Leveraging Models From Design-time to Runtime. A Live Demo

Brice Morin, Grégory Nain, Olivier Barais, Jean-Marc Jézéquel

Leveraging Models From Design-time to Runtime. A Live Demo.*

Brice Morin[1], Grégory Nain[1], Olivier Barais[1,2], and Jean-Marc Jézéquel[1,2]

[1] INRIA, Centre Rennes - Bretagne Atlantique
Brice.Morin@inria.fr | Gregory.Nain@inria.fr
[2] IRISA, Université Rennes1
barais@irisa.fr | jezequel@irisa.fr

Abstract. This paper describes a demo which leverages models at design-time and runtime in the context of adaptive system, some details about the underlying approach as well as some implementation details. Our tool allows deploying and dynamically reconfiguring component-based applications, in a guided and safe way, based on the OSGi platform. It combines reflexive and generative programming techniques, based on models, to achieve this goal.

1 An Overview of the Demo

The demonstration illustrates our tool-chain on a simple HelloWorld application. In this section, we briefly present the different tools of the chain, as well as the scenario to be demonstrated during the workshop. More details about the tools are given in Sections 3 and 4 and in previous publications [1,2,3].

1.1 Quick Overview of the Tool-chain

Our tool leverages models at design-time but also at runtime in order to deploy and dynamically reconfigure component-based applications. We rely on the SCA (Service Component Architecture, see http://www.eclipse.org/stp/sca/) editor to graphically design architectural models in the Eclipse IDE. We have extended this editor with code generation capabilities. This way, we can generate at design-time all the platform-specific infrastructure we need to be able to leverage models at runtime. We currently target OSGi [4] and Fractal/FraSCAti [5] as runtime platforms.

In the remainder, we will particularly focus on the OSGi version of the tool. Once all the infrastructure has been generated, it is possible to start the tool responsible for managing adaptive systems. This tool is based on [1,2] and is an extension of the early prototype [3], demonstrated last year at the workshop [6],

* The research leading to these results has received funding from the European Community's Seventh Framework Program FP7 under grant agreements 215412 (DiVA, http://www.ict-diva.eu/) and 215483 (S-Cube, http://www.s-cube-network.eu/).

for managing Fractal-based systems. Unlike pure OSGi or Spring DM (see Section 2), this tool leverages models both for the initial deployment and for subsequent dynamic reconfigurations, preventing programmers from writing low-level reconfiguration scripts.

1.2 Components to be Manipulated during the demo

The HelloWorld application manipulates three kinds of component types: Client, Server and VoiceServer. It can be composed of any number of Client component instances and any number of (Voice)Server component instances, collaborating according to different schemes. The demo will manipulate up to 34 component instances and 33 bindings (connections) among these components.

Server components provide the IHelloWorld interface and optionally require [0..1] a VoiceServer. Depending on their real implementation, the Server components may "say hello" in different languages: English, Spanish, etc, by returning the corresponding String and optionally ask a voice server to actually say "hello", "hola", etc.

Client components have two required ports, both typed with the IHelloWorld interface:

- default: a single mandatory required port [1..1]
- others: a multiple optional required port [0..*]

Each client component is associated with a GUI: a window composed of a button and a text area. By clicking on the button, the client component simply asks the default server to say hello and, if any, all the other servers to also say hello. The results are printed in the text area, as illustrated in Figure 1.

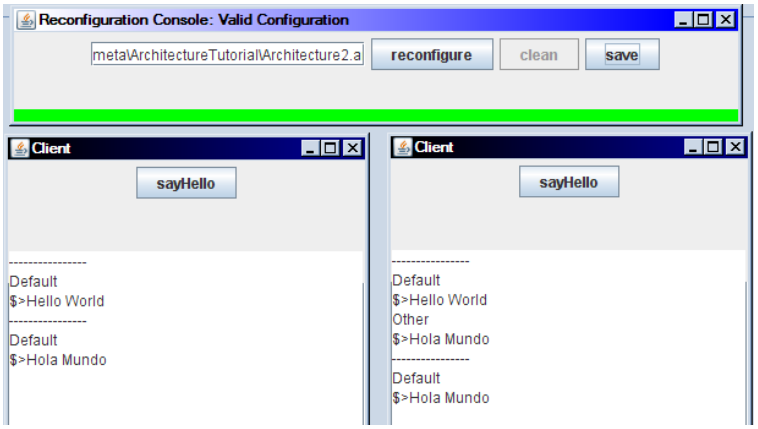

Fig. 1. Two clients after the initial deployment and a reconfiguration

Using these two types of components, it is possible to define an arbitrary number of valid and invalid configurations. For example, a configuration where a client component is not connected to a default server is invalid, since the default port is mandatory.

