Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 279 (2) (2011) 43–57

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Combining Proof and Model-checking to Validate Reconfigurable Architectures

Arnaud Lanoix[1]

LINA, Nantes University, Nantes, France

Julien Dormoy[2] Olga Kouchnarenko[2]

LIFC, University of Franche-Comté, Besançon, France

Abstract

This paper deals with the formal specification and verification of dynamic reconfigurations of component-based systems. To validate such complex systems, there is a need to check model consistency and also to ensure that dynamic reconfigurations satisfy architectural and integrity constraints, invariants, and also temporal constraints over (re)configuration sequences. As architectural constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we propose to associate proof and model-checking within the well-established B method, to support the modelling of such systems and the (partial-)validation of their dynamic reconfigurations. The objective of the paper is twofold. First, given a hierarchical B model of component-based architectures, we validate it by proving its consistency. Second, given linear temporal logic formulae expressing the desirable dynamic behaviour of the system, we validate reconfigurable system architectures by using bounded model-checking tools supporting the B method. The main contributions are illustrated on the example of a HTTP server architecture.

Keywords: Component, Architecture, Reconfiguration, Proof, Model-checking, B method

1 Introduction

This article is dedicated to an automatic checking of dynamic reconfigurations of component-based systems whose development provides significant advantages like portability, adaptability, re-usability, etc. Dynamic reconfiguration of distributed applications is an active research topic [2,3,17] motivated by practical distributed applications like, e.g., those modelled in Fractal [9]. In many recent works, the idea of using temporal logics to specify dynamic reconfigurations and to manage applications at runtime has been explored [7,15,12].

[1] Email: arnaud.lanoix@univ-nantes.fr

[2] Email: {jdormoy,okouchnarenko}@lifc.univ-fcomte.fr

doi:10.1016/j.entcs.2011.11.011

In [12], a formal semantics of component-based systems architectures with reconfigurations together with a linear time temporal logic over (re)configuration sequences have been proposed. This logic, called FTPL, is based on architectural constraints and on event properties. To validate such complex systems, there is a need to check model consistency and also to ensure that dynamic reconfigurations satisfy architectural and integrity constraints, invariants, and also FTPL constraints. As these constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we propose to combine proof and model-checking techniques within the well-established B method [1].

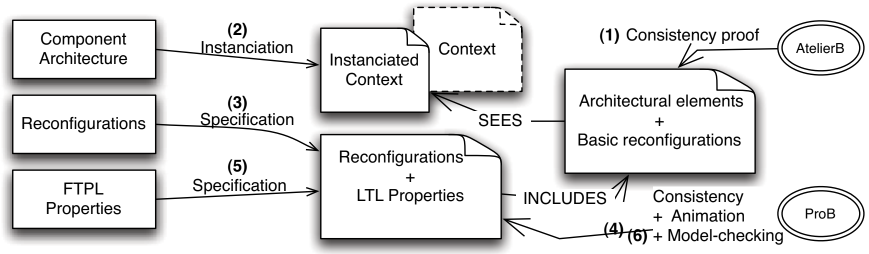


Fig. 1. Principle and contributions

Let us explain our current B-based validation approach and contributions on Fig. 1. First of all, we propose to model components and basic dynamic reconfigurations using the B formal framework. It allows us to define and to validate a generic B model for component architectures. We use the AtelierB tool to interactively prove the architectural constraints consistency **(1)**. Then this generic architecture is instantiated to represent an architecture under consideration **(2)**; The dynamic architectural reconfigurations are specified by using the previously defined primitive B operations **(3)**. Note that we do not take into consideration specification of the controller which would use these dynamic reconfigurations (through adaptation policies as instance). Consequently, the validation of the instantiated architectural model cannot be done using interactive proof. Nevertheless, we perform a partial validation of these dynamic reconfigurations through animations of samples of the running model, thanks to the ProB model-checker features **(4)**. In addition, temporal properties over configuration sequences expressed in FTPL are translated into LTL **(5)**. These properties can be checked with the ProB model-checker **(6)**.

The remainder of the paper is organised as follows. After giving a motivating example in Sect. 2, the B method and its tools supports are introduced in Sect. 3. We formally define a generic B model of component architectures in Sect. 4. This model is then instantiated to validate a particular architecture in Sect. 5. To automatically verify temporal properties, Section 6 introduces FTPL and gives its translation into LTL. Finally, Section 7 concludes before discussing related work.

2 Motivating Example

To motivate and to illustrate our approach, let us consider an example of an HTTP server from [10]. The architecture of this server is displayed in Fig. 2.

