Structural Refinement of Components Keeps Temporal Properties over Reconfigurations

Julien Dormoy, Olga Kouchnarenko, Arnaud Lanoix

Structural Refinement of Components Keeps Temporal Properties over Reconfigurations

Julien Dormoy[1], Olga Kouchnarenko[1,3], and Arnaud Lanoix[2]

[1] FEMTO-ST CNRS and University of Franche-Comté, Besançon, France
firstname.name@univ-fcomte.fr
[2] LINA CNRS and Nantes University, Nantes, France
arnaud.lanoix@univ-nantes.fr
[3] INRIA/CASSIS France

Abstract. Dynamic reconfigurations increase the availability and the reliability of component-based systems by allowing their architecture to evolve at runtime. Recently, a linear temporal pattern logic, called FTPL, has been defined to express desired—architectural, event and temporal— properties over dynamic reconfigurations of component systems. This paper is dedicated to the preservation of the FTPL properties when refining components and introducing new reconfigurations. To this end, we use architectural reconfiguration models giving the semantics of component-based systems with reconfigurations, on which we define a new refinement relation. This relation combines: (i) a *structural* refinement which respects the component encapsulation within the architectures at two levels of refinement, and (ii) a *behavioural* refinement which links dynamic reconfigurations of a refined component-based system with their abstract counterparts that were possible before the refinement. The main advantage of the new refinement is that this relation *preserves* the FTPL properties. The main contributions are illustrated on the example of an HTTP server architecture.

1 Introduction

The refinement-based design and development simplifies complex system specification and implementation [1,2]. For component-based systems, it is important in practice to associate a design by refinement with a design by a composition of their components [3,4].

In this paper we propose a refinement of component-based systems with reconfigurations which preserves event and temporal properties. Our main goal is to respect component encapsulation, i.e. the refinement of a component must not cause any changes outside of this component. Moreover, we want the refinement to respect the availability of reconfigurations from an abstract level to a refined one: new reconfigurations handling new components introduced by the refinement must not take control forever, and no new deadlock is allowed. The present paper's contributions are based on our previous works [5,6,7] where the semantics of component-based architectures with dynamic reconfigurations

has been given in terms of labelled transition systems **(1)**. The first contribution of this paper is a definition of a *structural refinement* **(2)** which links two architectures at two development levels: in a refined architecture every refined component must have the same interfaces of the same types as before. This way other components do not see the difference between the refined components and their abstract versions, and thus there is no need to adapt them. The second contribution is the definition of a *reconfiguration refinement relation* **(3)** linking dynamic reconfigurations of a refined component-based system with their abstract counterparts that were possible before the refinement.

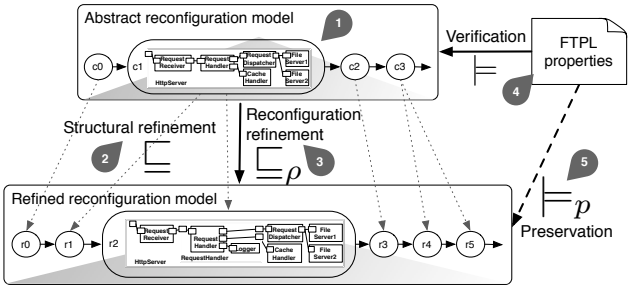

Fig. 1. Verification and preservation through refinement

Moreover, we want the refinement to preserve temporal properties. To express temporal properties over architectural reconfigurations of component-based systems a temporal pattern logic, called FTPL, has been defined [5] **(4)**. FTPL allows expressing architectural invariants, both event and temporal properties involving different kinds of temporal patterns which have been shown useful in practice. The third contribution of this paper consists in proving that the refinement relation—a special kind of simulation—*preserves* **(5)** the FTPL properties: any property verified at a given refinement level is ensured, "for free", at the following refinement levels, provided that the refinement relation holds.

The remainder of the paper is organised as follows. We briefly recall in Sect. 2 the architectural (re-)configuration model and the FTPL syntax and semantics. We then define in Sect. 3 the structural refinement between two architectural configurations, before integrating it into the reconfiguration model refinement. Section 4 shows that the refinement relation preserves FTPL properties. Finally, Section 5 concludes and gives some perspectives.

2 Architectural Reconfiguration Model

This section briefly recalls the architectural reconfiguration model given in [5,6], and the temporal pattern logic for dynamic reconfigurations, called FTPL in [5].

2.1 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 express various links between these basic architectural elements. In this section we sum up formal definitions given in [5,6]. To this end, we consider a graph-based representation in Fig. 2, inspired by the model for Fractal in [8].

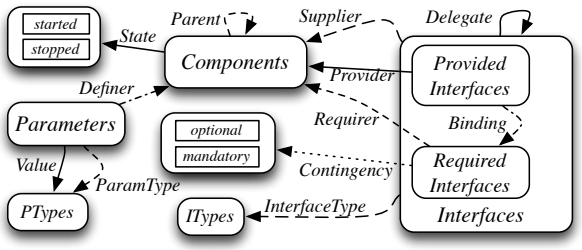

Fig. 2. Architectural elements and relations

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 $Interfaces$;
- $Parameters$ is a set of component parameters;
- $ITypes$ is the set of the types associated with interfaces;
- $PType$ 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 the previously mentioned architectural elements.

- $InterfaceType$ is a total function that associates a type with each interface;
- $Supplier$ is a total function to determine the component of a provided or of a required interface; $Provider$ is a total surjective function which gives the component having at least a provided interface of interest, whereas $Requirer$ is only a total function;
- $Contingency$ is a total function which indicates for each required interface whether 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;
- $Binding$ is a partial function to connect a provided interface with a required one: a provided interface can be linked to only one required interface, whereas a required interface can be the target of one or more provided interfaces. Moreover, two linked interfaces do not belong to the same component, but their corresponding components are sub-components of the same composite component. The considered interfaces must have the same interface type. Also, they have not been involved in a delegation yet;

– *Delegate* describes 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 been involved in a binding yet;
– *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.

