K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines

Brice Morin, Olivier Barais, Jean-Marc Jézéquel

K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines*

Brice Morin, Olivier Barais, and Jean-Marc Jézéquel

IRISA / INRIA Rennes / Université Rennes 1
EPI Triskell, Campus de Beaulieu
35042 Rennes Cedex, France

Abstract. Software systems should often propose continuous services and cannot easily be stopped. However, in order to meet new requirements from the user or the marketing, systems should be able to evolve in order to propose new services or modify existing ones. Adapting software systems at runtime is not an easy task and should be realize with attention. In this paper, we present K@RT, our generic and extensible framework for managing dynamic software product lines. K@RT is composed of three parts: *i)* a generic and extensible metamodel for describing running systems at a high-level of abstraction, *ii)* a set of meta-aspects that extends the generic metamodel with constraint checking, supervising and connections with execution platforms *iii)* some platform-specific causal connections that allow us to supervise systems running on different execution platforms.

1 Introduction

Developing, testing and validating adaptive systems is a daunting task. Indeed, such systems can propose a wide range of possible configurations at runtime [15, 19]. These systems can be seen as Dynamic Software Product Lines (DSPL) that can reconfigure themselves at runtime.

In order to facilitate the development, test and validation of DSPLs, we propose K@RT, our aspect-oriented and model-oriented framework for supervising component-based systems. This generic framework is independent from any underlying execution platform and proposes to maintain a reference model at runtime [8]. Using this high-level view of the running system, we can navigate the runtime architecture using model-oriented languages [21] and invoke services that are delegated to the running system. K@RT also allows to adapt the running system by modifying its runtime model, checking constraints on the modified model and comparing the actual reference model to the modified model. This process produces a safe reconfiguration script that is executed on the running system. The modified model may be obtained with high-level model-transformation languages [21] or Aspect-Oriented Modeling (AOM) approaches [13, 17, 20], avoiding users to write low-level platform-specific reconfiguration scripts.

The remainder of this paper is organized as follows. Section 2 introduces our generic and extensible metamodel for representing models at runtime. Section 3 briefly presents

* This work was funded by the DiVA project (EU FP7 STREP, Theme 1.2: Service and Software Architectures, Infrastructures and engineering, Contract 215412)

our causal link between a running system and a runtime model. Section 4 details the aspect-oriented architecture of K@RT. Section 5 evaluates our framework. Finally, Section 6 concludes and outlines future works.

2 A Generic and Extensible Metamodel for Runtime Models

In this section, we present our generic metamodel[1] for representing component-based systems at runtime. This metamodel does not aim at representing high-level architectures but focuses on abstracting a running system. This metamodel is independent from any execution platform and can easily be mapped on Fractal [9, 10], OpenCOM [11], PEtALS ESB [1] or SCA [2].