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

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

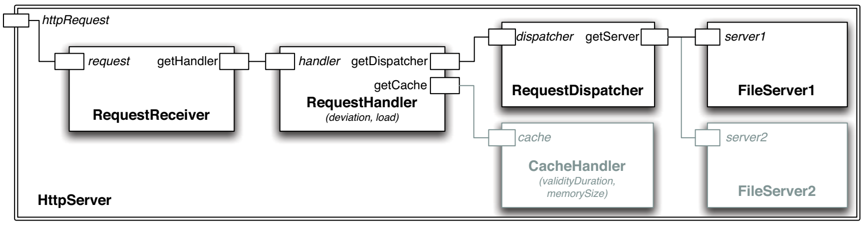


Fig. 2. HTTP Server architecture

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

(i) AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the **CacheHandler** component when the deviation value increased/decreased around 50;

(ii) AddFileServer and removeFileServer which are respectively used to add and remove the **FileServer2** component;

(iii) MemorySizeUp and MemorySizeDown which are respectively used to increase and to decrease the MemorySize value;

(iv) DurationValidityUp and DurationValidityDown to respectively increase and decrease the ValidityDuration value.

As an illustration, we specify the AddCacheHandler reconfiguration expressed in the FScript language [11]. When the deviation value exceeds 50, the reconfiguration consists in instantiating a **CacheHandler** component. Then, the component is integrated into the architecture, and the binding with the required interface of **RequestHandler** is established. Finally, the component **CacheHandler** is started.

```
action AddCacheHandler(root)
if(value($context/child:∈RequestHandler/attribute:∈deviation) > 50){
  newCache = new("CacheHandler");
  add($root, $newCache);
  bind($root/child:∈RequestHandler/interface:∈getcache, $newCache/interface:∈cache);
  start($newCache);
}
```

3 Proof-based Approach: the B Method

B is a formal software development method used to model systems and to reason about their development [1]. When building a B model, the principle is to express system properties which are always true after each evolution step of the model, the evolution being specified by the B operations. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on set theory, relations and first-order logic. Constraints are specified in the **INVARIANT** clause of the model, and its evolution is specified by the operations in the **OPERATIONS** clause. Let us assume here that the initialisation is a special kind of operation. In this setting, the verification of a B model consists in verifying that each operation—assuming its precondition and the invariant hold—satisfies the **INVARIANT**, i.e. the model is *consistent*. A strength of the B method is its stepwise refinement feature: each refinement makes a model more deterministic and also more precise by introducing programming language-like features.

Tool supports, such as B4free or AtelierB[3], automatically generate proof obligations (POs) to ensure the consistency in sense of B [1]. Some of them are obvious POs whereas the other POs have to be proved interactively if it was not done fully automatically by the different provers embedded into AtelierB. Another tool, called ProB[4], allows the user to animate B specifications for their debugging and testing. On the verification side, ProB contains a constraint-based checker and a LTL bounded model-checker with particular features; Both can be used to detect various errors in B specifications [18,19].

4 Specifying a General Architectural Model with B

In [12], we have defined a configuration to be a set of architectural elements (components, interfaces and parameters) together with a relation to structure and to link them, through a graph-based representation. The model we have proposed was inspired by the model in [16,17] given for Fractal. Unlike [16,17], 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* or *composite* components. Only primitive components can have some attributes used as configuration parameters.

Component-based models must provide mechanisms for systems to be dynamically adapted—through their reconfigurations—to their environments during their

[3] http://www.b4free.com / http://www.atelierb.eu
[4] http://www.stups.uni-duesseldorf.de/ProB

lifetime. These dynamic reconfigurations may happen because of architectural modifications specified in primitive operations. Notice that reconfigurations are not the only manner to make an architecture evolve. The normal running of different components also changes the architecture by modifying parameter values or stopping components, for instance.

In this paper, we give a B specification of the generic model for component architectures in [12]. Firstly, we express all the architectural constraints between the architectural elements as B properties and invariants; secondly, we model the primitive reconfiguration operations as B operations; thirdly, we prove the consistency of this architectural B model.

4.1 Specifying the Architectural Configurations with B

The architectural elements we consider are the core entities of a component-based system: COMPONENTS, INTERFACES, INTERFACE_TYPE, and the component PARAMETERS; and relations over them to express various links between these architectural elements. Some of these relations do not evolve during the system reconfigurations. They are then defined as B **CONSTANTS**, and architectural constraints over them are expressed in the **PROPERTIES** clause.

```
SETS
    COMPONENTS ; INTERFACES ; INTERFACE_TYPE ; PARAMETERS
CONSTANTS
    ProvidedInterfaces , RequiredInterfaces , InterfaceType , Provider , Requirer ,
    Contingency, Definer
PROPERTIES
    ProvidedInterfaces ⊆ INTERFACES
    ∧ RequiredInterfaces ⊆ INTERFACES
    ∧ ProvidedInterfaces ∪ RequiredInterfaces = INTERFACES
    ∧ ProvidedInterfaces ∩ RequiredInterfaces = ∅
    ∧ InterfaceType ∈ INTERFACES → INTERFACE_TYPE
    ∧ Provider ∈ ProvidedInterfaces ↠ COMPONENTS
    ∧ Requirer ∈ RequiredInterfaces → COMPONENTS
    ∧ Contingency ∈ RequiredInterfaces → CONTINGENCY
    ∧ Definer ∈ PARAMETERS → COMPONENTS
```