Example 1. To illustrate our model, let us consider an example of an HTTP server from [9,8]. The architecture of this server is depicted in Fig. 3. 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:

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

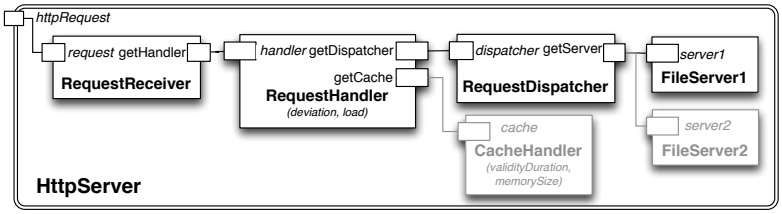

Fig. 3. HTTP Server architecture

We now introduce a set CP of configuration propositions which are constraints on the architectural elements and the relations between them. These constraints are specified using first order (FO) logic formulas over constants $\{\top, \bot\}$, variables in $\mathcal{V}$ to reason on elements of $Elem$, functions and relations from Rel, predicates $\mathcal{S}_{\mathcal{P}} = \{\in, =, \ldots\}$, connectors $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and quantifiers $\exists$, $\forall$ [10]. Then the interpretation of functions, relations, and predicates over $Elem$ is done according to basic definitions in [10] and the model definition in [5].

The configuration properties are expressed at different specification levels. At the component model level, the constraints are common to all the component architectures. Furthermore, 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 2. Let *CacheConnected* be a configuration property defined by

$$\exists\ \text{cache}, \text{getCache} \in Interfaces. \begin{pmatrix} Provider(\text{cache}) = \textsf{CacheHandler} \\ \wedge\ Requirer(\text{getCache}) = \textsf{RequestHandler} \\ \wedge\ Binding(\text{cache}) = \text{getCache} \end{pmatrix}$$

This property expresses that the **CacheHandler** component is connected to the **RequestHandler** component through their respective interfaces.

2.2 Reconfigurations: from a Component Architecture to Another

To make the component-based architecture evolve dynamically, we introduce reconfigurations which are combinations of primitive operations such as instantiation/destruction of components; addition/removal of components; binding/unbinding of component interfaces; starting/stopping components; setting parameter values of components. The normal running of different components also changes the architecture by modifying parameter values or stopping components. Let $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ be a set of evolution operations, where $\mathcal{R}$ is a finite set of reconfiguration operations, and *run* is an action to represent running operations. Given a component architecture and $\mathcal{R}_{run}$, the possible evolutions of the component architecture are defined as a transition system over $\mathcal{R}_{run}$.

Definition 1. *The operational semantics of component systems with reconfigurations is defined by the labelled transition system* $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ *where* $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ *is a set of configurations,* $\mathcal{C}^0 \in \mathcal{C}$ *is a set of initial 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*[4]*, and* $l : \mathcal{C} \rightarrow CP$ *is a total function to label each* $c \in \mathcal{C}$ *with the largest conjunction of* $cp \in CP$ *evaluated to true on* c*.*

Let us note $c \stackrel{ope}{\rightarrow} c'$ when a target configuration $c' = \langle Elem', Rel' \rangle$ is reached from a configuration $c = \langle Elem, Rel \rangle$ by an evolution operation $ope \in \mathcal{R}_{run}$. Given the model $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$, an evolution path (or a path for short) σ of S is a (possibly infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0.(\exists\ ope_i \in \mathcal{R}_{run}.(c_i \stackrel{ope_i}{\rightarrow} c_{i+1} \in \rightarrow))$. We write $\sigma(i)$ 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. Let Σ denotes the set of paths, and Σ^f $(\subseteq \Sigma)$ the set of finite paths.

[4] Actually, $\rightarrow$ is a reconfiguration function because of the architectural model.

Example 3. For the HTTP server, the reconfiguration operations are: AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the **CacheHandler** component; AddFileServer and removeFileServer which are respectively used to add and remove the **FileServer2** component; MemorySizeUp and MemorySizeDown which are respectively used to increase and to decrease the MemorySize value; DurationValidityUp and DurationValidityDown to respectively increase and decrease the ValidityDuration value. A possible evolution path of the HTTP server architecture is given in Fig. 4.

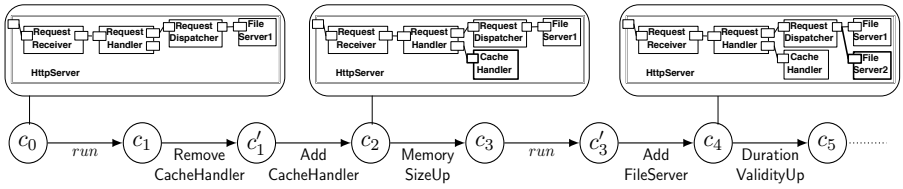

Fig. 4. Part of an evolution path of the HTTP server architecture

2.3 FTPL: a Temporal Logic for Dynamic Reconfigurations

Let us first give the FTPL syntax. Basically, constraints on the architectural elements and the relations between them are specified as configuration propositions in Sect. 2.1. In addition, the language contains events from reconfiguration operations, trace properties and, finally, temporal properties. Let $Prop_{FTPL}$ denote the set of FTPL formulae.

$<temp>$	::=	**after** $<event>$ $<temp>$
	|	**before** $<event>$ $<trace>$
	|	$<trace>$ **until** $<event>$
$<trace>$	::=	**always** cp
	|	**eventually** cp
	|	$<trace> \wedge <trace>$
	|	$<trace> \vee <trace>$
$<event>$	::=	ope **normal**
	|	ope **exceptional**
	|	ope **terminates**

Let $cp \in CP$ be a configuration property, and c a configuration. We say that c satisfies cp, written $c \models cp$, when $l(c) \Rightarrow cp$. We also say that cp is valid on c. Otherwise, we write $c \not\models cp$ when c does not satisfy cp. For example, for the $CacheConnected$ configuration property from Example 2 and the path from Fig. 4, we have $c_2 \models CacheConnected$ whereas $c_1 \not\models CacheConnected$.