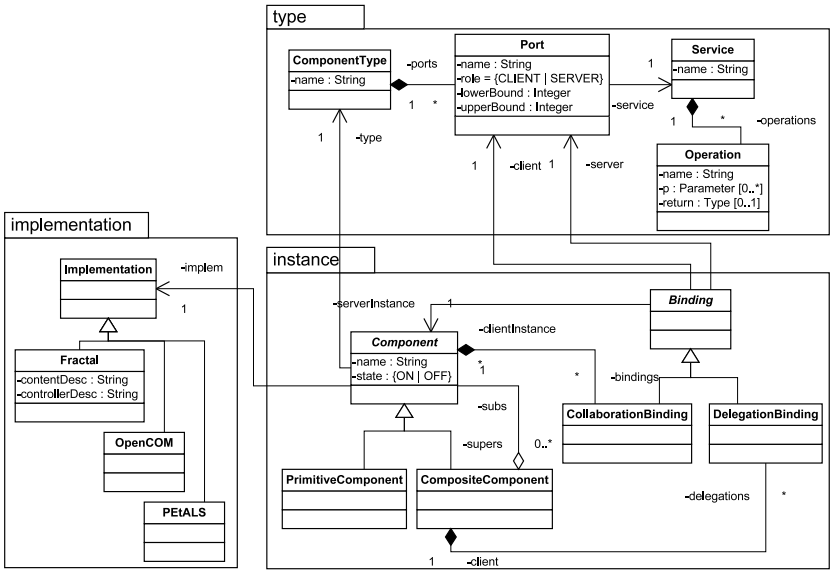

Fig. 1. A Generic and Extensible Metamodel

Our generic metamodel is separated in three packages, as illustrated in Figure 1. The **type** package defines the notion of component type. A component type contains some ports. Each port has a UML-like cardinality (upper and lower bounds) indicating if the port is optional (*lowerBound* == 0) or mandatory (*lowerBound* $>$ 0). It also indicate if the port only allows single bindings (*upperBound* == 1) or multiple bindings (*upperBound* $>$ 1). A port also declares a role (client or server) and is associated to a service. A service encapsulates some operations, defined by a name, a return type and some parameters. Basically, a service has a similar structure than a Java interface.

[1] In this paper, "metamodel" refers to the MOF terminology, not the middleware terminology.

The **instance** package defines the actual topology of a running system. A component has a type and a state (ON/OFF), specifying whether the component is started or stopped. It can be bound to other instances by a collaboration binding, linking a provided service (server port) to a required service (client port). A composite instance can additionally declare sub-instances and delegation bindings. Note that our metamodel allows shared components as a component may have several super components. A delegation binding specifies that a service from a sub-component is exported by the composite instance.

The **implementation** package contains metaclasses responsible for encapsulating the platform-specific attributes needed to implement components for a given platform. For example in Fractal, we should specify the implementation class (*contentDesc*) and a controller (*controllerDesc*) in order to be able to create a component.

We prefered to define a domain-specific metamodel (DSM) rather than reusing for example the UML 2.0 metamodel. Indeed, a reference model conforming to this metamodel is causally connected to the running system. Using a DSM allows us to reduce the number of entities that have to be maintained at runtime and consequently limit the memory overhead. This metamodel is strongly-typed and allows us to define algorithms with few casts whereas it is often necessary to perform casts when working at the platform level as they often deal with losely-typed objects. Moreover, this metamodel is aligned on the Service Component Architecture (SCA) [2] metamodel proposed by the Open Service Oriented Architecture collaboration (OSOA) [3] that brings together industrial partners like IBM, Sun, Oracle, SAP or Siemens. Our metamodel can be seen as a ligthweight version of SCA. This allows us to easily map our metamodel to SCA [2] and reuse the tools provided by SCA, such as a graphical editor to visualize the runtime architecture.

3 A Model-Driven Causal Connection

This section briefly presents our model-driven causal connection between a reference model, conforming to the metamodel we have presented in Section 2, and an execution platform. Currently, we have implemented such a causal connection for the Fractal [9, 10] platform but it can also be implemented for other component-based execution platforms like OpenCOM [11] or PEtALS [1], if they provide reflection and dynamic reconfiguration mechanisms. The architecture of this causal connection is illustrated in Figure 2 and is detailed in the next two subsections.

The *Model2Platform* component is in charge of reflecting the changes of the model to the platform. This components will be detailed in this section. Identically, the *Platform2Model* component reflects the changes of the running system to the model. These two components use the *Factory* component in order to instantiate model elements from runtime entities, and vice-versa. The *Root* component is a composite component that contains the system designed by the user. This component is not really part of the causal link and may be deployed on a different site than the other components implementing the causal connection.

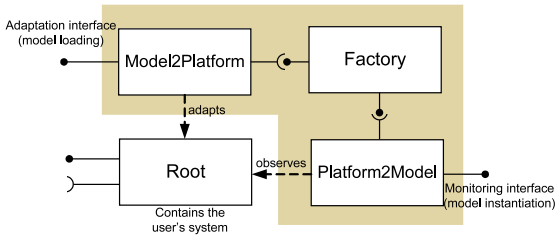

Fig. 2. Architecture of our Causal Connection

3.1 From Platform to Model

This subsection describes how we generate and update a reference model that represents, at a higher level of abstraction, the running system.