The ProvidedInterfaces and RequiredInterfaces are defined to be subsets of INTERFACES. Their union is disjunctive. InterfaceType is a total function that associates a type with each required and provided interface. Provider is a total surjective function which gives the component having at least a provided interface, whereas Requirer is only a total function. Contingency is a total function which indicates for each required interface if it is mandatory or optional. Definer is a total function which gives the component of a considered parameter.

The other architectural relations we consider may evolve. They are defined as B **VARIABLES**, and architectural constraints over them are expressed in the **INVARIANT** clause.

```
VARIABLES
    InstantiatedComponents, Parent, Binding, ProvDelegate, ReqDelegate,
    State, Value
INVARIANT
    InstantiatedComponents ⊆ COMPONENTS
    ∧ Parent ∈ InstantiatedComponents ⇸ InstantiatedComponents
    ∧ ran(Parent) ∩ ran(Definer) = ∅
    ∧ closure1(Parent) ∩ id(InstantiatedComponents) = ∅
```

InstantiatedComponents is a subset of COMPONENTS. Parent is partial function linking sub-components to the corresponding composite component. Composite components have no parameter, and a sub-component must not be a composite including its parent component, and so on.

```
    ∧ Binding ∈ ProvidedInterfaces ⇸ RequiredInterfaces
    ∧ ∀ (iprov , ireq ) . ( iprov↦ireq ∈ Binding ⇒ (
            Provider( iprov ) ∈ InstantiatedComponents
            ∧ Requirer( ireq ) ∈ InstantiatedComponents
            ∧ Provider( iprov ) ≠ Requirer( ireq )
            ∧ Parent(Provider( iprov )) = Parent(Requirer( ireq ))
            ∧ InterfaceType( iprov ) = InterfaceType( ireq ) ) )
    ∧ dom(Binding) ∩ dom(ProvDelegate) = ∅
    ∧ ran(Binding) ∩ dom(ReqDelegate) = ∅
```

Binding is a partial function which connects together a provided interface and a required one: a provided interface can be linked to only one required interface, whereas a required interface can be the target of more than one provided interface. Moreover, two linked interfaces do not belong to the same component, but their corresponding instantiated components are sub-components of the same composite component. The considered interfaces must have the same interface type, and they have not yet been involved in a delegation.

```
    ∧ ProvDelegate ∈ ProvidedInterfaces ⤔ ProvidedInterfaces
    ∧ ∀ (isub , isuper ) . ( isub↦ isuper ∈ ProvDelegate ⇒ (
            isub ≠ isuper
            ∧ Provider( isub ) ∈ InstantiatedComponents
            ∧ Provider( isuper ) ∈ InstantiatedComponents
            ∧ Parent(Provider( isub )) = Provider( isuper )
            ∧ InterfaceType( isub ) = InterfaceType( isuper ) ) )
    ∧ ReqDelegate ∈ RequiredInterfaces ⤔ RequiredInterfaces
    ∧ ∀ (isub , isuper ) . ( isub↦ isuper ∈ ReqDelegate ⇒ (
            isub ≠ isuper
            ∧ Requirer( isub ) ∈ InstantiatedComponents
            ∧ Requirer( isuper ) ∈ InstantiatedComponents
            ∧ Parent(Requirer( isub )) = Requirer( isuper )
            ∧ InterfaceType( isub ) = InterfaceType( isuper ) ) )
```

ProvDelegate and ReqDelegate express delegation links and are similarly defined. They are both partial bijections that associate a provided (resp. required) interface of a sub-component with a provided (resp. required) interface of its composite: the parent of the component which provides (resp. requires) isub must be the provider (res. requirer) of isuper. Finally, both interfaces must have the same type, and they have not yet been involved in a binding.

```
   ∧ State ∈ InstantiatedComponents → STATE
   ∧ ∀ (ireq ) . ( ( ireq ∈ RequiredInterfaces
                      ∧ Contingency(ireq) = mandat∨y
                      ∧ Requirer( ireq ) ∈ InstantiatedComponents
                      ∧ State(Requirer( ireq )) = started )
            ⇒ ( ireq ∈ ran(Binding) ∨ ireq ∈ dom(ReqDelegate) ) )
    ∧ Value ∈ PARAMETERS → INT
```

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.

