2002.