Fractal [9, 10] and all the reflective component-based execution platforms propose mechanisms for introspecting a running system. These mechanisms allows to discover which components actually compose the system, how they are bound to each others, etc. We extend the introspection operations provided by middleware approaches in order to discover the operations and their parameters that are provided/required by ports. In the Java-based distribution of Fractal or OpenCOM, each port (provided or required interface) is associated to a Java interface. We use the `java.lang.reflect` API to discover these operations and give a more precise view of the system.

Using reflection is very useful to instantiate a model from scratch. But, if we want to keep the model up-to-date, instantiating a complete model periodically may be time and resource consuming if only minor changes occurs. We have instrumented the Fractal platform to observe and notify all the architectural reconfigurations that appear in the running system. This allows us to update the reference model.

Finally, it is possible to visualize the runtime architecture in the graphical editor provided by SCA. Indeed, we have defined a model transformation in Kermeta [21], that maps the concepts of our metamodel to the concepts of SCA.

3.2 From Model to Platform

This subsection describes the other part of our causal connection. In K@RT, the only way to adapt a running system is to submit a new model to the causal link (see Section 4). When a new model is submitted to the causal link, we perform a difference analysis between the modified model and the actual reference model. In the current implementation of K@RT, we use EMF Compare [4] in order to realize this analysis. EMFCompare provides a generic comparison engine that can be customized for any domain-specific metamodel.

After analyzing the output provided by the comparison engine, we can determine what has been removed from the model, added into the model or updated. However, we cannot directly adapt the running system using these elements. Indeed, we cannot ensure that the order we discover the modifications during the analysis will result in

a consistent adaptation of the running system. For example, if we discover that some bindings and some components have been removed, it would probably lead to dangling bindings in the running system if we directly adapt the system. In order to adapt the running system in a consistent way, we reify every significant modification as a command. Each command declares a priority (*e.g.*, a command that removes a binding has a higher priority than a command that removes a component). These commands are automatically ordered with a Comparator. Once all the commands are instantiated, they are executed in the right order in order to actually adapt the running system. We first stop the components that needs to be stopped, we remove all the bindings and the components, add the new components and the bindings and finally restarts the components.

4 K@RT: Kermeta at RunTime

This section presents our aspect-oriented and model-oriented framework for supervising component-based systems at runtime. This framework is based on the generic and extensible metamodel presented in Section 2 (Figure 1) and is implemented in Kermeta [21]. Three Kermeta meta-aspects, **constraint checker**, **supervising** and **platform adapter** extends the generic metamodel, as illustrated in Figure 3. Kermeta meta-aspects allows us to statically introduce new features in existing model elements: adding classes in packages, adding super classes in the inheritance tree, adding and implementing new operations and adding contracts (invariants, pre/post conditions).

4.1 Constraint checker meta-aspect

This subsection details the constraint checker meta-aspect. This aspect weaves invariants into metaclasses. These invariants can be written in OCL [5] and translated into Kermeta thanks to the OCL Kermeta plugin, or directly written in Kermeta. We illustrate this aspect by detailing one of the invariants we have implemented.

The *completeCollaborationBindings* invariant illustrated in Figure 4 specifies that all the client (*PortRole.CLIENT*) and non optional ports defined in the type (*self.type*) of the component should be targeted (*b.client*) by the client reference of a binding owned by the component (*self.binding*).

This invariant uses the OCL-compliant operators provided by Kermeta (*e.g.* select, forAll, exists, etc), which significantly reduce the complexity of writing invariants. The same invariants implemented in Java/EMF needs 15 lines of code and would even be more complex if it was directly implemented using the platform API.

Specifying constraints on the metamodel allows us to check well-formedness rules that all the runtime models, and consequently all the running systems must respect. Using model-oriented constraint languages like OCL or Kermeta allows designers to rapidly implement such invariants as these languages propose high-level operators for manipulating models. Note that it is possible to aspectized this constraint checker aspect in order to implement additional constraint. For example, if the underlying execution platform do not support shared component, an invariant can check that components have no more than one super component. Currently, 6 invariants are implemented in the constraint checker aspect.