4.2 Modelling the Dynamic Reconfigurations with B

Once a configuration-based model is given, the primitive *reconfiguration* operations can be specified as B operations of this B model. Namely, we specify:

- instantiate (newComponent) and delete (component) to instantiate/destroy a component;
- add(subComponent,composite) and remove(subComponent) to add/remove sub-components to/from a composite;
- bind(iprov , ireq) and unbind(iprov) to bind/unbind component interfaces;
- delegate(isub , isuper) and undelegate(isub) to delegate/undelegate component interfaces;
- start (component) and stop(component) to start/stop components;
- set (parameter, newValue) to set a new parameter value.

Let us detail some of these B operations. The example below defines in B the primitive reconfiguration operation adding a subcomponent to a composite component:

```
add(subcomponent, composite) =
PRE subcomponent ∈ COMPONENTS ∧ composite ∈ COMPONENTS THEN
  SELECT
    subcomponent ∈ InstantiatedComponents
    ∧ composite ∈ InstantiatedComponents
    ∧ subcomponent ≠ composite
    ∧ composite ∉ ran(Definer)
    ∧ subcomponent ∉ dom(Parent)
    ∧ subcomponent↦composite ∉ Parent
    ∧ composite↦subcomponent ∉ closure1(Parent)
    ∧ ∀ (iprov) . ( ( iprov ∈ ProvidedInterfaces ∧ Provider( iprov ) = subcomponent )
        ⇒ ( iprov ∉ dom(Binding) ∧ iprov ∉ dom(ProvDelegate) ) )
    ∧ ∀ (ireq) . ( ( ireq ∈ RequiredInterfaces ∧ Requirer( ireq ) = subcomponent )
        ⇒ ( ireq ∉ ran(Binding) ∧ ireq ∉ dom(ReqDelegate) ) )
  THEN
    Parent(subcomponent) := composite
  END
END ;
```

The add(subComponent,composite) operation must establish that the both components are instantiated components, composite is a composite component (i.e. a component without parameters). Moreover, subComponent is not a sub-component of another composite nor is already used: none of its interfaces is bound or delegated. Finally, the modification does not introduce a cycle into Parent.

```
bind( iprov , ireq ) =
PRE iprov ∈ INTERFACES ∧ ireq ∈ INTERFACES THEN
  SELECT
    iprov ∈ ProvidedInterfaces
    ∧ ireq ∈ RequiredInterfaces
    ∧ Provider( iprov ) ∈ InstantiatedComponents
    ∧ Requirer( ireq ) ∈ InstantiatedComponents
    ∧ iprov ∉ dom(Binding)
    ∧ iprov ∉ dom(ProvDelegate)
    ∧ ireq ∉ dom(ReqDelegate)
    ∧ InterfaceType( iprov ) = InterfaceType( ireq )
    ∧ iprov↦ireq ∉ Binding
  THEN
    Binding( iprov ) := ireq
  END
END ;
```

The binding of component interfaces is expressed by bind(iprov , ireq): this operation must establish that the considered interfaces are correct, i.e. they are provided (resp. required) interfaces of instantiated components, they are not bound nor delegated, and their types are compatible. Furthermore, the binding has not been done yet.

```
unbind(iprov) =
PRE iprov ∈ INTERFACES THEN
  SELECT
    iprov ∈ ProvidedInterfaces
    ∧ Provider(iprov) ∈ InstantiatedComponents
    ∧ State(Provider(iprov)) = stopped
    ∧ ∃ (ireq) . (
        ireq ∈ RequiredInterfaces
        ∧ Requirer(ireq) ∈ InstantiatedComponents
        ∧ State(Requirer(ireq)) = stopped
        ∧ iprov ≠ ireq
        ∧ InterfaceType(iprov) = InterfaceType(ireq)
        ∧ iprov↦ireq ∈ Binding )
  THEN
    Binding := {iprov} ⩤ Binding
  END
END ;
```

The unbinding primitive operation is specified as follows: this operation expresses as precondition that the considered interface is provided by an instantiated component. This provider must be stopped. Moreover, a required interface bound with the considered interface, must exist. Then, the considered interface is removed from Binding.

```
stop(component) =
PRE component ∈ COMPONENTS THEN
  SELECT
    component ∈ InstantiatedComponents ∧ State(component) = started
  THEN
    State := State <+ ({component} ∪ dom(closure1(Parent) ▷ {component})) × {stopped}
  END
END ;
```

When considering a started component, applying stop(component) changes to stopped the state of the considered component and the states of all its sub-components, if they exist; and so on.