Definition 2 (FTPL semantics). *Let $\sigma \in \Sigma$. The FTPL semantics $\Sigma \times Prop_{FTPL} \to \mathbb{B}$ is defined by induction on the form of the formulae as follows:*

For the events:			
	$\sigma(i) \models ope$ **normal**	if	$i > 0 \wedge \sigma(i-1) \neq \sigma(i) \wedge \sigma(i-1) \overset{ope}{\to} \sigma(i) \in \to$
	$\sigma(i) \models ope$ **exceptional**	if	$i > 0 \wedge \sigma(i-1) = \sigma(i) \wedge \sigma(i-1) \overset{ope}{\to} \sigma(i) \in \to$
	$\sigma(i) \models ope$ **terminates**	if	$\sigma(i) \models ope$ **normal** $\vee\ \sigma(i) \models ope$ **exceptional**
For the trace properties:			
	$\sigma \models$ **always** cp	if	$\forall i.(i \geqslant 0 \ \Rightarrow \sigma(i) \models cp)$
	$\sigma \models$ **eventually** cp	if	$\exists i.(i \geqslant 0 \wedge \sigma(i) \models cp)$
	$\sigma \models trace_1 \wedge trace_2$	if	$\sigma \models trace_1 \wedge \sigma \models trace_2$
	$\sigma \models trace_1 \vee trace_2$	if	$\sigma \models trace_1 \vee \sigma \models trace_2$
For the temporal properties:			
	$\sigma \models$ **after** $event\ temp$	if	$\forall i.(i \geqslant 0 \wedge \sigma(i) \models event \Rightarrow \sigma_i \models temp)$
	$\sigma \models$ **before** $event\ trace$	if	$\forall i.(i > 0 \wedge \sigma(i) \models event \Rightarrow \sigma_0^{i-1} \models trace)$
	$\sigma \models trace$ **until** $event$	if	$\exists i.(i > 0 \wedge \sigma(i) \models event \wedge \sigma_0^{i-1} \models trace)$

An architectural reconfiguration model $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ satisfies a property $\phi \in Prop_{FTPL}$, denoted $S \models \phi$, *if* $\forall \sigma.(\sigma \in \Sigma(S) \wedge \sigma(0) \in \mathcal{C}^0 \Rightarrow \sigma \models \phi)$.

Example 4. The FTPL framework allows handling architectural invariants from [9,8]. The following property expresses an architectural constraint saying that at least there is always one file server component connected to **RequestDispatcher**.

$$\textbf{always}\ \left(\exists \mathsf{getServer} \in Interfaces.\ \begin{pmatrix} Requirer(\mathsf{getServer}) = \textbf{RequestDispatcher} \\ \wedge \exists i \in Interfaces.(Binding(i) = \mathsf{getServer}) \end{pmatrix}\right)$$

Example 5. The following temporal property specifies that after calling up the AddCacheHandler reconfiguration operation, the **CacheHandler** component is always connected to **RequestHandler**. In other words, the $CacheConnected$ configuration property from Example 2 holds on all the path after calling up AddCacheHandler:

$$\textbf{after}\ \mathsf{AddCacheHandler}\ \textbf{normal always}\ CacheConnected$$

3 Refinement of Architectural Reconfiguration Models

This section defines a new notion of a structural configuration refinement between two architectural configurations, and then gives the reconfiguration model refinement as defined in the style of Milner-Park's simulation.

3.1 Structural Configuration Refinement

In this section we introduce a *structural* refinement of a component-based architecture. This refinement aims to respect component encapsulation, i.e. the refinement of a component does not cause any changes outside of this component. In fact, the refined component must have the same interfaces of the same types as before. This way other components do not see the difference between the component and its refined version, and thus there is no need to adapt them.

Example 6. Let us illustrate our goal on the example of the HTTP server. We consider the configuration c_A given Fig. 5, and we refine the **RequestHandler** by two new components: **RequestAnalyzer** and **Logger**, to obtain a new *refined* configuration c_R. **RequestAnalyzer** handles requests to determine the values of the deviation and load parameters. **Logger** allows **RequestAnalyzer** to memorise requests to chose either **RequestDispatcher** or **CacheHandler**, if it is available, to answer requests. The "old" **RequestHandler** component becomes a composite component which encapsulates the new components. Its interfaces remain the same as the interfaces of the old component.

Let $c_A = \langle Elem_A, Rel_A \rangle$ and $c_R = \langle Elem_R, Rel_R \rangle$ be two architectural configurations at two—an abstract and a refined—levels of refinement. To distinguish architectural elements at the abstract level and at the refined level, the elements are renamed to have $Elem_A \cap Elem_R = \varnothing$. To define the structural

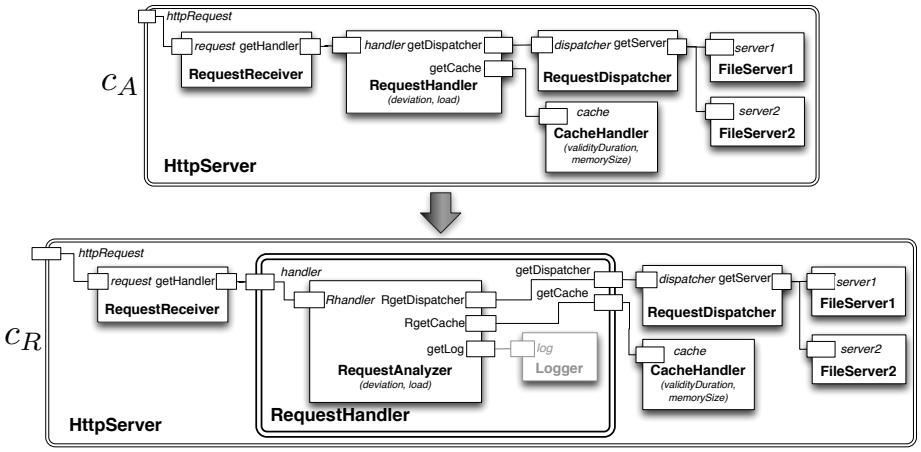

Fig. 5. A refinement of the **HttpServer** component

refinement, we have to link together an abstract and a refined configuration, i.e. express how all the architectural elements and relations are associated with their refined versions: a *gluing* predicate *gp* must be defined as the conjunction of equalities between the abstract and the refined elements and relations.

