Using Temporal Logic for Dynamic Reconfigurations of Components

Julien Dormoy, Olga Kouchnarenko, Arnaud Lanoix

HAL Id: hal-00662875
https://hal.science/hal-00662875

Submitted on 25 Jan 2012

Using Temporal Logic 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 run-time. This paper deals with the formal specification and verification of dynamic reconfigurations of those systems using architectural constraints and temporal logic patterns.

The proposals of the paper are applied to the Fractal component model. Given a Fractal reference implementation of a component-based system, we specify its dynamic reconfigurations using a temporal pattern logic for Fractal, called FTPL, characterizing the correct behaviour of the system under some architectural constraints. We study system reconfigurations on which we verify these requirements, in particular by reusing the FPath and FScript tools.

1   Introduction

Component-based development provides significant advantages like portability, adaptability, re-usability, etc. when developing, e.g., Java Card smart card applications or when composing components or services within Service Component Architecture (SCA). The adaptability means that component-based systems must be adapted, or even adapt themselves [12] to their environments during their lifetime, and there is a need to support dynamic reconfigurations, including unanticipated ones [17]. To take up this challenge, this paper deals with the formal specification and verification of dynamic reconfigurations of component-based systems, and uses temporal patterns to monitor them.

The present paper makes the following contributions. The first contribution is a formal definition of the semantics of component-based systems with reconfigurations. To specify system reconfigurations, the second contribution is the definition of a linear time temporal logic based on architectural constraints which are first order configuration properties, and on event properties. For temporal operators, its expressive power is related to the well known linear time temporal logic (LTL) [16]. The third contribution is the application of the paper proposals to the Fractal component model. For the Fractal component model, run-time verification issues are addressed to monitor reconfigurations during system lifetime.

More precisely, this paper follows the line of reasoning suggested in [9], where the system consistency during its dynamic reconfigurations relies on *integrity constraints*—predicates on assemblies of architectural elements and component states. To go further, we propose to support dynamic reconfigurations by using more complex architectural constraints and linear temporal logic patterns. These temporal patterns have been inspired by the pragmatic work of the SanTos Specification Pattern Project [10], and works on temporal extension of JML [18, 6, 11] helping Java programmers in writing formal specifications. We refer to this temporal extension as FTPL, for Temporal Pattern Language, prefixed by an 'F' to denote its adaptation to Fractal-like component systems and to first-order integrity constraints over them.

The proposals of the paper are applied to the Fractal component model. Given a Fractal reference implementation of a component-based system, we specify its dynamic reconfigurations using FTPL, characterizing the correct behaviour of the system under some architectural constraints. We monitor system reconfigurations by reusing the FPath and FScript tool supports.

The remainder of the paper is organised as follows. After giving a motivating example in Sect. 2, we formally define the semantics of component-based systems with reconfigurations in Sect. 3. To support system reconfigurations, Sect. 4 introduces a linear time temporal logic based on architectural constraints and events. Then, the proposals of the paper are applied to and illustrated on the Fractal component model in Sect. 5. Finally, Section 6 concludes before discussing related work.

2 Motivating Example

To motivate and to illustrate our approach, let us consider an example of a HTTP server from [8]. The architecture of this server is displayed in Fig. 1.
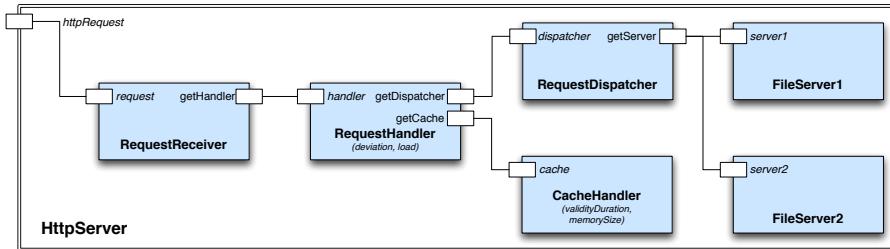

Fig. 1. HTTP Server architecture

The **RequestReceiver** component reads HTTP requests from the network and transmits them to the **RequestHandler** component. In order to perform a request, **RequestHandler** can either use the cache (with the component

CacheHandler) or transmit the request to the **RequestDispatcher** component. This component uses a set of file servers (like the **FileServer1** and **FileServer2** components) to answer the request.

This architecture provides a cache (**CacheHandler** component) and a load controller (**RequestDispatcher** component) in order to control the response time of the HTTP server. To keep the response time as short as possible whatever the number of requests is, in [8] the authors propose to dynamically reconfigure the HTTP server. For that, some requirements have been identified:

1. The **CacheHandler** component is used only if the number of similar HTTP requests is high.
2. The quantity of allocated memory for the **CacheHandler** component must depend on the overall load of the server.
3. The validity of data in the cache must also depend on the overall load of the server.
4. The number of used file servers depends on the overall load of the server.

In order to take these requirements into account, the **RequestHandler** and **CacheHandler** components are equipped with some parameters. The number of requests (load) and the percentage of similar requests (deviation) are two parameters defined for the **RequestHandler** component. The memory size (memorySize) and the data validity duration (validityDuration) are two parameters defined for the **CacheHandler** component.