1.3 Scenario

We now outline the scenario to be demonstrated at the workshop.

Models At design-time At design-time, we will briefly introduce the SCA graphical editor and show how configurations are designed. Then, we will automatically generate all the code we need for the remainder of the demo.

Models At Runtime At runtime, we will demonstrate how the initial configuration is deployed and perform several attempts of model-driven reconfigurations, with valid and invalid configurations. Moreover, we will show how our tool manages unpredicted changes at runtime *e.g.*, when a component is directly uninstalled at the platform level.

1. **Start**: It displays a simple GUI to be able to load configurations
2. **Load the initial configuration**: Using the GUI, load the initial configuration (architectural model composed of two clients and two servers). It displays the GUI associated with the two clients.
3. **Say Hello**: Click on the "sayHello" button of each client. Depending on the way clients are connected with servers, this will produce a different result.
4. **Load another configuration and Say Hello**: This configuration is valid and actually produces a reconfiguration of the system (*e.g.*, one client disappear, the other uses the servers in a different way).
5. **Load another configuration**: This configuration is not valid (default port not connected). This does not produce a reconfiguration of the system.
6. **Servers crash!** The server components are manually (via the OSGi textual console) removed from the system. The tool detects this change and notifies the user that the current configuration is no more consistent.
7. **Load another configuration**: If the configuration is valid, the system will be reconfigured into a consistent configuration.

2 Background

This section briefly introduces the different technologies related to our tool and highlights their limitations.

2.1 OSGi

The OSGi (Open Services Gateway initiative) consortium is composed of famous companies from different domains: automotive industry (BMW, Volvo), mobile industry (Nokia), e-Health (Siemens), SmartHome (Philips), etc. It provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. See http://www.osgi.org for more details about OSGi.

Typically, an OSGi bundle (component) is responsible for managing its dependencies by itself. It can try to set its dependencies when it is starting, by searching required services from the OSGi service registry, or by registering to the OSGi event mechanism to be notified when required services appear or disappear. For example, the following code fragment sets the *helloworld* reference when the client component starts.

```
/*
 * Client Component, in OSGi
 */
public class Client implements BundleActivator, IClient {

  private BundleContext context;
  private IHelloWorld helloworld;

  public void start(BundleContext context) {
    this.context = context;

    //registering the component as IClient
    context.registerService(IClient.class.getName(), this, null);

    //setting the helloworld reference
    ServiceReference[] refs = context.getAllServiceReferences(null,
        "(objectClass="+IHelloWorld.class.getName()+")");
    helloworld = (IHelloWorld)context.getService(refs[0]);
  }
}
```

OSGi provides very flexible and powerful mechanisms for managing the life-cycle of components. However, since the dependencies should be handled inside the components themselves, it is very difficult to separate the business logic from the adaptive logic and implement complex adaptation policies involving several collaborating components.

2.2 Spring DM

Spring DM (Dynamic Modules) allows managing configurations of OSGi-based applications by deploying the initial configuration from a XML file, dynamically adding, removing, and updating modules in a running system. Moreover, it has the ability to deploy multiple versions of a module simultaneously. See http://www.springsource.org/osgi for more information on Spring DM.

Typically, a Spring bean (component) is a POJO with getters and setters for the reference that can be accessed and setted. Unlike OSGi, components are not responsible for setting their dependencies by themselves. On the contrary, these references are set from the outside, by calling the appropriate getters/setters. For example, the following code fragment illustrates the Client component.

```
/*
 * Client Component, in Spring
 */
public class Client implements IClient {

  private IHelloWorld helloworld;

  public IHelloWorld getHelloworld(){
    return helloworld;
  }

  public void setHelloworld(IHelloWorld helloworld){
    this.helloworld = helloworld;
  }
}
```

The initial runtime configuration is instantiated, deployed and started by loading the XML file describing this initial configuration, similarly to an ADL (Architecture Description Language). This file mainly describes the component types

(factories) needed to instantiate component instances (beans in the Spring terminology), and how beans are wired together.