In addition to this gluing predicate *gp*, component-based structural constraints are necessary to ensure that the proposed refinement respects the component semantics, i.e. which changes are allowed or prescribed during the refinement process. These *architectural constraints*, named *AC*, are defined as the conjunction of the propositions given in Table 1, with the following meanings:

- *In the system parts not concerned by the refinement, all the core entities and all the relations between them remain unchanged through refinement* (constraints (G), (H), (I) and (J));
- *The new elements introduced during the refinement process must satisfy the following constraints:*
  - In the refined architecture the new components must be subcomponents of components existing before refinement (constraint (K));
  - The new interfaces are associated with the new components (constraint (C));
  - The new parameters are associated with the new components (constraint (E));
- *Finally, for the architectural elements existing before and impacted by the refinement, the constraints are as follows:*
  - All the interfaces of the components existing before and detailed during the refinement must be delegated interfaces, these components being composites after refinement (constraints (A), (B) and (L));
  - All the parameters of the components existing before and detailed during the refinement must be associated with the new subcomponents (constraints (D) and (F)).

$$\forall i_A \in Interfaces_A, \exists i_R \in Interfaces_R \cdot \left(\begin{array}{l} gp \Rightarrow (i_A = i_R) \wedge \\ Contingency_A(i_A) = Contingency_R(p_R) \wedge \\ \forall t_A \in ITypes_A. \\ \ InterfaceType_A(i_A) = t_A \Rightarrow \\ \ \exists t_R \in ITypes_R. (InterfaceType_R(i_R) = t_R \wedge gp \Rightarrow (t_A = t_R)\ ) \end{array} \right) \quad \text{(A)}$$

$$\forall i_A \in Interfaces_A, \forall c_A \in Components_A \cdot \left(\begin{array}{l} Supplier_A(i_A) = c_A \Rightarrow \\ \exists i_R \in Interface_R, \exists c_R \in Components_R. \\ \ (Supplier_R(i_R) = c_R \wedge gp \Rightarrow (i_A = i_R \wedge c_A = c_R)\ ) \end{array} \right) \quad \text{(B)}$$

$$\forall i_R \in Interface_R, \forall i_A \in Interface_A \cdot \left(\begin{array}{l} \neg(gp \Rightarrow (i_A = i_R)) \Rightarrow \\ \exists c_R \in Components_R. \\ \ \left(\begin{array}{l} Supplier_R(i_R) = c_R \wedge \\ \forall c_A \in Components_A. \neg(gp \Rightarrow (c_A = c_R)) \end{array} \right) \end{array} \right) \quad \text{(C)}$$

$$\forall p_A \in Parameters_A, \forall t_A \in PTypes_A \cdot \left(\begin{array}{l} ParameterType_A(p_A) = t_A \Rightarrow \\ \exists p_R \in Parameters_R, \exists t_R \in PTypes_R. \\ \ \left(\begin{array}{l} ParameterType_R(p_R) = t_R \\ \wedge\ Value_A(p_A) = Value_R(p_R) \\ \wedge gp \Rightarrow (p_A = p_R \wedge t_A = t_R) \end{array} \right) \end{array} \right) \quad \text{(D)}$$

$$\forall p_R \in Parameters_R, \forall p_A \in Parameters_A \cdot \left(\begin{array}{l} \neg(gp \Rightarrow (p_A = p_R)) \Rightarrow \\ \exists c_R \in Components_R.\ \forall c_A \in Components_A. \\ \ (\ Definer_R(p_R) = c_R \wedge \neg(gp \Rightarrow (c_A = c_R))\ ) \end{array} \right) \quad \text{(E)}$$

$$\forall p_A \in Parameters_A, \forall c_A \in Components_A \cdot \left(\begin{array}{l} Definer_A(p_A) = c_A \Rightarrow \\ \exists p_R \in Parameters_R, \exists c_R \in Components_R. \\ \ \left(\begin{array}{l} Definer(p_R) = c_R \vee (Definer_R(p_R), c_R) \in Parent_R^+ \\ \wedge gp \Rightarrow (p_A = p_R \wedge c_A = c_R) \end{array} \right) \end{array} \right) \quad \text{(F)}$$

$$\forall ri_A \in IRequired_A, \forall pi_A \in IProvided_A \cdot \left(\begin{array}{l} Binding_A(ri_A) = pi_A \Rightarrow \\ \exists ri_R \in IRequired_R, \exists pi_R \in IProvided_R. \\ \ Binding_R(ri_R) = pi_R \wedge gp \Rightarrow (ri_A = ri_R \wedge pi_A = pi_R) \end{array} \right) \quad \text{(G)}$$

$$\forall i_A, i'_A \in Interface_A. \left(\begin{array}{l} Delegate_A(i_A) = i'_A \Rightarrow \\ \exists i_R, i'_R \in Interface_R. \\ \ Delegate_R(i_R) = i'_R \wedge gp \Rightarrow (i_A = i_R \wedge i'_A = i'_R) \end{array} \right) \quad \text{(H)}$$

$$\forall c_A, c'_A \in Components_A, \exists c_R, c'_R \in Components_R \cdot \left(\begin{array}{l} gp \Rightarrow (c_A = c_R \wedge c'_A = c'_R) \\ \wedge\ Parent_A(c'_A) = c_A \Rightarrow Parent_R(c'_R) = c_R \end{array} \right) \quad \text{(I)}$$