Once these primitive reconfiguration operations are specified, more complex reconfiguration operations can be written, as explained in Sect. 2.

4.3 Validating the Architectural B Model

We use AtelierB to validate the consistency of our generic B model for component architectures. The tool generates proof obligations (POs) to check the consistency of all the architectural constraints expressed in the **INVARIANT** and the fact that each B reconfiguration operation respects these architectural constraints.

	PO	Proved	Unproved (closure1())	% proved
Initialisation	30	30	0	100
instantiate	20	18	2 (1)	90
delete	27	21	6	77
add	18	14	4 (3)	77
remove	18	17	1 (1)	94
bind	9	4	5	44
unbind	9	8	1	88
start	3	0	3 (3)	0
stop	3	0	3 (3)	0
set	2	2	0	100
TOTAL	**139**	**114**	**25 (11)**	**82**

The AtelierB generates 139 POs. Notice that the work on POs is in progress: at this time, 114 of them are automatically or interactively discharged. The proof of 25 POs remains to be done.

That does not mean that they are false, but that the AtelierB provers simply don't succeed in demonstrating the rule: it may be due to the fact that some B operators are not handled in an efficient manner by the provers (like closure1(), for 11 unproved POs), or due to the fact that the heuristics used for the proof are not efficient enough in this case. An effort remains to be done to manually demonstrate the unproved POs.

5 Validating a Specific Architecture

As summarized in Fig. 1, our approach consists in instantiating the generic architectural B model to specify and to verify a particular architecture. Primitive reconfiguration operations are used to give the complex reconfiguration operations corresponding to the running example.

As explained in Section 1, the instanciation of the general model cannot be checked with the proof approach. Indeed, we don't consider any specification of the controller to manage the dynamic reconfigurations. Nevertheless, to partially validate the model, we apply bounded model-checking to validate the instantiated model.

5.1 Instantiating a Running Architecture

To instantiate a specific architecture using the generic B model, we just give values to all the previously defined sets, constants and variables in Subsect. 4.1, to make them represent the running architecture.

Let us now specify the different manipulated basic architectural elements corresponding to the HTTP server example:

```
COMPONENTS     = { HttpServer, RequestReceiver, RequestHandler, CacheHandler,
                          RequestDispatcher, FileServer1 , FileServer2 } ;
INTERFACES     = { httpRequest, request, getHandler, handler, getDispatcher,
                          getCache, cache, dispatcher , getServer , server1 , server2} ;
INTERFACE_TYPE = { Trequest, Thandler, Tdispatcher, Tcache, Tserver} ;
PARAMETERS     = { deviation, load, validityDuration , mem∨ySize }
```

The **PROPERTIES** clause is extended to state values of the architectural relations:

```
∧ ProvidedInterfaces = { httpRequest,request , handler ,cache, dispatcher , server1 , server2 }
∧ RequiredInterfaces = { getHandler, getDispatcher , getCache, getServer }
∧ Contingency        = { getHandler↦mandat∨y, getDispatcher↦mandat∨y,
                         getCache↦optional, getServer↦mandat∨y }
∧ InterfaceType      = { httpRequest↦Trequest, request↦Trequest, handler↦Thandler,
                         getHandler↦Thandler, getDispatcher↦Tdispatcher,
                         getCache↦Tcache, cache↦Tcache, dispatcher↦Tdispatcher,
                         getServer↦Tserver, server1↦Tserver, server2↦Tserver}
∧ Provider           = { httpRequest↦HttpServer, request↦RequestReceiver,
                         handler↦RequestHandler, cache↦CacheHandler, server1↦FileServer1,
                         dispatcher↦RequestDispatcher, server2↦FileServer2 }
∧ Requirer           = { getHandler↦RequestReceiver, getDispatcher↦RequestHandler,
                         getCache↦RequestHandler, getServer↦RequestDispatcher }
∧ Definer            = { deviation↦RequestHandler, load↦RequestHandler,
                         validityDuration ↦CacheHandler, mem∨ySize↦CacheHandler }
```

Finally, we initialise the remaining architectural relations as follows:

```
∧ InitComponents   = { HttpServer, RequestReceiver, RequestHandler, FileServer1 ,
                       RequestDispatcher}
∧ InitParent       = { RequestReceiver↦HttpServer, RequestHandler↦HttpServer,
                       RequestDispatcher↦HttpServer, FileServer1↦HttpServer }
∧ InitBinding      = { handler↦getHandler, dispatcher↦getDispatcher,
                       server1 ↦getServer }
∧ InitProvDelegate = { request↦httpRequest }
∧ InitReqDelegate  = ∅
∧ InitState        = { HttpServer↦started, RequestReceiver↦started,
                       RequestHandler↦started, RequestDispatcher↦stopped,
                       FileServer1 ↦stopped }
∧ InitValue        = { deviation↦49, load↦75, validityDuration↦2, mem∨ySize↦100 }
```