We consider that the HTTP server can be reconfigured during the execution by the following reconfigurations

1. AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the **CacheHandler** component,
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.

3 Architectural (Re-)Configuration Model

This section gives means for specifying component-based systems with reconfigurations. A model we propose here is inspired by the model in [13, 15] given for Fractal. Both models are graphs allowing one to represent component-based architectures and reconfiguration operations and to reason about them. Unlike [13, 15], in our model, only the basic and generic concepts are considered to allow their application to various hierarchical component models: *components* as runtime entities, *required* and *provided interfaces* as interaction points between components, *bindings* to link component interfaces. Components are either *primitive* components or *composite* components, and primitive components can have some attributes used as configuration parameters.

Basically, a component-based system with reconfigurations is characterized by a set of configurations and a set of actions that allow the modification of configurations.

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. We define a configuration to be a set of architectural elements (components, interfaces and parameters) together with a relation to structure and to link them. We consider a graph-based representation as proposed in [13, 15].

Definition 1. *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 between architectural elements.*

In our model, the architectural elements are the core entities of a component-based system: components, required or provided interfaces, and parameters.

Definition 2. *The set of architectural elements Elem is defined by:*

$$Elem \ = \ Component \ \uplus \ Interface \ \uplus \ Parameter \ \uplus \ Type$$

where

- *Component is a non-empty set of the core entities, i.e components;*
- *$Interface = Required \uplus Provided$ is a set of the (required and provided) interfaces;*
- *Parameter is a set of component parameters;*
- *Type is a set of data types associated with parameters.*

Each data type is a set of data values. For the sake of readability, we identify data type names with the corresponding data domains.
The architectural relation *Rel* then expresses various links between architectural elements. For example, it allows specifying that a component has an interface or a parameter, or that a component contains other (sub-)components, or that an interface is linked to another one.

Definition 3. *The architectural relation Rel is defined by:*

$$Rel \ = \ \begin{array}{l} ProvidedBy \ \uplus \ RequiredBy \ \uplus \ ParameterOf \ \uplus \\ TypeOf \ \uplus \ ValueOf \ \uplus \ ChildOf \ \uplus \ Binding \ \uplus \ Delegate \end{array}$$

where

- *$ProvidedBy : Provided \rightarrow Component$ is a total surjective function which gives the component having a provided interface;*
- *$RequiredBy : Required \rightarrow Component$ is a total function which gives the component with a required interface;*

- $ParameterOf : Parameter \rightarrow Component$ is a total function which gives the component of a considered parameter;
- $TypeOf : Parameter \rightarrow Type$ is a total function which gives the type associated with a considered parameter;
- $ValueOf : Parameter \rightarrow \bigcup_{type \in Type} type$ such that $\forall p \in Parameter : ValueOf(p) \in TypeOf(p)$, is a total function which gives the current value of a considered parameter;
- $ChildOf \subseteq Component \times Component$ is a irreflexive and antisymetric relation linking composite components to their sub-components [3] such that:
  - $\forall\ c, c'\ \in\ Component.\ ((c, c')\ \in\ ChildOf\ \Rightarrow\ \forall\ p\ \in\ Parameter.\ (ParameterOf(p) \neq c))$, i.e, composite components have no parameter;
  - Let $ChildOf^+$ be the transitive closure of $ChildOf$. Then, $\forall c, c' \in Component.\ ((c, c') \in ChildOf^+ \Rightarrow c \neq c')$, i.e., $ChildOf$ is an acyclic relation;
- $Binding : Provided \rightarrow Required$ is a partial function such that $\forall\ ip \in Provided, ir \in Required.\ (Binding(ip) = ir \Rightarrow ProvidedBy(ip) \neq RequiredBy(ir) \wedge \exists\ c \in Component.\ ((c, ProvidedBy(ip)) \in ChildOf \wedge (c, RequiredBy(ir)) \in ChildOf))$, i.e., two linked interfaces do not belong to the same component, but the corresponding components are sub-components of the same composite component;
- $Delegate : Interface \rightarrow Interface$ is a partial injective function to specify the delegation from a sub-component interface to the composite interface such that $\forall\ i, i'\ \in Interface.\ (Delegate(i) = i' \Rightarrow (ProvidedBy(i'), ProvidedBy(i)) \in ChildOf \vee (RequiredBy(i'), RequiredBy(i)) \in ChildOf)$, i.e., when delegating, the component providing i is a sub-component of the component providing i', or the component requiring i is a sub-component of the component requiring i'.

Example 1. Figure 2 illustrates main lines of Definition 3 on the example from Section 2. In this figure, the architectural elements are depicted as boxes and circles, whereas architectural relations are represented by arrows. For example, the request architectural element (at the bottom on the left) is an interface provided by **RequestReceiver**. The **RequestReceiver** architectural element is a sub-component of the **HttpServer** composite component which provides the httpRequest interface. The request interface delegates results passing to the httpRequest interface.

