

Unifying Runtime Adaptation and Design Evolution

Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel,
Olivier Barais, Jean-Marc Jézéquel

▶ **To cite this version:**

Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais, et al.. Unifying Runtime Adaptation and Design Evolution. IEEE 9th International Conference on Computer and Information Technology (CIT'09), 2009, Xiamen, China, China. inria-00468517

HAL Id: inria-00468517
https://inria.hal.science/inria-00468517

Submitted on 31 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Unifying Runtime Adaptation and Design Evolution

Brice Morin[1], Thomas Ledoux[1,2], Mahmoud Ben Hassine[1], Franck Chauvel[3], Olivier Barais[1,4], and Jean-Marc Jézéquel[1,4]
[1]INRIA, Centre Rennes - Bretagne Atlantique
[2]Ecole des Mines de Nantes
[3]National Laboratory of High Confidence Software Technologies
School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, PRC.
[4]IRISA, Université de Rennes1
Campus de Beaulieu
35042 Rennes Cedex - FRANCE
{Brice.Morin | Thomas.Ledoux | Mahmoud.Ben_Hassine | barais | jezequel}@inria.fr
{franck.chauvel}@sei.pku.edu.cn

***Abstract*—The increasing need for continuously available software systems has raised two key-issues: self-adaptation and design evolution. The former one requires software systems to monitor their execution platform and automatically adapt their configuration and/or architecture to adjust their quality of service (optimization, fault-handling). The later one requires new design decisions to be reflected on the fly on the running system to ensure the needed high availability (new requirements, corrective and preventive maintenance). However, design evolution and self-adaptation are not independent and reflecting a design evolution on a running self-adaptative system is not always safe. We propose to unify run-time adaptation and run-time evolution by monitoring both the run-time platform and the design models. Thus, it becomes possible to correlate those heterogeneous events and to use pattern matching on events to elaborate a pertinent decision for run-time adaptation. A flood prediction system deployed along the Ribble river (Yorkshire, England) is used to illustrate how to unify design evolution and run-time adaptation and to safely perform runtime evolution on adaptive systems.**

I. Introduction

Society's increasing dependency on software systems is driving the need for robust, dependable, and continuously available systems. A very promising approach to the above issue is to implement systems as Self-Adaptive Systems (SAS) which include some self-adaptation facilities. SAS automatically adapt themselves at runtime according to the execution context in order to always provide the expected QoS [1].

In addition to their self-adaptation capability, SAS should still be opened to design evolution, because of requirements evolution, predictive or corrective maintenance. Recent research efforts on Model-Driven Engineering (MDE) such as Rainbow [2], ABCTool [3], Plastik [4] or DiVA [5] tackle the design evolution issue by building a causal connection between abstract design models and the running system. Thus, the designer can refine and update the conceptual models and then automatically synchronize them with the running system, thus preventing developers to write by hand ad-hoc and platform-dependent scripts [6].

However, applying such techniques (so called "models@runtime" [7]) to SAS raises a key-issue: how to ensure the consistency between changes resulting from design evolution and changes resulting from self-adaptation? Indeed, runtime self-adaptations are triggered by platform events describing the current state of the execution platform (memory levels, CPU load, network bandwidth, etc.). For instance, a design evolution corresponding to the addition of a new resource-consuming functionality might breakdown a SAS self-adaption decision to save resources.

The proposition of this paper is to unify design evolution and runtime adaptation by monitoring both the design model and the runtime platform in a homogeneous manner. It allows the designer to correlate design events reflecting the design evolution and runtime events reflecting the runtime self-adaptation and thus to make relevant adaptation decisions. Our monitoring framework combines three recent technological advances: EMF[8] to build and reason on models, WildCAT[1] [9] to define and organize the needed probes and Esper [10] to express complex queries over occuring events. This monitoring framework has been successfully used to perform evolution on the flood prediction system deployed along the Ribble River.

The remainder of this paper is organized as follows. Section II motivates this work through a toy and very intuitive case-study. Section III presents the Wildcat framework and how designer can use it to monitor runtime events, in the context of dynamic adaptations, and the modifications of the design model, in the context of design evolution. Section IV shows how design and runtime monitoring are combined and outlines how our approach was validated in the context of a sensor network system deployed along the river Ribble (Yorkshire, England) to carry out flood predictions. Section V discusses related work and section VI concludes by presenting a set of open research problems based on our experiment.

[1]http://wildcat.objectweb.org

II. The need for mixing runtime and design events

In this section, we propose to motivate the need for a uniform management of runtime adaptations and design evolutions with a very intuitive example. Let us consider a distributed system composed of a set of nodes with limited resources (memory, energy, CPU, etc), such as the flood prediction system that will be further detailed in Section IV. The distributed system includes a control loop to perform self-adaptation and therefore to dynamically adapt itself to its execution context, *i.e.* depending on runtime events such as memory is low, energy is high, etc.