5.2 Modelling the Running Reconfigurations with B

After having specified the primitive reconfigurations in Subsect. 4.2, we can write more complex reconfiguration operations calling the primitive ones, by composing them sequentially and by using **SELECT**, **IF** and/or **WHILE** statements.

```
AddCacheHandler =
SELECT (Value(deviation) > 50)
THEN
    instantiate (CacheHandler) ;
    add(CacheHandler, HttpServer) ;
    bind(cache, getCache) ;
    start (CacheHandler)
END ;
```

The AddCacheHandler reconfiguration from Sec. 2 can be expressed using the B reconfiguration primitives as depicted here. All the reconfigurations of the running example can be expressed by B operations in a similar manner.

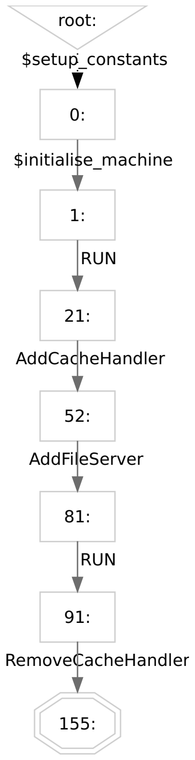
In addition, the normal running of different components could also change the architecture by modifying, for example, parameter values. To handle this behaviour, a solution would be to define an abstraction of the running of the system as a (set of) B operation(s). For our example, a B operation, called RUN, is added: basically, it changes the values of the (load) and (deviation) parameters of the **RequestHandler** component.

5.3 Validating the Running Architectural B Model

Addressing the validation of the instantiated B model using a proof process is not possible at this step of the work. Indeed, we have not taken into consideration a specification of the necessary associated controller to manage dynamic reconfigurations (by the mean of adaptation policies, as instance). Then, there is no information about the context where the reconfiguration operations are called. The AtelierB cannot be used to verify the instantiated model because its provers have no hypothesis to help them to prove that the reconfigurations preserve the architectural invariant.

Nevertheless, in order to (partially) validate the instantiated B model, ProB can be used to check the consistency of samples of the instantiated B model: at each reconfiguration step, ProB checks the architectural constraints to find an example violating the invariant. It is therefore possible to produce traces containing sequences of running and reconfiguration operations.

For example, it is easy to reproduce this short scenario (the right-side figure depicts the corresponding trace generated by ProB):



(i) The initial configuration of the HTTP server is without the **CacheHandler** and **FileServer2** components;

(ii) Next configuration is obtained by running the architecture and changing the load and deviation values;

(iii) At this stage, we add **CacheHandler** to the global architecture, following the AddCacheHandler reconfiguration operation;

(iv) After that the **FileServer2** component is added to the architecture, through the AddFileServer reconfiguration;

(v) The architecture is running;

(vi) By applying the RemoveCacheHandler reconfiguration, the component **CacheHandler** is deleted from the global architecture.

Automatic random explorations allow us to check the instantiated B model in a more general manner. As example, ProB model-checks 1000 nodes into 5301 milliseconds: neither invariant violation nor deadlock was found, and all the operations were covered.

6 Checking Temporal Formulae over Reconfigurations

In this section, we exploit the linear temporal logic for dynamic reconfigurations introduced in [12] and called FTPL. It allows us to characterise the correct behaviour of reconfiguration-based systems by using architectural invariants and linear temporal logic patterns. FTPL has been inspired by proposals in [13], and their temporal extensions for JML [21,8,14]. In this work, we propose to translate the FTPL patterns into LTL formulae in order to check FTPL properties with the ProB model-checker.

6.1 FTPL Syntax and Semantics

Let us consider the subset of FTPL in the figure below. It is based on trace properties, each of them being a temporal constraint on (a part of) the execution of the dynamic reconfiguration model. The configuration properties, called $conf$, are first order logic formulae over sets and relational operations on the primitive sets and over relations defined in Sect. 4.1. Further, for a reconfiguration operation ope, its ending is considered as an event.

event ::=	*ope* **terminates**
trace ::=	**always** *conf*
|	**eventually** *conf*
|	$trace_1 \wedge trace_2$
|	$trace_1 \vee trace_2$
temp ::=	**after** *event temp*
|	**before** *event trace*
|	*trace* **until** *event*
|	**between** $event_1$ $event_2$ *trace*

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

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

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

Example 1 *The following property expresses an architectural constraint saying that always there is at least one file server. In other words, always there is at least one provided interface connected to the required interface* getServer *of* **RequestDispatcher***:*

$$\textbf{always}\ \exists\ \mathsf{iprov} \in \mathsf{ProvidedInterfaces}.\ \mathsf{Binding(iprov)} = \mathsf{getServer}$$