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:

---

[3] For any $(p, q) \in ChildOf$, we say that p has a sub-component q, i.e. q is a child of p.
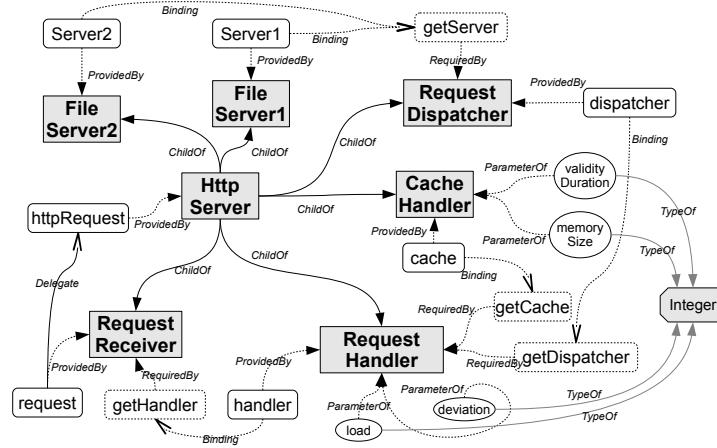
Fig. 2. Graph representation of the HTTP Server example

- 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. Notice that reconfigurations are not the only manner to make a component architecture evolve. The normal running of different components also changes the architecture by modifying parameter values or stopping components, like in the example.

Considering the component-based architecture model given in Sect. 3.1, an operation which makes the component architecture evolve by a reconfiguration action or by running, 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 $c \xrightarrow{op} c'$. 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.

Definition 4. *The set of evolution operations $\mathcal{R}_{run}$ is defined by:*

$$\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$$

with

- $\mathcal{R}$ *is a finite set of reconfiguration operations;*

– *run is an action renaming one or more running operations.*

Given a component architecture and the set $\mathcal{R}_{run}$ of reconfiguration operations, the behaviour of the component architecture is defined as a transition system labelled over $\mathcal{R}_{run}$.

Definition 5. *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,*
– $\mathcal{R}_{run}$ *is a finite set of evolution operations,*
– $\rightarrow \ \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ *is the reconfiguration relation.*

Given the evolution of a component architecture, we can now define the useful notions of path, trace, etc.

Definition 6. *Given the model $M = \langle \mathcal{C}, \mathcal{R}_{run}, \rightarrow \rangle$, an evolution path (or a path for short) σ of M is a (possibly infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0.\exists r_i \in \mathcal{R}_{run}.c_i \xrightarrow{r_i} c_{i+1} \in \rightarrow)$).*

We use $\sigma(i)$ to denote the i-th configuration of a path σ. The notation σ_i denotes the suffix path $\sigma(i), \sigma(i+1), \ldots,$ and σ_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.
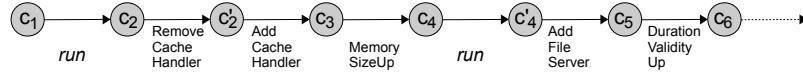
Fig. 3. Example

Example 2. A possible evolution path of the HTTP server is given in Fig. 3. In this path,

– c_1 is a configuration of the HTTP server without the **CacheHandler** and **FileServer2** components;
– c_2 is obtained from c_1: the load value was changed following the running of the **RequestHandler** component;
– c_2' is the same configuration as c_2. Without the **CacheHandler** component, the RemoveCacheHandler reconfiguration cannot terminate, it is then rollbacked without any modification;
– c_3 is obtained from the configuration c_2 by adding **CacheHandler**, following the AddCacheHandler reconfiguration operation;
– c_4 is the configuration c_3 in which the memorySize value was increased;
– c_4' is the same configuration as c_4. The result of the running is not observable;
– c_5 is obtained from c_4 by adding the **FileServer2** component;
– c_6 is like the configuration c_5 but the durationValidity value was increased.

4   Temporal Logic for Dynamic Reconfigurations

This section presents the syntax and the semantics of the temporal logic for dynamic reconfigurations. This logic, called FTPL, is inspired by the temporal logic in [18] designed to help Java programmers in writing formal specifications.

4.1   Syntax of the Logic

Let us first give the FTPL syntax. Figure 4 describes the syntax of the temporal logic for dynamic reconfigurations. The language consists of different layers:

- the configurations properties,
- the reconfiguration operations,
- the trace properties,
- the temporal properties.

$<TempProp>$	::=	**after** $<Events>$ $<TempProp>$
	|	**before** $<Events>$ $<TraceProp>$
	|	$<TraceProp>$ **until** $<Events>$
	|	$<TraceProp>$ **unless** $<Events>$
	|	**between** $<Events>$ $<Events>$ $<TraceProp>$
	|	$<TraceProp>$
$<TraceProp>$	::=	**always** ConfigProp
	|	**eventually** ConfigProp
	|	**never** ConfigProp
	|	$<TraceProp>$ $\wedge$ $<TraceProp>$
	|	$<TraceProp>$ $\vee$ $<TraceProp>$
$<Events>$	::=	$<Event>$, $<Events>$
	|	$<Event>$
$<Event>$	::=	ReconfigOp **called**
	|	ReconfigOp **normal**
	|	ReconfigOp **exceptional**
	|	ReconfigOp **terminates**

Fig. 4. Syntax of the temporal logic for reconfigurations

4.2   Semantics of FTPL

Let us now give the FTPL semantics. It is defined by induction on the form of the formulas w.r.t. Fig. 4.