Spring DM does not allow the declarative updating of this XML file in order to dynamically reconfigure the system. On the contrary this should be done programmatically by modifying the properties of the components and calling the `modifyConfiguration(ServiceReference ref, Dictionary props)` method provided by the Configuration plugin, or directly calling the getters/setters provided by the components.

This is a major drawback since the initial configuration and the subsequent reconfigurations are not handled in a consistent way. On the one hand, the initial configuration is instantiated from a declarative specification describing the overall configuration of the system. On the other hand, the dynamic reconfigurations are realized in an imperative style, which requires the developer to set the dependencies in a programmatic way. In the case of a large reconfiguration, this requires to implement a long script describing how all the properties are updated. Moreover, it is very difficult to understand or validate the new configuration *a priori*, since no explicit representation (model or XML file) of this new configuration exists.

3 An Overview of SC@rt

This section describes SC@rt (SCA at runtime), to be demonstrated (live demo) during the workshop. The main objective of our approach is to leverage models at design-time in order to support the initial deployment of the system, but also at runtime, to provide a high-level support for dynamic reconfigurations.

3.1 Designing Architecture with the SCA Editor

The first step consists in designing the component types to be manipulated by the application, in the SCA editor. Here, we only manipulate two types: client and server. This diagram simply consists of non-connected components describing their provided and required ports.

Then, it is possible to define different configurations of the hello world application. In practice, this consists in creating a new diagram, and copy/pasting component types from the type diagram in order to instantiate component instances. These instances can then be connected together on compatible ports. Figure 2 illustrates one possible configuration.

3.2 Generating the Code of Component Types

After the type diagram has been designed, we propose to generate the code of the type components. These type components are factories responsible for instantiating component instances. We have extended the SCA editor so that the code generation is simply invoked by a right click on the diagram. The code of the factory is rather systematic and only varies on some well identified points. We use String Template (see http://www.stringtemplate.org/) to generate this code.

3.3 Leveraging Architecture Models to Deploy and Reconfigure the System

Figure 3 illustrates our causal connection between an architectural model and a running system, based on our previous works [1,2].

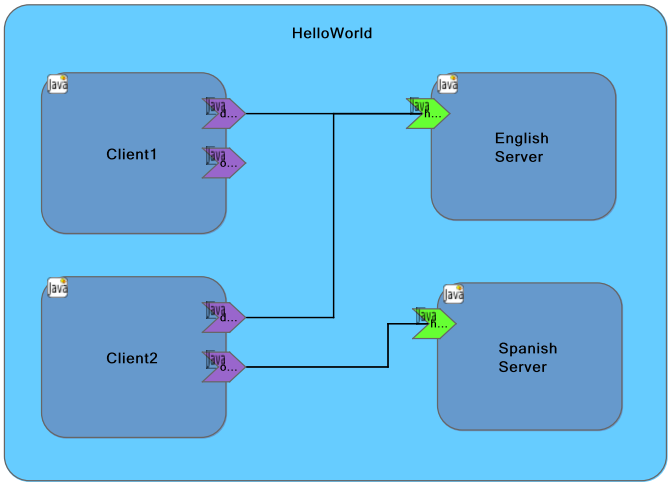

Fig. 2. One possible configuration of the Hello World application

The key idea is to keep an architectural model synchronized with the running system [6]. This reflection model, is updated by observers integrated in the execution platform (Figure 3, 1) when significant changes appears in the running system (addition/removal of components/bindings).

When a target architectural model is defined (*e.g.*, by modifying a copy of the reflection model), it is first validated using classic design-time validation techniques, such as invariant checking or simulation. This new model, if valid, represents the target configuration the running system should reach.

Then, we generate a reconfiguration script by first comparing the source configuration (the reflection model) with the target configuration (Figure 3, 3) and generating an ordered set of reconfiguration commands. This set of commands describes a safe reconfiguration script (no life-cycle errors or dangling bindings) which is submitted (Figure 3, 4) to the running system in order to actually reconfigure it. Note that when a new component is added into the model, we generate, compile and package some parts of its code at runtime (the Activator and the MANIFEST.MF)[3]. This way, component instances are handled as OSGi bundle, making it possible to properly install, uninstall, start and stop them.