$$\forall c_A \in Components_A. \left(\begin{array}{l} \exists c_R \in Components_R. \\ \ gp \Rightarrow (c_A = c_R) \wedge State_A(p_A) = State_R(p_R) \end{array} \right) \quad \text{(J)}$$

$$\forall c_A, c'_A \in Components_A, \forall c_R \in Components_R \cdot \left(\begin{array}{l} (c_A, c'_A) \notin Parent_A \wedge (gp \wedge c'_A \neq c_R) \wedge \\ \exists c'_R \in Components_R. \\ \ gp \Rightarrow (c'_A = c'_R) \wedge Parent_R(c_R) = c'_R \end{array} \right) \quad \text{(K)}$$

$$\forall c_A, c'_A \in Components_A, \forall c'_R \in Components_R \cdot \left(\begin{array}{l} (c_A, c'_A) \notin Parent_A \wedge gp \Rightarrow (c'_A = c'_R) \wedge \\ \exists c_R \in Components_R. \\ \ Parent_R(c_R) = c'_R \wedge \\ \forall i_A \in Interface_A, \forall i_R \in Interface_R. \\ \ \left(\begin{array}{l} gp \Rightarrow (i_A = i_R) \wedge Supplier_R(i_R) = c_R \wedge \\ \exists i'_R \in Interface_R. (i_R = Delegate(i'_R)) \end{array} \right) \end{array} \right) \quad \text{(L)}$$

Table 1. Structural refinement constraints AC

Definition 3 (Structural Configuration Refinement). *Let* $c_A = \langle Elem_A, Rel_A \rangle$ *and* $c_R = \langle Elem_R, Rel_R \rangle$ *be two configurations, gp the gluing predicate and AC the architectural constraints. The configuration* c_R refines c_A *wrt. gp and AC, written* $c_R \sqsubseteq c_A$*, if* $l_R(c_R) \wedge gp \wedge AC \Rightarrow l_A(c_A)$.

3.2 Reconfiguration Models Refinement

As an architecture may dynamically evolve through reconfigurations, it concerns refined architectures, where new non primitive reconfigurations may be introduced to handle the new components. For example, in the refined system presented Fig. 5, a possible new reconfiguration RemoveLogger consists in removing the **Logger** component which does not exist at the abstract level.

We consider the new reconfigurations introduced during the refinement process as being *non observable*: they are called τ-reconfiguration. In addition, we define a one-to-one function fc to link the refined reconfiguration actions with the abstract ones as follows: $fc : \mathcal{R}_{run_R} \setminus \{\tau\} \rightarrow \mathcal{R}_{run_A}$ such that $\forall r_R.(r_R \in \mathcal{R}_{run_R} \setminus \{\tau\} \Rightarrow \exists r_A.(r_A \in \mathcal{R}_{run_A} \wedge fc(r_R) = r_A)$.

Following [11], the refinement relation ρ is defined in the style of Milner-Park [12] as a τ-simulation having the following properties[5]:

1. The new reconfiguration actions renamed by τ should not take control forever: the τ- livelocks are forbidden.
2. Moreover, the new reconfiguration actions should not introduce deadlocks.

Definition 4 (Refinement relation). *Let* $S_A = \langle \mathcal{C}_A, \mathcal{C}_A^0, \mathcal{R}_{run_A}, \rightarrow_A, l_A \rangle$ *and* $S_R = \langle \mathcal{C}_R, \mathcal{C}_R^0, \mathcal{R}_{run_R}, \rightarrow_R, l_R \rangle$ *be two reconfiguration models,* $r \in \mathcal{R}_{run R}$ *and* σ_R *a path of* S_R*. We define the relation* $\rho \subseteq \mathcal{C}_R \times \mathcal{C}_A$ *as the greatest binary relation satisfying the following conditions: structural refinement* ($c_R \sqsubseteq c_A$)*, strict transition refinement* (4.1)*, stuttering transition refinement* (4.2)*, non* τ*-divergence* (4.3)*, non introduction of deadlocks* (4.4)*.*

$$\forall c_A \in \mathcal{C}_A, \forall c_R, c'_R \in \mathcal{C}_R.(c_R \ \rho \ c_A \wedge c_R \xrightarrow{r} c'_R \ \Rightarrow \ \exists c'_A.(c_A \xrightarrow{fc(r)} c'_A \wedge c'_R \ \rho \ c'_A)) \quad (4.1)$$

$$\forall c_A \in \mathcal{C}_A, \forall c_R, c'_R \in \mathcal{C}_R.(c_R \ \rho \ c_A \wedge c_R \xrightarrow{\tau} c'_R \ \Rightarrow \ c'_R \ \rho \ c_A) \quad (4.2)$$

$$\forall k.(k \geq 0 \ \Rightarrow \ \exists k'.(k' > k \wedge \sigma_R(k'-1) \xrightarrow{r} \sigma_R(k') \in \rightarrow_R)) \quad (4.3)$$

$$\forall c_A \in \mathcal{C}_A, \forall c_R \in \mathcal{C}_R.(c_R \ \rho \ c_A \wedge c_R \not\rightarrow \Rightarrow \ c_A \not\rightarrow) \quad (4.4)$$

We say that S_R refines S_A, written $S_R \sqsubseteq_\rho S_A$, if $\forall c_R.(c_R \in \mathcal{C}_R^0 \Rightarrow \exists c_A.(c_A \in \mathcal{C}_A^0 \wedge c_R \ \rho \ c_A))$.

As a consequence of Definition 4, we give an important property of this relation allowing to ensure the existence of an abstract path for any refined path.

Proposition 1. *Let* S_A *and* S_R *be two reconfiguration models such that* $S_R \sqsubseteq_\rho S_A$*. Then,* $\forall c_R.(c_R \in \mathcal{C}_R \Rightarrow \exists c_A.(c_A \in \mathcal{C}_A \wedge c_R \ \rho \ c_A))$.

[5] These features are common to other formalisms, like action systems refinement [13] or LTL refinement [1].