Configuration properties Basically, there is a need to express properties on the configurations, i.e constraints on the architectural elements and the relations between them. These constraints are specified using first order logic formulas, sets and relational operations on the primitive sets and relations given in Sect. 3.

Given the model M, we say that a configuration property cp is valid on a configuration $c = \langle Elem, Rel \rangle$, written $M, c \models cp$, when the evaluation of cp on the configuration $c = \langle Elem, Rel \rangle$ is true. When M is understood, we simply write $c \models cp$.

The configuration properties are expressed at different specification levels. At the component model level, the constraints are common to all the component architectures. In addition, some constraints must be expressed to restrict a family of component architectures (a profile level), or to restrict a specific component architecture (an application level).

Example 3. Let $CacheConnected$ be a configuration property defined by

$$\exists \mathsf{cache}, \mathsf{getCache} \in Interface.(ProvidedBy(\mathsf{cache}) = \mathbf{CacheHandler} \wedge$$
$$RequiredBy(\mathsf{getCache}) = \mathbf{RequestHandler} \wedge Binding(\mathsf{cache}) = \mathsf{getCache}).$$

It expresses that the component **CacheHandler** is connected to **RequestHandler**. For the evolution path from Fig. 3 we have: $c3 \models CacheConnected$ whereas $c2 \not\models CacheConnected$.

Event properties We want to observe reconfiguration action effects, for example when a reconfiguration is called or when it terminates, to specify and verify properties over them. Given a reconfiguration operation r in $\mathcal{R}$, we consider the following events:

- r **called** denoting that the reconfiguration r has been invoked,
- r **normal** denoting that the reconfiguration r has terminated normally,
- r **exceptional** denoting that the configuration r has rollbacked.

Definition 7. *Let σ be an evolution path in M, and r a reconfiguration operation in $\mathcal{R}$. Given an event property ϕ depicted in Fig.4, $<Event>$, its validity on the i-th configuration of σ, denoted $\sigma(i) \models \phi$, is inductively defined on the form of ϕ by:*

$$
\begin{array}{ll}
\sigma(i) \models r \textbf{ called} & \text{iff } \exists \sigma(i+1).(\sigma(i) \xrightarrow{r} \sigma(i+1) \in \rightarrow) \\
\sigma(i) \models r \textbf{ normal} & \text{iff } i > 0 \wedge \sigma(i-1) \models r \textbf{ called} \wedge \sigma(i-1) \neq \sigma(i) \\
\sigma(i) \models r \textbf{ exceptional} & \text{iff } i > 0 \wedge \sigma(i-1) \models r \textbf{ called} \wedge \sigma(i-1) = \sigma(i) \\
\sigma(i) \models r \textbf{ terminates} & \text{iff } \sigma(i) \models r \textbf{ normal} \\
& \quad \vee \sigma(i) \models r \textbf{ exceptional}
\end{array}
$$

Given an evolution path σ, and a list of event properties $E = e_1, \ldots, e_n$ as depicted in Fig.4, $<Events>$, we say that E is valid on the i-th configuration of σ, denoted $\sigma(i) \models E$, iff at least one event of the list E is valid on $\sigma(i)$.

Example 4. Let us consider again the evolution path displayed in Fig. 3. We have: $c3 \models$ MemorySizeUp **called**, $c5 \models$ AddFileServer **normal** and $c2 \models$ RemoveCacheHandler **exceptional**.

Trace Properties A trace property expresses a constraint which must be true when the component-based architecture changes, i.e on the evolution path.

Definition 8. *Let σ be an evolution path, and cp a configuration property. Given a trace property trp depicted in Fig.4, $<TraceProp>$, its validity on σ, denoted $\sigma \models trp$, is inductively defined on the form of trp by:*

$$
\begin{array}{ll}
\sigma \models \textbf{always } cp & \textit{iff } \forall i.(i \geqslant 0 \Rightarrow \sigma(i) \models cp) \\
\sigma \models \textbf{eventually } cp & \textit{iff } \exists i.(i \geqslant 0 \wedge \sigma(i) \models cp) \\
\sigma \models \textbf{never } cp & \textit{iff } \forall i.(i \geqslant 0 \Rightarrow \sigma(i) \not\models cp) \\
\sigma \models trp_1 \wedge trp_2 & \textit{iff } \sigma \models trp_1 \wedge \sigma \models trp_2 \\
\sigma \models trp_1 \vee trp_2 & \textit{iff } \sigma \models trp_1 \vee \sigma \models trp_2
\end{array}
$$

Intuitively,

- **always** cp is valid on an evolution path σ iff cp is valid on each configuration of σ;
- **eventually** cp is valid on an evolution path σ iff cp is valid on one configuration of σ, at least;
- **never** cp is valid on an evolution path σ iff no configuration of σ satisfies cp,
- the semantics of conjunction and disjunction is classical.

Let us remark that architectural invariants as presented in [13, 15], can be handled within the FTPL framework by using **always** cp, where cp represents the considered architectural invariant.