Finally, the reflection model is automatically updated when the reconfiguration commands are correctly executed. Commands not correctly executed do not update the reflection model and are logged so that they can be post-processed *e.g.*, to implement a roll-back. If all the commands are correctly executed, the updated reflection model becomes equivalent to the target model (Figure 3, 1).

[3] the business logic of the component is already present and compiled in the factory components

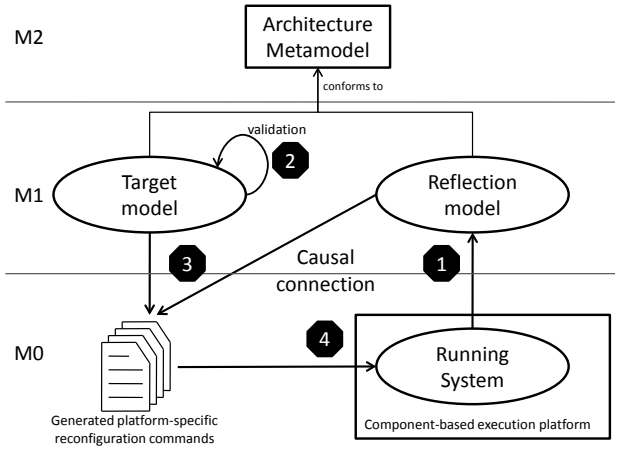

Fig. 3. Strong synchronization from runtime to model. Delayed synchronization (after validation) from model to runtime

4 Implementation Details

This section gives some implementation details of the prototype we have described.

4.1 Reconfiguration Commands

The dynamic reconfiguration process is based on the Command design pattern. Basically, we represent a reconfiguration script as a list of commands, as explained in [1,2]. These commands are ordered according to their priority: *i*) remove binding, *ii*) remove component, *iii*) add component and *iv*) add binding.

We define the parameter of each command as public attributes. The `check` method is responsible for verifying that the parameters of the command have been well initialized. The `execute` method actually performs an atomic reconfiguration on the running system (*e.g.* add a component). Finally, the `doAck` method is called when the command has successfully been executed.

```
1  public abstract class PlatformCommand {
2
3    private boolean ack = false;
4
5    abstract public int getPriority();
6
7    //Checks the consistency of the command
8    abstract public boolean check();
9
10   //Executes the command i.e., a performs a runtime adaptation
11   abstract public void execute();
12
13   public boolean ack(){
14     return ack;
15   }
16
17   /* Acknowledge the command i.e., update the reflection model
18      Should be overriden by concrete commands. */
19   public void doAck(){
```

```
20      ack = true;
21    }
22  }
```

4.2 OSC@rt: Models@Runtime over OSGi

The current prototype does not directly use SCA models at runtime. Instead, it uses architectural models conforming to a core architecture metamodel described in [3], to reduce the memory overhead at runtime implied by the causally connected model. However, we have defined a bi-directional model transformation between SCA and our metamodel, in Kermeta [7].

The main class of this prototype is the `OSGiCausalLink` class illustrated (fragment) in the following scripts.

```
public class OSGiCausalLink implements BundleActivator, CausalLink,
    EventHandler, BundleListener {
  private art.System system; //the reflection model
  private art.System updateModel; //a new model to switch to

  private Checker checker;

  public void reconfigure() {
    if (checker.check(updateModel)){
      computeMatch(system, updateModel);
      getCommands();
      for(PlatformCommand cmd : commands){
        cmd.execute();
      }

      Timer timer = new Timer();
      AckTimerTask att = new AckTimerTask();
      att.commands = commands;
      timer.schedule(att, ackPeriod);
    }
  }
```

The main method of the `OSGiCausalLink` class is `reconfigure`. This method loads a new architectural model and leverages this model to automatically adapt the running system. This prevents the programmer from writing long imperative reconfiguration scripts. This method first performs a model comparison between the reflection model and the new model to determine the commonalities (what has not changed) and the differences between these two models (what has been added or removed). Then, by analyzing the result of the model comparison, a sequence of reconfiguration commands is instantiated and finally executed. A separate thread (`AckTimerTask`) checks whether all the commands have been acknowledged within a given period, or not.

```
private void addBinding(TransmissionBinding b, ComponentInstance client) {
  AddBindingOSGi cmd = new AddBindingOSGi();
  cmd.b = b;
  cmd.client = client;
  commands.add(cmd);
}

private void doAddBinding(TransmissionBinding b, ComponentInstance client) {
  client.getBinding().add((TransmissionBinding)b);
}
```