Proof (Sketch). Suppose that c_R can be reached by a path σ_R such that $\sigma_R(0) \in \mathcal{C}_R^0$ and $\sigma_R(i) = c_R$. By Clause (4.3) of Def. 4 σ_R contains a finite number of τ-reconfiguration actions, and σ_R is of the form $\sigma_R(0) \xrightarrow{\tau} \ldots \xrightarrow{\tau} \sigma_R(n_1) \xrightarrow{r_1} \sigma_R(n_1+1) \xrightarrow{\tau} \ldots \xrightarrow{r_n} \sigma_R(i-n_m) \xrightarrow{\tau} \ldots \sigma_R(i)$. Moreover, there is a configuration $c_A \in \mathcal{C}_A^0$ such that $\sigma_R(0)\ \rho\ c_A$. We can then build a path from $c_A = \sigma_A(0)$ such that the configurations of σ_A are linked by transitions labelled by reconfigurations $fc(r_1)\ldots fc(r_n) : \sigma_A = c_A^0 \xrightarrow{fc(r_1)} c_A^1 \xrightarrow{fc(r_2)} \ldots \xrightarrow{fc(r_n)} c_A^n (= \sigma_A(j))$. This way the configuration $\sigma_A(j)$ is reached, and by Clauses (4.1) and (4.2) of Def. 4 we have $\sigma_R(i)\ \rho\ \sigma_A(j)$. □

Example 7. The reconfiguration path of the HTTP server from Fig. 4 can be refined as depicted in Fig. 6, where the abstract configuration c_4 are refined by the configurations r_5 and r_6: the new reconfigurations renamed by τ concern the new component **Logger** introduced during the refinement: it is possible to add or to remove the **Logger** component.

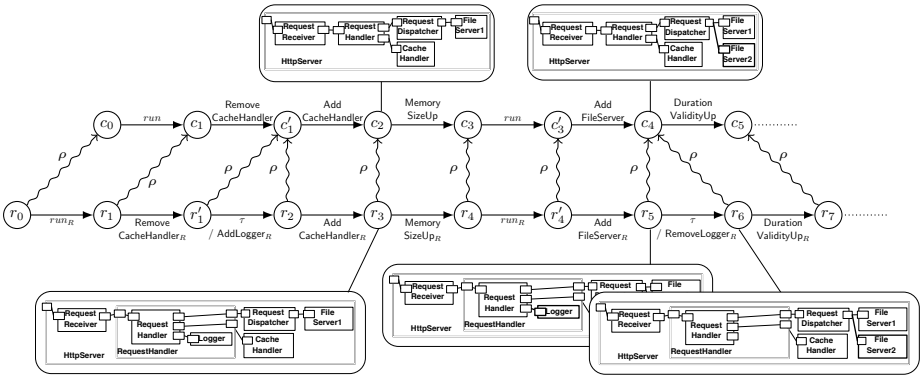

Fig. 6. A refinement of a reconfiguration path of the HTTP server

4 Preservation of FTPL Properties through Refinement

In many formalisms supporting a design by refinement, systems properties are preserved from abstract models to their refined models [14,1,15]. In this section we show that FTPL properties are also preserved through our architectural reconfiguration models refinement. This idea is depicted by Fig. 1.

Let S_A and S_R be two reconfiguration models such that S_R refines S_A. These systems being defined over different sets of architectural elements and reconfigurations, we have to give a new validity definition to be able to deal with an abstract system at a refined level. Actually, we make use of the fc function to link reconfiguration actions, and of the gluing predicate gp to define the validity of a FTPL property by preservation, as follows.

Definition 5 (FTPL semantics by preservation). *Let* $S_R = \langle \mathcal{C}_R, \mathcal{C}_R^0, \mathcal{R}_{run_R}, \rightarrow_R, l_R \rangle$ *and* $S_A = \langle \mathcal{C}_A, \mathcal{C}_A^0, \mathcal{R}_{run_A}, \rightarrow_A, l_A \rangle$ *be two reconfiguration models such that* $S_R \sqsubseteq_\rho S_A$*, gp their gluing predicate and* ρ *their refinement relation. Let* σ_R *be a path of* S_R*,* ϕ_A *a FTPL property over* S_A*. We define the validity of* ϕ_A *on* σ_R *by preservation, written* $\sigma_R \models_p \phi_A$*, by induction on the form of* ϕ_A*:*

$\sigma_R(i) \models_p cp_A$	if	$(\sigma_A(j) \models cp_A \wedge \sigma_R(i)\ \rho\ \sigma_A(j)) \Rightarrow (l_R(\sigma_R(i)) \wedge gp \Rightarrow cp_A)$
$\sigma_R(i) \models_p ope_A$ **normal**	if	$i > 0 \wedge \sigma_R(i-1) \neq \sigma_R(i) \wedge \sigma_R(i-1) \xrightarrow{fc^{-1}(ope_A)}_R \sigma_R(i)$
$\sigma_R(i) \models_p ope_A$ **exceptional**	if	$i > 0 \wedge \sigma_R(i-1) = \sigma_R(i) \wedge \sigma_R(i-1) \xrightarrow{fc^{-1}(ope_A)}_R \sigma_R(i)$
$\sigma_R \models_p$ **always** cp_A	if	$\forall i.(i \geq 0 \Rightarrow \sigma_R(i) \models_p cp_A)$
$\sigma_R \models_p$ **eventually** cp_A	if	$\exists i.(i \geq 0 \wedge \sigma_R(i) \models_p cp_A)$
$\sigma_R \models_p$ **after** $e_A\ tpp_A$	if	$\forall i.(i \geq 0 \wedge \sigma_R(i) \models_p e_A \Rightarrow \sigma_{iR} \models_p tpp_A)$
$\sigma_R \models_p$ **before** $e_A\ trp_A$	if	$\forall i.(i > 0 \wedge \sigma_R(i) \models_p e_A \Rightarrow \sigma_0^{i-1}{}_R \models_p trp_A)$
$\sigma_R \models_p trp_A$ **until** e_A	if	$\exists i.(i > 0 \wedge \sigma_R(i) \models_p e_A \wedge \sigma_0^{i-1}{}_R \models_p trp_A)$

We note $S_R \models_p \phi_A$ when $\forall \sigma_R.(\sigma_R \in \Sigma(S_R) \wedge \sigma_R(0) \in \mathcal{C}^0 \ \Rightarrow\ \sigma_R(0) \models_p \phi_A)$.