Example 2 *The reconfiguration* AddCacheHandler *(resp.* RemoveCacheHandler*) adds (resp. removes)* **CacheHandler** *when the* deviation *value is greater (resp. less) than 50. The following property specifies that the deviation value eventually becomes less than 50 between the considered reconfigurations:*

$$\textbf{between}\ \mathsf{AddCacheHandler}\ \textbf{terminates}$$
$$\mathsf{RemoveCacheHandler}\ \textbf{terminates eventually}\ \mathsf{deviation} < 50$$

These examples show that FTPL is more expressive than the proposals in [11], which only handle architectural invariants. Indeed, FTPL allows expressing event properties and temporal properties involving different kinds of temporal patterns which have been shown useful for practical applications [13].

6.2 From FTPL to LTL

We adapt the results in [21] and [8] and propose a translation of FTPL patterns into the LTL dialect considered by ProB, called $\mathrm{LTL}^{[e]}$ [19]. The translation procedure, denoted LTL(x), is inductively defined on the structure of the FTPL formula. Let *conf* be a configuration property, B(*conf*) a rewriting procedure giving the B predicate corresponding to *conf*, and *ope* a reconfiguration.

LTL(*conf*)	{B(*conf*)}
LTL(*ope* **terminates**)	[*ope*]
LTL(**always** *conf*)	G(LTL(*conf*))
LTL(**eventually** *conf*)	F(LTL(*conf*))
LTL($trace_1 \wedge trace_2$)	LTL($trace_1$) $\wedge$ LTL($trace_2$)
LTL($trace_1 \vee trace_2$)	LTL($trace_1$) $\vee$ LTL($trace_2$)

Let *trace*, $trace_1$ and $trace_2$ be trace properties, *event* an event property, and *temp* a temporal property. 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 an auxiliary functions LTL_B.

LTL(**after** *event temp*)	G(LTL(*event*) ⇒ LTL(*temp*))
LTL(**after** *event trace*)	G(LTL(*event*) ⇒ LTL(*trace*))
LTL(**before** *event trace*)	F(LTL(*event*)) ⇒ LTL_B(*event*, *trace*)
LTL(*trace* **until** *event*)	F(LTL(*event*)) ∧ LTL_B(*event*, *trace*)
LTL_B(*event*, **always** *conf*)	LTL(*conf*) U LTL(*event*)
LTL_B(*event*, **eventually** *conf*)	¬(¬(LTL(*conf*)) U LTL(*event*))
LTL(**between** $event_1$ $event_2$ *trace*)	LTL(**after** $event_1$ (*trace* **until** $event_2$))

The FTPL property presented in Example 1 has been translated into the LTL formula below. This formula has been partially checked with ProB in 126 milliseconds. The model checker generates 2002 atoms and 16064 transitions when the maximum number of new states is 1000.

```
G( { ∃(iprov) . (iprov ∈ ProvidedInterfaces ∧ Binding(iprov) = getServer) } )
```

Applying the above translation to the property in Example 2 results in the LTL property below, checked in 1802 milliseconds. The model checker generates 16048 atoms and 129704 transitions when the maximum number of new states is 1000.

```
G( [AddCacheHandler] ⇒
    F([RemoveCacheHandler]) ∧ ¬(¬({Value(deviation) < 50}) U [RemoveCacheHandler]) )
```

More sophisticated temporal properties involving architectural constraints can be written thanks to FTPL [12]. Then, thanks to our translation procedure, their verification can be investigated with ProB. Note that this verification is size-bounded and partial because of ProB features.

7 Conclusion

The different proposals presented in this paper concern the verification of dynamic reconfigurations of concurrent component-based systems. Dynamic architectural constraints could be expressed with a linear time temporal logic over (re)configuration sequences, as FTPL [12]. As architectural constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we have proposed an approach combining proof and model-checking to support the modelling of such systems and the validation of their dynamic reconfigurations, as depicted Fig. 1. We first have proposed a generic B model for component architectures and we have proved the consistency of the model architectural constraints. Then, we have instantiated the general model to address a particular architecture validation: its consistency and some temporal properties over (re-)configuration sequences—expressed in FTPL and translated into LTL—have been model-checked.

Our contributions for temporal properties specification and verification—including static and dynamic analysis—allow monitoring instrumentation and managing applications at runtime, thanks to available tools to animate specifications.

Related work. In the context of dynamic reconfigurations, ArchJava [4] gives means to reconfigure Java architectures, and the ArchJava language guarantees communication integrity at run-time. Barringer et al. give a temporal logic based framework to reason about the evolution of systems [6]. In [5], a temporal logic is proposed to specify and verify properties on graph transformation systems.