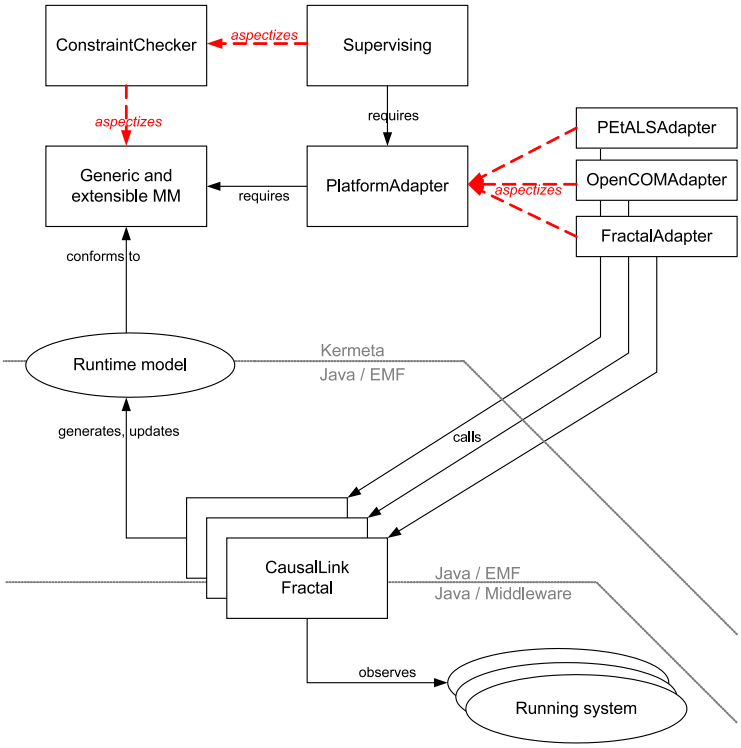

Fig. 3. K@RT overview

4.2 Supervising meta-aspect

The supervising aspect implements an administration console. It introduces two metaclasses: *DisplayContext* and *DisplayElement*. The *DisplayContext* metaclass is responsible for managing the history of the administration console and provides some useful method for displaying information. The *DisplayElement* simply defines an abstract operation *display(context : DisplayContext)*. In the aspect, this metaclass is introduced as a super class for all the elements that may be displayed: *Component*, *ComponentType*, *Binding*, etc. The *display* operation is implemented in each subclass. The *DisplayContext* and *DisplayElement* metaclasses can be seen as an interactive and history-aware visitor pattern allowing to display the elements chosen by the user and to go back to the previously visited elements.

4.3 Adapter meta-aspect

This aspect is responsible for connecting Kermeta to the execution platform. Kermeta proposes a seamless mechanisms for calling Java programs. Thus, it is possible to connect our K@RT framework with Java-based distribution of Fractal (Julia, AOKell),

```
1 aspect class Component {
2   inv completeCollaborationBindings is do
3     self.type.ports.select{p |
4       not p.isOptional and p.role == PortRole.CLIENT}
5         .forAll{p |self.binding.exists{b | b.client == p}}
6   end
7 }
```

Fig. 4. Component metaclass aspectized with an invariant

OpenCOM, PEtALS ESB, etc. Currently, the Fractal adapter is fully functional and other adapters are under development. The adapter aspect proposes operations for:

- Instantiating the reference model from scratch using the introspection API provided by the underlying middleware platform. In Fractal, we use the content, binding, name, lifecycle and attribute controllers.
- Getting the current reference model using the notification mechanisms provided by the underlying middleware platform. This allows updating the reference model instead of generating it from scratch. We have implemented a new controller for Fractal that notifies all the runtime architectural changes to registered observers.
- Loading a model. It loads the model, analyzes the diff and match models and computes a safe reconfiguration script, as described in Section 3.
- Invoking services. Fractal does not propose controllers for easily accessing and invoking methods in a reflective way. We tackle this issue by directly using the `java.lang.reflect` API in order to discover which operations can be called and actually call them from Kermeta. We plan to integrate this implementation in a Fractal controller.