Now, we prove that FTPL properties are preserved by the reconfiguration refinement defined in Sect. 3.

Theorem 1 (Preservation of a FTPL property on a path). *Let* S_A *and* S_R *be two reconfiguration models such that* $S_R \sqsubseteq_\rho S_A$*, gp their gluing predicate. Let* ϕ *be a FTPL property. Let* $\sigma_A \in \Sigma(S_A)$ *and* $\sigma_R \in \Sigma(S_R)$ *be two paths. Then we have* $\forall i, j.(0 \leq i \leq j \wedge (\sigma_R(j)\ \rho\ \sigma_A(i)) \wedge \sigma_A \models \phi \Rightarrow \sigma_R \models_p \phi)$.

Proof (Part of Theo. 1). Let $\sigma_R \in \Sigma(S_R)$ be a path refining a path $\sigma_A \in \Sigma(S_A)$ (the proof of Proposition 1 ensures that this path exists). Besides, $ope_R, ope'_R, \ldots \in \mathcal{R}_{run_R}$ label the transitions of S_R, and τ labels each transition introduced during refinement. The proof is done by structural induction on the form of ϕ; only two cases are given here because of lack of room[6].

1. Let us prove that ope_A **normal** is preserved by refinement. By hypothesis, $\sigma_A(i) \models ope_A$ **normal**, and so, by Def. 2 we have (i). As by hypothesis $\sigma_R(j)\ \rho\ \sigma_A(i)$, by construction there is a path σ_A such that $\sigma_R(0)$ refines $\sigma_A(0)$ and where $ope_A = fc(ope_R)$. Consequently, by Proposition 1 we have (ii). Moreover, it implies that there are two configurations $\sigma_R(j)$ and $\sigma_R(l)$ such that $\sigma_R(l)\ \rho\ \sigma(i-1)$ and $\sigma_R(j)\ \rho\ \sigma_A(i)$. There are two cases:
   (a) If $\sigma_R(l) \overset{ope_R}{\rightarrow}_R \sigma_R(j)$ then $l = j-1$, and immediately we can deduce (iii). Then by Def. 5, $\sigma_R \models_p ope_A$ **normal**, and we are done.
   (b) If $\sigma_R(l) \overset{\tau}{\rightarrow}_R \sigma_R(l+1)$, then by Clause (4.2) of Def. 4 we have $\sigma_R(l+1)\ \rho\ \sigma_A(i-1)$, and we can continue with the following configuration of σ_R. According to Clauses (4.3) and (4.4) of Def. 4, the reconfigurations labelled by τ cannot take control forever, and the refinement does not introduce deadlocks. So, there is a configuration $\sigma(l+n)$ such that $\sigma_R(l+n)\ \rho\ \sigma_A(i-1)$ and $\sigma_R(l+n) \overset{ope_R}{\rightarrow} \sigma_R(j)$. We set $l+n = j-1$ and consequently we have (iv). Then, by Def. 5, $\sigma_R \models_p ope_A$ **normal**.

[6] The whole proof can be found in [16].

$$i > 0 \wedge \sigma_A(i-1) \neq \sigma_A(i) \wedge (\sigma_A(i-1) \overset{fc(ope_R)}{\rightarrow}_A \sigma_A(i)) \in \rightarrow_A \quad \text{(i)}$$
$$\forall j.(j \geq 0 \Rightarrow \exists k.(k \geq 0 \wedge \sigma_R(j)\ \rho\ \sigma_A(k))) \quad \text{(ii)}$$
$$j > 0 \wedge \sigma_R(j-1) \neq \sigma_R(j) \wedge (\sigma_R(j-1) \overset{r}{\rightarrow}_R \sigma_R(j)) \in \rightarrow_R \quad \text{(iii)}$$
$$j > 0 \wedge \sigma_R(j-1) \neq \sigma_R(j) \wedge (\sigma_R(j-1) \overset{r}{\rightarrow}_R \sigma_R(j)) \in \rightarrow_R \quad \text{(iv)}$$

2. Let us prove that *trp* **until** *e* is preserved by refinement, with the recurrence hypotheses that *trp* and *e* are preserved by refinement. By hypothesis, we have $\sigma_A \models tpp$ **until** e. So, by Def. 2 we have (v). As by hypothesis $\sigma_R(j)\ \rho\ \sigma_A(i)$, by construction there is a path σ_A such that $\sigma_R(0)$ refines $\sigma_A(0)$ and where $ope_A = fc(ope_R)$. Consequently, by Proposition 1 we have (vi). Moreover, by construction, there is a finite part ${\sigma_0^{j-1}}_R$ of σ_R whose configurations refine the configurations of a corresponding finite part ${\sigma_0^{i-1}}_A$ of σ_A, ensuring (vii). By recurrence hypotheses, *trp* and *e* are preserved by refinement. So, we have (viii). Then, by Def. 5, $\sigma_R \models_p trp$ **until** e.

$$\exists i.(i > 0 \wedge \sigma_A(i) \models e \Rightarrow {\sigma_0^{i-1}}_A \models trp) \quad \text{(v)}$$
$$\forall j.(j \geq 0 \Rightarrow \exists k.(k \geq 0 \wedge \sigma_R(j)\ \rho\ \sigma_A(k))) \quad \text{(vi)}$$
$$\forall k.(0 \leq k < j \Rightarrow \exists k'.(0 \leq k' < i \wedge {\sigma_0^{j-1}}_R(k)\ \rho\ {\sigma_0^{i-1}}_A(k'))) \quad \text{(vii)}$$
$$\exists j.(j > 0 \wedge \sigma_R(j) \models_p e \Rightarrow {\sigma_0^{j-1}}_R \models_p trp) \quad \text{(viii)}$$

□

We are ready to generalise Theorem 1 from paths to reconfiguration models.