Temporal Properties Temporal properties are based on all the properties above, i.e. they exploit architectural constraints, event properties and trace properties, together with some temporal patterns, like in [18]. Let us recall that the SanTos Specification Pattern Project [10] has identified these temporal patterns as useful in practice.

Definition 9. *Let σ be an evolution path, E, E_1 and E_2 be lists of events, trp a trace property and tpp a temporal property, as depicted in Fig. 4.*

$$
\begin{array}{ll}
\sigma \models \textbf{after } E \; tpp & \textit{iff } \forall i.(i \geqslant 0 \wedge \sigma(i) \models E \Rightarrow \sigma_i \models tpp) \\
\sigma \models \textbf{before } E \; trp & \textit{iff } \forall i.(i > 0 \wedge \sigma(i) \models E \Rightarrow \sigma_0^{i-1} \models trp) \\
\sigma \models trp \; \textbf{until } E & \textit{iff } \exists i.(i > 0 \wedge \sigma(i) \models E \wedge \sigma_0^{i-1} \models trp) \\
\sigma \models trp \; \textbf{unless } E & \textit{iff } \forall i.(i \geqslant 0 \wedge \sigma(i) \not\models E \Rightarrow \sigma \models trp) \\
& \quad \vee \; \exists i.(i \geqslant 0 \wedge \sigma(i) \models E \wedge \sigma_0^{i-1} \models trp) \\
\sigma \models \textbf{between } E_1 \; E_2 \; trp & \textit{iff } \exists i, j.(i \geqslant 0 \wedge j > i \wedge \sigma(i) \models E_1 \\
& \quad \wedge \; \sigma(j) \models E_2 \wedge \sigma_i^{j-1} \models trp)
\end{array}
$$

Intuitively,

- the property **after** E tpp is valid on an evolution path σ iff the validity of the event property E on a configuration of σ implies the validity of the temporal property tpp on the suffix of σ starting at this configuration;

- **before** E trp is valid on an evolution path σ iff for each configuration of σ, the validity of E on it means that the trace property trp is valid on the prefix of σ before the considered configuration;
- trp **until** E is valid on an evolution path σ iff there is a configuration of σ satisfying the event property E, and the trace property trp is valid on the prefix of σ ending before this event occurs;
- trp **unless** E is valid on an evolution path σ iff either the event property E is not valid on σ implying that the trace property trp is valid on σ, or there is a configuration of σ satisfying the event property E, and the trace property trp is valid on the prefix of σ before the corresponding event occurs;
- **between** E_1 E_2 trp is valid on an evolution path σ iff both event properties E_1 and E_2 are valid on σ, and the trace property trp is valid on the segment of σ consisting of the configurations in-between the configuration where E_1 holds (including it), and the configuration where E_2 holds (excluding it).

4.3 Application to the HTTP Server Example

Let us now illustrate the FTPL language use by expressing some properties for the example of HTTP server from Sect. 2.

Let us consider a temporal property saying that after the invocation of the reconfiguration operation AddCacheHandler, the **CacheHandler** component is always connected to **RequestHandler**, i.e. the $CacheConnected$ configuration property from Example 3 holds on the path configurations after the invocation. This property is valid on the evolution path σ depicted in Fig. 3:

$$\sigma \models \textbf{after } \textsf{AddCacheHandler } \textbf{called always } CacheConnected.$$

The following property expresses an architectural constraint saying that at least there is always one file server component connected.

$$\textbf{always } \big(\exists \textsf{getServer} \in Interface.(RequiredBy(\textsf{getServer}) = \\ \textbf{RequestDispatcher} \land \exists i \in Interface.Binding(i) = \textsf{getServer})\big)$$

Let us now specify that the deviation must always be lower than 50 until the AddCacheHandler reconfiguration operation terminates normally:

$$(\textbf{always } \\ \exists deviation \in Parameter.(ParameterOf(deviation) = RequestHandler \\ \land \ deviation < 50)) \ \textbf{until } \texttt{AddCacheHandler normal}$$

The following property says that between the exceptional termination of either the `MemorySizeUp` reconfiguration or the `DurationValidityUp` reconfiguration, and the normal termination of the `AddCacheHandler` reconfiguration operation, the number of used file servers is greater than 1:

$$\textbf{between } \big(\texttt{MemorySizeUp exceptional}, \texttt{DurationValidityUp exceptional}\big) \\ \big(\texttt{addCacheHandler normal}\big) \\ \big(\exists \textsf{getServer} \in Interface.(RequiredBy(\textsf{getServer}) = \textbf{RequestDispatcher} \land \\ \exists i, i' \in Interface.(i \neq i' \land Binding(i) = \textsf{getServer} \land Binding(i') = \textsf{getServer})\big)$$

These examples show that architectural invariants and properties on immediate predecessors or target configurations of reconfiguration operations can be expressed by FTPL formulas. Further, they show that FTPL is more expressive than the proposals in [9]. Indeed, FTPL allows expressing event properties and temporal properties involving different kinds of properties satisfying temporal patterns which have been shown useful for practical applications.

4.4   On the Expressiveness of FTPL

Before considering FTPL temporal properties, let us recall that configuration properties are first order logic formulas.