In the Fractal-based framework, the work in [17] has defined integrity constraints on a graph-based representation of Fractal, to specify the reliability of component-based systems. Unlike [17], our model lays down only general architectural constraints, thus providing an operational semantics to other component-based systems. On the integrity constraints side, the FTPL logic allows specifying architectural constraints more complex than architectural invariants in [11].

To enforce software robustness while adding adaptive behaviour, the work in [20] proposes a formal framework for the Fractal component model, named FracL. Like our B-based proposal, the FracL static approach allows verifying the consistency of the application architecture. However, our proposal allows checking the model consistency and monitoring temporal properties, both fully automatically.

Among other applications, our proposals aim at a monitoring of component-based systems. In [7], 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). Like the model in [7], our model is a first-order structure, but instead of considering a sequence of time stamps, we focus on reconfiguration operations. In [15], knowledge-based controllability is studied for constructing distributed controllers. The problem there is somewhat different than ours: the goal is to make the system behave exactly according to a given knowledge-based priority property, while here the reconfigurations must satisfy some given architectural and temporal constraints.

References

[1] Abrial, J.-R., "The B Book - Assigning Programs to Meanings," Cambridge University Press, 1996.

[2] Aguilar Cornejo, M., 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).

[3] Aguirre, N. and T. Maibaum, *A temporal logic approach to the specification of reconfigurable component-based systems*, Automated Software Engineering (2002).

[4] Aldric, J., *Using types to enforce architectural structure*, in: *In WICSA'08, February 2008*, 2008, pp. 23–34.

[5] Baldan, P., A. Corradini, B. König and A. L. Lafuente, *A temporal graph logic for verification of graph transformation systems*, in: *WADT'06: Proceedings of the 18th Int. Conf. on Recent trends in algebraic development techniques* (2007), pp. 1–20.

[6] Barringer, H., D. M. Gabbay and D. E. Rydeheard, *From runtime verification to evolvable systems*, in: *RV*, LNCS **4839** (2007), pp. 97–110.

[7] Basin, D. A., F. Klaedtke, S. Müller and B. Pfitzmann, *Runtime monitoring of metric first-order temporal properties*, in: *IARCS, FSTTCS 2008, India*, LIPIcs **2** (2008), pp. 49–60.

[8] Bellegarde, F., J. Groslambert, M. Huisman, J. Julliand and O. Kouchnarenko, *Verification of liveness properties with JML*, Technical report RR-5331, INRIA (2004).

[9] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *The fractal component model and its support in java*, Softw., Pract. Exper. **36** (2006), pp. 1257–1284.

[10] Chauvel, F., 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, 2009.

[11] David, P.-C., T. Ledoux, M. Léger and T. Coupaye, *FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures*, Annales des Télécommunications **64** (2009), pp. 45–63.

[12] Dormoy, J., O. Kouchnarenko and A. Lanoix, *Using temporal logic for dynamic reconfigurations of components*, in: *7th Int. Ws. on Formal Aspects of Component Software (FACS 2010)*, LNCS (2010), to appear.

[13] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, in: *ICSE*, 1999, pp. 411–420.

[14] Giorgetti, A., J. Groslambert, J. Julliand and O. Kouchnarenko, *Verification of class liveness properties with Java modelling language*, IET Software (2008).

[15] Graf, S., D. Peled and S. Quinton, *Achieving distributed control through model checking*, in: *Proc. of the 22nd Int. Conf. CAV 2010*, LNCS **6174** (2010), pp. 396–409.

[16] Léger, M., "Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composant," Ph.D. thesis, Ecole Nationale Supérieure des Mines de Paris (2009).

[17] Léger, M., T. Ledoux and T. Coupaye, *Reliable dynamic reconfigurations in a reflective component model*, in: *CBSE 2010*, LNCS **6092** (2010), pp. 74–92.

[18] Leuschel, M. and M. J. Butler, *ProB: A model checker for B*, in: *Int. Symp. of Formal Methods Europe FME'03*, LNCS **2805** (2003), pp. 855–874.

[19] Leuschel, M. and D. Plagge, *Seven at one stroke: Ltl model checking for high-level specifications in b, z, csp, and more*, in: *ISoLA'07*, Revue des Nouvelles Technologies de l'Information **RNTI-SM-1**, 2007, pp. 73–84.

[20] Simonot, M. and M. Aponte, *Une approche formelle de la reconfiguration dynamique.*, L'OBJET **14** (2008), pp. 73–102.

[21] Trentelman, K. and M. Huisman, *Extending jml specifications with temporal logic*, in: *AMAST 2002*, LNCS **2422**, 2002, pp. 334–348.