4.4 Discussion

K@RT is implemented according to our generic metamodel, instead of directly refering to an underlying execution platform *e.g.* Fractal, OpenCOM, PEtALS, etc. It allows us to reuse it for different platforms provided that they could be mapped, in both directions, to the metamodel. However, if the execution platform cannot directly be mapped to the metamodel, it is possible to aspectize the metamodel and the related meta-aspects to extend them with new concepts.

Fractal Explorer [6] is a tool for managing Fractal-based applications via a graphical console. Currently, our console is textual but it would be possible to connect K@RT to a Java-based graphical console, as Kermeta programs can be connected to Java programs. The main differences between Fractal Explorer and K@RT are summarized below:

- K@RT is technology independent while Fractal Explorer is based on Fractal. Indeed, K@RT is based on our generic and extensible metamodel that allows us to connect it to different execution platforms

– K@RT offers a higher level of abstraction. Indeed, in Fractal Explorer all the details of the Fractal component model are displayed in the console: content-controller, binding-controller, lifecycle-controller, etc. In K@RT, a Fractal component that declares a binding-controller and a lifecycle-controller is simply represented by a component that contains some bindings and declares a state. The Fractal-specific notion of controller is abstracted.
– K@RT offers a higher level reconfiguration process. Indeed, K@RT proposes to adapt the running system by modifying the reference model. It is possible to use any transformation languages like Kermeta [21] or Aspect-Oriented Modeling tools [13, 17, 20] to modify the model. Then, this modified model is checked and automatically translated in a safe reconfiguration script. Fractal explorer is limited to fine grained reconfiguration.

5 Evaluation of K@RT

In order to evaluate K@RT, we have implemented a prototype in Fractal. This example is based on the service discovery system [14]. A service discovery system can either *advertise* or *request* services. It can also provide both functionalities. A service discovery system can communicate via several technologies (at least one). In this example, we propose WiFi and Bluetooth (BT). Figure 5 shows the complete architecture of the service discovery system, with both roles and both communication technologies. This model is automatically generated from the running system and mapped to SCA.

We define each functional role (Advertiser or Requester) as an aspect and each communication technology (WiFi or Bluetooth) as an aspect. We use an Aspect-Oriented Modeling tool (SmartAdapters [16–18]) to weave these aspects and produce all the possible configurations [17, 22]. There exists 9 possible configurations for the service discovery system: Advertiser role, Requester role or both and WiFi, Bluetooth or both. Consequently there is 9*(9-1) = 72 possible transitions from one configuration to another. Our causal link succeed to reconfigure the system at runtime for all these 72 transitions. The average time for reconfiguration was 200 ms.

Since all the aspects are independent from each others, it would be possible to handle the adaptive behavior of the service discovery systems with 8 scripts, for adding/removing each aspects. The identification of aspect dependencies and the generation of the minimal set of reconfiguration scripts will be subject to future work.

6 Conclusion and Future Works

In this paper, we have presented K@RT, our framework for developing, testing and validating Dynamic Software Product Lines (DSPL). This framework allows us to construct adaptive systems by defining model transformations [21] or weaving aspects into a base model [17, 22]. It is possible to check the different configurations of the system, represented by platform independent models (compliant with SCA) that can be visualized in a graphical editor. Finally, using our causal link, it is possible to adapt a system at runtime and switch from one configuration to another, without writing reconfiguration scripts.