Let us now consider temporal patterns. As explained in Sect.4, FTPL has been inspired by proposals in [10], and works on a temporal extension of JML [18, 6, 11], called JTPL. The semantics of JTPL temporal formulas and translation rules into JML annotations are detailed in [18] for *safety properties* and in [6] for *liveness properties*.

Let $\mathrm{LTL}_k()$ denote a function translating the FTPL properties of the kind k into LTL properties. We adapt the above-mentioned works and propose the following translation of FTPL patterns into LTL formulas. In this translation cp is a configuration property, trp, trp_1 and trp_2 are trace properties, E, E_1 and E_2 are lists of event properties, and tpp is a temporal property. In FTPL, there is a way to decide whether a list of event properties is valid on a configuration or not. The following functions suppose that the same decision procedure exists in LTL.

Leaving aside the FTPL and LTL models, it is easy to see that FTPL trace properties can be rewritten in LTL as follows:

$\mathrm{LTL}_{Trace}(\textbf{always } cp)$	$\mathrm{G}(cp)$
$\mathrm{LTL}_{Trace}(\textbf{eventually } cp)$	$\mathrm{F}(cp)$
$\mathrm{LTL}_{Trace}(trp_1 \ \& \ trp_2)$	$\mathrm{LTL}_{Trace}(trp_1) \wedge \mathrm{LTL}_{Trace}(trp_2)$
$\mathrm{LTL}_{Trace}(trp_1 \mid trp_2)$	$\mathrm{LTL}_{Trace}(trp_1) \vee \mathrm{LTL}_{Trace}(trp_2)$

For example, if in LTL atomic properties could be configuration properties, the safety property specifying that always there is at least one file server component connected, would be written in LTL as follows:

$$\mathrm{G}(\exists \textsf{getServer} \in Interface.(RequiredBy(\textsf{getServer}) =$$
$$\textbf{RequestDispatcher} \wedge \exists i \in Interface.Binding(i) = \textsf{getServer}))$$

For temporal properties, we have:

$\mathrm{LTL}_{Temp}(\textbf{after } E \ tpp)$	$\mathrm{G}(E \Rightarrow \mathrm{LTL}_{Temp}(tpp))$
$\mathrm{LTL}_{Temp}(\textbf{after } E \ trp)$	$\mathrm{G}(E \Rightarrow \mathrm{LTL}_{Trace}(trp))$
$\mathrm{LTL}_{Temp}(\textbf{before } E \ trp)$	$\mathrm{F}(E) \Rightarrow \mathrm{LTL}_{Trace_B}(E, trp)$
$\mathrm{LTL}_{Temp}(trp \ \textbf{until } E)$	$\mathrm{F}(E) \wedge \mathrm{LTL}_{Trace_B}(E, trp)$
$\mathrm{LTL}_{Temp}(trp \ \textbf{unless } E)$	$\mathrm{LTL}_{Trace_C}(E, trp)$
$\mathrm{LTL}_{Temp}(\textbf{between } E_1 \ E_2 \ trp)$	$\mathrm{LTL}_{Temp}(\textbf{after } E_1 \ (trp \ \textbf{until } E_2))$
$\mathrm{LTL}_{Temp}(trp)$	$\mathrm{LTL}_{Trace}(trp)$

where:

$\text{LTL}_{Trace_B}(E, \textbf{always } cp)$	$cp \ \mathsf{U} \ E$
$\text{LTL}_{Trace_B}(E, \textbf{eventually } cp)$	$\neg(\neg cp \ \mathsf{U} \ E)$
$\text{LTL}_{Trace_C}(E, \textbf{always } cp)$	$\text{G}(cp) \vee (cp \ \mathsf{U} \ E)$
$\text{LTL}_{Trace_C}(E, \textbf{eventually } cp)$	$\text{F}(cp) \wedge \neg(\neg P \ \mathsf{U} \ E)$

Remark that a trace property is translated into LTL according to the temporal context in which the property is used, that is why we define two auxiliary functions LTL_{Trace_B} and LTL_{Trace_C}. The LTL_{Trace} function translates a trace property which either does not depend on a temporal property, or is inside an **after** temporal property. The LTL_{Trace_B} function is used to translate a trace property which is inside a **before** or an **until** temporal properties. Finally, the LTL_{Trace_C} function translates a trace property bounded by a **unless** temporal property.

For example, the property specifying that the deviation must always be lower than 50 until the `AddCacheHandler` reconfiguration operation terminates normally can be written in LTL as follows:

$$\text{F}(\texttt{AddCacheHandler } \textbf{normal}) \wedge ($$
$$\exists deviation \in Parameter.(ParameterOf(deviation) = RequestHandler$$
$$\wedge \ deviation < 50) \ \mathsf{U} \ \texttt{AddCacheHandler } \textbf{normal})$$

5   Application to the Fractal Component Model

The Fractal component model [7] is one of the motivations of the present work because of its native support for dynamic architectures. Fractal also provides means for introspection and reconfigurations. Existing implementations for Fractal and its extensions offer a framework to experiment with FTPL-based reconfigurations. This section briefly describes some Fractal features and the existing language support for reconfigurations, before reporting on our experiments.