Theorem 2 (Preservation of a FTPL property by refinement). *Let* $S_A = \langle \mathcal{C}_A, \mathcal{C}_A^0, \mathcal{R}_{run_A}, \rightarrow_A, l_A \rangle$ *and* $S_R = \langle \mathcal{C}_R, \mathcal{C}_R^0, \mathcal{R}_{run_R}, \rightarrow_R, l_R \rangle$ *be two reconfiguration models such that* $S_R \sqsubseteq_\rho S_A$*. Let* ϕ *be a FTPL property. If* $S_A \models \phi$ *then* $S_R \models_p \phi$*.*

Proof. Immediate. If $S_R \sqsubseteq_\rho S_A$ then $\forall \sigma_R.(\sigma_R \in \Sigma(S_R) \wedge \sigma_R(0) \in \mathcal{C}_R^0 \Rightarrow \exists \sigma_A.(\sigma_A \in \Sigma(S_A) \wedge \sigma_A(0) \in \mathcal{C}_A^0 \wedge \sigma_R(0)\ \rho\ \sigma_A(0)))$. Moreover, if $S_A \models \phi$ then by definition $\forall \sigma_A.(\sigma_A \in \Sigma(S_A) \Rightarrow \sigma_A \models \phi)$. The reconfiguration relations of both S_R and S_A being functionnal, there is no abstract path different from σ_A which could be refined by σ_R. We then can apply Theorem 1. □

Example 8. For our running example of the HTTP server, let us consider again the path refinement in Fig. 6. In this refinement, the **RequestHandler** component is refined as depicted in Fig. 5. Let us consider again the temporal property from Example 5:

$$\sigma \models \textbf{after}\ \mathsf{AddCacheHandler}\ \textbf{normal always}\ CacheConnected$$

It is easy to see that this property is valid on the abstract path depicted in Fig. 6. Moreover, as presented in this figure, the ρ refinement relation holds between the configurations of the illustrated part of the refined path and the corresponding part of the abstract path. Consequently, this property is also valid by preservation on the refined path depicted in Fig. 6.

5 Conclusion

In this paper, we have enriched a theoretical framework for dynamic reconfigurations of component architectures with a new notion of a structural refinement of architectures, which respects the component encapsulation. Then we have integrated this structural refinement into a behavioural refinement relation for dynamic reconfigurations defined in the style of Milner-Park's simulation [12] between reconfiguration models. Afterwards, we have shown that this refinement relation preserves the FTPL properties—architectural invariants, event properties and temporal properties involving different kinds of temporal patterns shown useful in practice. The preservation means that any FTPL property expressed and established for an abstract system is also established for the refined counterparts, provided that the refinement relation holds. This way we ensure the system's consistency at different refinement levels, and we free the specifier from expressing and verifying properties at these levels with new details, components, reconfigurations.

To check the structural refinement, we plan to pursue further and to extend our previous work on the verification of the architectural consistency through reconfigurations [6]. The structural refinement constraints in Table 1 could be formalised and validated in a similar manner. Another solution would be to exploit the architectural description language (ADL) describing component architectures in XML. It becomes possible then to use XML tools for checking the structural refinement between two component architectures.

To conclude, this work on property preservation is used as a hypothesis for our running work on the runtime FTPL verification [7]. We have reviewed FTPL from a runtime point of view [7] by introducing a new four-valued logic, called RV-FTPL, characterising the "potential" (un)satisfiability of the architectural and temporal constraints: 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. We intend to accompany this work with a runtime checking of a "potential" reconfiguration model refinement using the proposals in [17,18].

References

1. Kesten, Y., Manna, Z., Pnueli, A.: Temporal verification of simulation and refinement. In: A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium. Volume 803 of LNCS., Springer (1994) 273–346
2. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6) (2010) 447–466
3. de Alfaro, L., Henzinger, T.: Interface-based design. In: Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry 195, Springer (2005) 83–104
4. Mikhajlov, L., Sekerinski, E., Laibinis, L.: Developing components in the presence of re-entrance. In: Wold Congress on Formal Methods. FM'99, London, UK, Springer-Verlag (1999) 1301–1320

5. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: FACS 2010, 7th Int. Ws. on Formal Aspects of Component Software. Volume 6921 of LNCS., Springer (2012) 200–217
6. Lanoix, A., Dormoy, J., Kouchnarenko, O.: Combining proof and model-checking to validate reconfigurable architectures. In: FESCA 2011. ENTCS (2011)
7. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Runtime verification of temporal patterns for dynamic reconfigurations of components. In: FACS 2011. Volume * of LNCS., Springer (2011) ***–*** To appear.
8. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in a reflective component model. In: CBSE 2010. Volume 6092 of LNCS. (2010) 74–92
9. David, P.C., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. Annales des Télécommunications **64**(1-2) (2009) 45–63
10. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press, Cambridge (1978)
11. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Ready-simulation is not ready to express a modular refinement relation. In: Fondamental Aspects of Software Engineering 2000, FASE'2000. Volume 1783 of LNCS. (2000) 266–283
12. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)
13. Butler, M.J.: Stepwise refinement of communicating systems. Sci. Comput. Program. **27**(2) (1996) 139–173
14. Pnueli, A.: System specification and refinement in temporal logic. In: Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science, London, UK, Springer-Verlag (1992) 1–38
15. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3) (1994) 872–923
16. Dormoy, J.: Contributions à la spécification et à la vérification des reconfigurations dynamiques dans les systèmes à composants. PhD thesis, Université de Franche-Comté, France (December 2011)
17. Elmas, T., Tasiran, S.: Vyrdmc: Driving runtime refinement checking with model checkers. ENTCS **144** (May 2006) 41–56
18. Tasiran, S., Qadeer, S.: Runtime refinement checking of concurrent data structures. ENTCS **113** (January 2005) 163–179