The `addBinding` method shows how a command is instantiated. When a new binding is detected in the model, we simply instantiate and initialize the right command. Note that, at this point, the reflection model has not been updated. In other words, the new binding has not been actually added to the reflection model. The binding will be added to the reflection model, in the `doAddBinding` method, only when the command will be properly acknowledged.

```
public void handleEvent(Event event) {
  PlatformCommand cmd = (PlatformCommand)event.getProperty("command");

  if (event.getTopic().startsWith("binding/ok/")){
    TransmissionBinding b = (TransmissionBinding)event.getProperty("binding");
    doAddBinding(b, ((AddBindingOSGi)cmd).client);
    cmd.doAck();
  }
  else if (event.getTopic().startsWith("binding/nok/")){
    TransmissionBinding b = (TransmissionBinding)event.getProperty("binding");
    System.err.println("Problem when binding "+b);
  }
  ...
}
```

We rely on the standard `EventAdmin` service provided by OSGi to acknowledge commands. When a command properly execute, an event containing the command is posted on the appropriate topic *e.g.*, "binding/ok". When receiving this event (`handleEvent(Event event)`), we simply call the `doAck` method of the command and update the reflection model (*e.g.*, by calling the `doAddBinding` method). When a command does not execute properly, and event is posted on another topic (*e.g.*, on "binding/nok"). In this case, the command is not acknowledged and the reflection model is not updated. In this case, the `AckTimerTask` generates a report specifying which commands have been acknowledged and which commands have not.

```
public void bundleChanged(BundleEvent event) {
  Bundle b = event.getBundle();

  if (event.getType()==BundleEvent.UNINSTALLED) {
    ComponentInstance cpt = runtime2model.get(b);
    system.getRoot().getSubComponent().remove(cpt);
    runtime2model.remove(b);
  }
  else if (event.getType()==BundleEvent.STOPPED){
    ComponentInstance cpt = runtime2model.get(b);
    cpt.getBinding().clear();
    removeAllDanglingBindings(cpt);
  }
}
```

Our causal link implements the standard `BundleListener` interface specified in the OSGi standard. This way, we are automatically notified when components are stopped or uninstalled. When a component is stopped, we clear all its dependencies and clear all the dependencies of all the components using this stopped component, by calling the `removeAllDanglingBindings`, and remove all the associated bindings from the reflection model. Then, if the component is uninstalled, we simply remove it from the reflection model.

5 Conclusion

This paper has presented our tool-chain, to be demonstrated at the workshop. This tool chain leverages models at design-time but also at runtime to deploy and dynamically reconfigure component-based applications, in a guided and safe way. At design-time, we use the SCA editor to design the architecture, and code generation techniques to produce all the infrastructure we need at runtime. At runtime, the reconfiguration process is totally driven by the design models. Before adaptation, we check that the target configuration is valid. In this case, we generate the corresponding reconfiguration script, based on the command pattern. Our tool also handles (in a limited way) unpredicted events, and is able to update the reflection model accordingly and notifies the user when this model becomes invalid.

In future work, we plan to improve the adaptation process by considering more constraints during the generation of commands. Currently, commands are ordered in a very simple way in order to avoid common errors like dangling bindings. We would like to consider other constraints, specified by the architect, such as *Component A* should be stopped *before Component B*, etc.

References

1. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming Dynamically Adaptive Systems with Models and Aspects. In: ICSE'09: 31st International Conference on Software Engineering, Vancouver, Canada (May 2009)
2. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: An aspect-oriented and model-driven approach for managing dynamic variability. In: MODELS'08: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, Toulouse, France (October 2008)
3. Morin, B., Barais, O., Jézéquel, J.M.: K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In: 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)
4. The OSGi Alliance: OSGi Service Platform Core Specification, Release 4.1 (May 2007) http://www.osgi.org/Specifications/.
5. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL Component Model and its Support in Java. Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11-12) (2006) 1257–1284
6. N. Bencomo, G. Blair, R.F.: Proceedings of the international workshops on models@run.time (2006-2008)
7. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (October 2005) 264–278