5.1   Overview of Fractal, FPath and FScript

The Fractal model is a hierarchical and reflective component model intended to implement, deploy and manage software systems [7]. A Fractal component is both a design and a run-time 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.

FPath [9] 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 [9] 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.

5.2 From the FTPL Model to Fractal

As explained above, the architectural model presented in Sec. 3 has been developed to capture the Fractal component model, among other component-based models with reconfigurations, like CCM, GCM, etc. To illustrate our proposals, in this section we give a part of the Http Server example encoded using our model (Fig. 5) as well as its implementation in FractalADL (Fig. 6).

$$
\begin{array}{ll}
Component & = \{\ \textbf{HttpServer}, \textbf{RequestReceiver}, \textbf{RequestHandler}, \ldots\ \} \\
Required & = \{\ \text{getHandler}, \text{getDispatcher}, \text{getCache}, \ldots\} \\
Provided & = \{\ \text{httpRequest}, \text{request}, \ \text{handler}, \ldots\} \\
Parameter & = \{\ \text{load}, \text{deviation}, \ldots\} \\
Type & = \{\ Int, \ Real, \ \} \\
ProvidedBy & = \{\ \text{httpRequest} \mapsto \textbf{HttpServer}, \text{request} \mapsto \textbf{RequestReceiver}, \\
& \qquad \text{handler} \mapsto \textbf{RequestHandler}, \ldots\} \\
RequiredBy & = \{\ \text{getHandler} \mapsto \textbf{RequestReceiver}, \\
& \qquad \text{getDispatcher} \mapsto \textbf{RequestHandler}, \\
& \qquad \text{getCache} \mapsto \textbf{RequestHandler}, \ldots\} \\
ParameterOf & = \{\ \text{load} \mapsto \textbf{RequestHandler}, \text{deviation} \mapsto \textbf{RequestHandler}, \ldots\} \\
TypeOf & = \{\ \text{load} \mapsto Int, \text{deviation} \mapsto Int, \ldots\} \\
ValueOf & = \{\ \text{load} \mapsto 100, \text{deviation} \mapsto 50, \ldots\ \} \\
ChildOf & = \{\ (\textbf{HttpServer}, \textbf{RequestReceiver}), \\
& \qquad (\textbf{HttpServer}, \textbf{RequestHandler}), \ldots\ \} \\
Binding & = \{\ \text{getHandler} \mapsto \text{handler}, \ldots\} \\
Delegate & = \{\ \text{request} \mapsto \text{httpRequest}\ \}
\end{array}
$$

Fig. 5. HttpServer example using Definitions 2 and 3

Let us recall that FractalADL[4] is the architecture description language for Fractal which allows implementing the Fractal component model.

We then use the FScript language to specify the reconfiguration operations presented in Sec. 3, and the FScript tool support to execute them. FScript is focused on the manipulation of architectural concepts and provides complete control of the architecture of the systems modeled in Fractal. Concretely, FScript takes an architecture of a current Fractal configuration and dynamically changes

---

[4] http://fractal.ow2.org/fractaladl/

```
1   <definition name="HttpServer">
2     <interface name="httpRequest" role="server"
           signature="java.lang.Runnable"/>
3     <component name="RequestReceiver">
4       <interface name="request" role="server"
             signature="java.lang.Runnable"/>
5       <interface name="getHandler" role="client"
             signature="Handler"/>
6       <content class="RequestReceiverImpl"/>
7     </component>
8     <component name="RequestHandler">
9       <interface name="handler" role="server" signature="Handler"/>
10      <content class="RequestHandlerImpl"/>
11      <attributes signature="RequestHandlerAttributes">
12        <attribute name="load" value="100"/>
13        <attribute name="deviation" value="50"/>
14      </attributes>
15      <controller desc="primitive"/>
16    </component>
17      ...
18    <binding client="this.httpRequest"
           server="RequestReceiver.request"/>
19    <binding client="RequestReceiver.getHandler"
           server="RequestHandler.handler"/>
20      ...
21  </definition>
```

Fig. 6. HTTP Server example in FractalADL

it according to a FScript file in order to create a new target architecture. The FScript implementation features guarantee that FScript reconfigurations always terminate and keep the system in a consistent and usable state.

For example, we specify the AddCacheHandler reconfiguration in FScript as presented in Fig. 7. This reconfiguration consists in creating a new instance of **CacheHandler** (name) and in specifying its name (set-name). Then, the component is integrated into the architecture (add) and the binding with the component **RequestHandler** is set (bind). Finally, the component **CacheHandler** is started (start).

```
1   action addCache(root)
2   {
3     newCache = new("CacheHandler");
4     set-name($newCache, "CacheHandler");
5     add($root, $newCache);
6     bind($root/child::RequestHandler/interface::getcache, $newCache/
           interface::cache);
7     start($newCache);
8   }
```

Fig. 7. AddCacheHandler Reconfiguration specified in FScript

5.3 Dynamic Verification

We now report on our experiments evaluating the feasibility of a run-time monitoring of FTPL properties. The monitoring on the execution of the architectural reconfiguration model depends on the property to be verified; It is either the satisfiability of a configuration property on one configuration, or the satisfiability of a temporal property on a sequence of component-based system architectures.