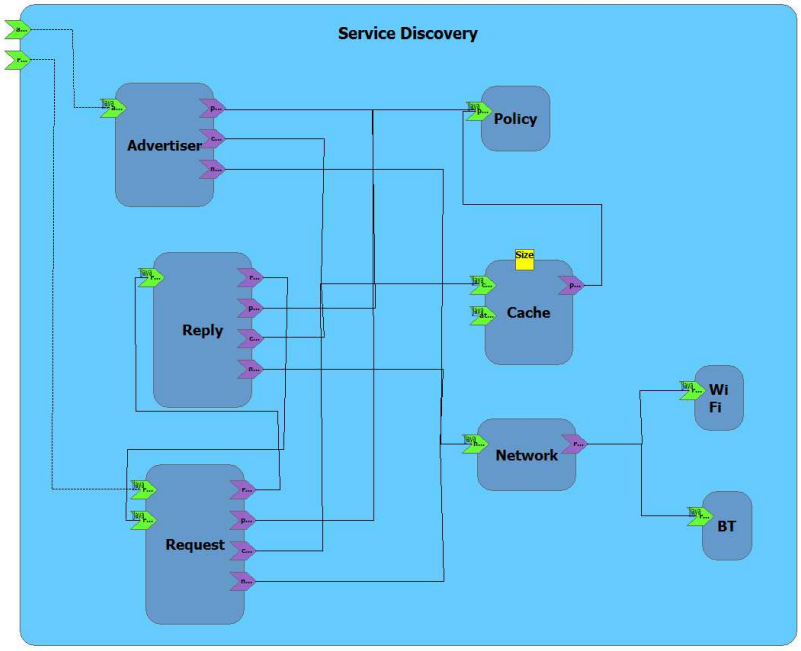

Fig. 5. Service Discovery Runtime Architecture

In future work, we plan to reuse an existing framework for monitoring interesting properties of the environment and develop a reasoning framework that will automatically select or generate (*e.g.*, by weaving some aspects into a base model) the most adapted configuration. After checking some constraints, our causal link will automatically reconfigure the running system. We plan to use the WildCAT monitoring framework [12] in combination with the Intel Mobile Platform Software Development Kit [7] that provides a set of implemented probes.

Acknowledgment

Brice Morin thanks the *Collège Doctoral International* of the *Université Européenne de Bretagne* for funding his visit to Lancaster University.

References

1. PEtALS: The Open Source ESB Solution for Services Oriented Architectures. http://petals.ow2.org/.
2. Service Component Architecture http://www.eclipse.org/stp/sca/.
3. Open Service Oriented Architecture http://www.osoa.org/.

4. EMF Compare http://www.eclipse.org/emft/projects/compare/.
5. Object Constraint Language Specification, version 2.0 http://www.omg.org/technology/documents/formal/ocl.htm/.
6. Fractal Explorer http://fractal.ow2.org/fractalexplorer/.
7. Intel Mobile Platform Software Development Kit. http://ossmpsdk.intel.com/ and http://sourceforge.net/projects/mpsdk.
8. N. Bencomo, G. Blair, and R. France. Models@run.time (at MoDELS) workshops. http://www.comp.lancs.ac.uk/bencomo/MRT.
9. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. An Open Component Model and Its Support in Java. In *CBSE'04: 7th Int. Symp. on Component-based Software Engineering*, pages 7–22, 2004.
10. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. 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):1257–1284, 2006. Fractal is available at: http://fractal.ow2.org/.
11. G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distrib. Comput.*, 15(2):109–126, 2002. OpenCOM is available at: http://sourceforge.net/projects/gridkit.
12. P.C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th Int. Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, Vienna, Austria, 2006.
13. F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.
14. C. Flores-Cortés, G. Blair, and P. Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. *IEEE Dist. Systems Online*, 8(7):1, 2007.
15. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4):93–95, 2008.
16. Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, October 2007.
17. B. Morin, O. Barais, and J. M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *VaMoS'08: 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.
18. B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.
19. B. Morin, F. Fleurey, N. Bencomo, J-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *MoDELS'08: 11th Int. Conf. on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.
20. B. Morin, J.Klein, O. Barais, and J. M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA@ICSE'08: Int. Workshop on Early Aspects*, Leipzig, Germany, May 2008.
21. P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer. Kermeta is available at: http://www.kermeta.org.
22. G. Perrouin, J. Klein, N. Guelfi, and J.M. Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC'08: 12th Int. Conf. on Software Product Lines*, 2008.