Verification of configuration properties. We use the FPath language support to verify a configuration property on one configuration. Indeed, any first order logic formula specifying a configuration property can be translated into an FPath expression, the FPath language having the same expressive power [13].

For example, the *CacheConnected* configuration property from Example 3 can be expressed in FPath by:

```
$HttpServer/child::RequestHandler/interface::getCache/
    ↪ binding::cache/component::CacheHandler
```

Verification of temporal properties. Once the configuration properties are handled thanks to FPath, there is a need to deal with FTPL temporal properties. In [5], it is shown that the monitoring works well on specific safety properties. In FTPL the safety properties are properties containing only the keywords **after**, **before**, **unless** and **always**; The safety properties are also properties containing the **eventually** keyword iff they contain the **before** keyword to bound the **eventually** part. The other properties are liveness properties.

We have studied the feasibility of the safety properties monitoring by developing a controller in Fractal. This controller supervises the properties of interest each time a reconfiguration operation occurs. It retains the configurations appearing during the system execution to build a history and to use it for verification purpose. In order to make the monitoring easier, the controller divides the property into sub-properties and keeps each sub-property until it manages to validate it. For example, for the property **after** AddCacheHandler **called always** *CacheConnected*, the controller tries first to find a configuration where the AddCacheHandler **called** event property holds. Once such a configuration is found, the controller continues with the monitoring of the *CacheConnected* configuration property for all the following configurations.

The FTPL properties can be divided into two classes: the properties dealing with the past and the properties dealing with the future. As it is possible to determine what has happened in the past thanks to the history built by the controller, all the past properties can be monitored. But, due to the run-time verification, there are properties that cannot be ensured by the controller. For any property about the future containing **always**, the controller only ensures that the current configuration does not violate the property at the moment. For any property about the future containing **eventually** key-word, the controller cannot conclude neither.

As explained before, the controller is able to monitor the safety properties about the past, where as for the properties about the future, it only ensures their non-violation by the current configuration. In [5], the monitoring of safety properties necessitates time intervals bounded in the past as well as in the future. To go further within the Fractal-based approach, and to handle liveness properties, a solution would be to exploit a variant specification of the **Loop** clause [11], to which liveness properties can be reduced.

6   Conclusion

In this paper we have developed a theoretical framework for dynamic reconfigurations of component-based systems. As a calculus for expressing and analysing reconfiguration and integrity constraints, we have utilised linear temporal logic, since formulas are interpreted over configuration sequences which naturally represent dynamic behaviour of component-based systems. For the Fractal component model, we have studied the feasibility of monitoring dynamic reconfigurations during system lifetime.

Related work Dynamic reconfiguration of distributed applications is an active research topic [1, 2, 5, 15] motivated by practical applications like those modelled in Fractal [7] or in ArchJava [3].

In the context of dynamic reconfigurations, ArchJava [3] gives means to reconfigure Java architectures, and the ArchJava language guarantees communication integrity at run-time. In the Fractal-based framework, in [14, 15] the authors have defined integrity constraints as architectural invariants specifying the reliability of component-based systems. There are tools to allow the user to ensure the reliability of those reconfigurations at run-time. Those works exploit a graph-based representation of component-based architectures. Our model in Sect. 3 is closely related to the model proposed in [15] for the Fractal component systems but unlike [15], our model lays down only general architectural constraints. In this sense it can be considered as a generalisation of the Fractal-oriented model. Moreover, our model seems to be general enough to give operational semantics to other component-based systems. On the integrity and architectural constraint side, the FTPL logic allows us to specify architectural constraints more complex that architectural invariants in [9].

Among other applications, our proposals aims at an active monitoring of component-based systems. The active monitoring involves interpreting a configuration data set and acting on those data to (re-)configure the system accordingly. This may simply be a validation of the target configuration, or a reconfiguration operation interruption. In [5, 4], Basin et.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). The semantics of MFOTL has been defined with respect to timed temporal structures. Like the model in [5], our model is a first-order structure, but instead of considering a sequence of time stamps, we focus on

reconfiguration operations. Although our main motivation and hence the model are different, their algorithms for monitoring temporal safety properties would be adapted for performing dynamic reconfigurations of component-based systems.

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 *In WICSA'08, February 2008*, pages 23–34, 2008.
4. D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV 2010, UK, July, 2010*, volume 6174 of *LNCS*, pages 1–18, 2010.
5. 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.
6. F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical report RR-5331, INRIA, 2004.
7. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.
8. F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Composition et expression qualitative de politiques d'adaptation pour les composants Fractal. In *GDR GPL 2009*, Toulouse, France, January 2009.
9. 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.
10. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
11. A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with java modelling language. *IET Software*, 2008.
12. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
13. M. Léger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composant*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2009.
14. M. Léger, Th. Ledoux, and Th. Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *ARM'07*, pages 1–6. ACM, 2007.
15. 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. Springer-Verlag, 2010.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
17. B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP 2002, Malaga, Spain, June 10-14, 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 205–230. Springer, 2002.
18. K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *AMAST 2002*, volume 2422 of *LNCS*, pages 334–348, 2002.