We assume that the requirements of the system have changed. It should now integrate a new feature, realized by several components and connections between these components. The designer modifies the current architecture of the running system, at the model level, with his favorite graphical editor for example. Then, using a design-time validation tool, the designer can check that this updated configuration ensures well-formedness rules, etc. and generate the code of the new components. Once all these design and implementation tasks have been performed, the designer decides to commit these changes to the running system. This is automatically performed by the causal connection [5]: it transforms the changes between the initial configuration and the new configuration (at the model level) into a low-level reconfiguration script in order to reconfigure the running system.

However, if the free memory of the running system is currently low, the introduction of the component would causes some problems: some components could not be instantiated because there is not enough memory. In this simple example, we can see that design events (*e.g.*, adding a component) may be highly dependent from runtime events (*e.g.*, memory is low).

In conclusion, design evolution is not independent from the runtime. To fill the gap between design evolution and runtime adaptation, our approach consists in monitoring design and runtime events in an homogeneous way, so that design evolution and dynamic adaptation activities can interact seamlessly. In the example above, we would like to specify that if components are added in the architecture (a design event), the memory (a runtime event) should be sufficient. Otherwise, the design evolution should not be directly reflected to the runtime. In Section IV, we will illustrate the aggregation of design and runtime events in the context of the flood prediction system.

III. Unifying Runtime Adaptation and Design Evolution

WildCAT is a generic monitoring framework for developing context-aware applications [9]. It allows monitoring large scale applications by easily organizing and accessing sensors through a hierarchical organization. This section presents the meta-model of WildCAT to organize sensors, how it is used to build Context-Aware Applications, and its extension to monitor model evolution in Eclipse. We conclude this section with a short discussion on the approach.

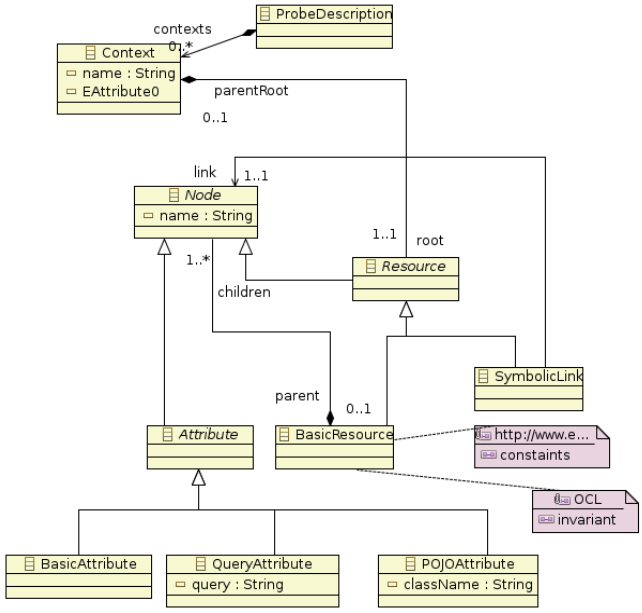


Fig. 1. WildCAT meta-model

A. WildCAT meta-model

The meta-model shown on Figure 1 allows developers to describe *monitoring models* by defining several *contexts*. Contexts are domain independant and can reprensent different aspects of the execution context: local hardware resources, the topology and performance of the network, geophysical informations, user preferences, etc. Each context is the root element of a hierarchy, similar to the Unix file system. We use EMF Eclipse Modeling Framework [8] to design this meta-model that defines the WildCAT main concepts.

A *context* is an oriented tree structure that contains two types of nodes:

- **Attribute**, which holds some values. Attributes are the leaves of the sensors tree (like files in a file system). WildCAT proposes three kinds of attribute:
  - **Basic attribute** holds static values. Their values do not evolve unless programmatically modified.
  - **Active attribute** or **POJOAttribute** represents WildCAT sensors. In general, these attributes are associated to probes linked to the environment or the execution context (CPU, Memory, Thermometer, Camera, etc).
  - **Synthetic attribute** or **Query Attribute** holds the results of expressions on other attributes. It can for example aggregate and transforms the values provided by other attributes *e.g.*, compute mean values during 10 seconds.
- **Resource**, which contains sub-resources and attributes (like folders in a file system).
  - **Basic resources** allows the designer to structure the monitoring model. For example, each node of a dis-

tributed system could be a basic resource, containing sub-resources related to the memory, the CPU, etc.

– **Symbolic links** are special resources that refer to another resource. Symbolic links are used to create "short cuts" in the monitoring model in order to access more rapidly to the important resources or attributes, without navigating the whole tree. An OCL constraint specifies that cycles are not allowed (constraint similar to inheritance cycle in UML or Java programs).

Based on this meta-model, the designer can graphically represent a monitoring model as illustrated in the right part of Figure 2. We use GMF[2] to define the graphical syntax of our meta-model. Then, using JET[3], we generate the Java code related to the integration of the WildCAT attributes and resources in the system.

B. WildCAT at runtime

In this sub-section, we describe how we can use WildCAT for monitoring runtime events. A typical hierarchy for runtime monitoring is illustrated in Figure 3.

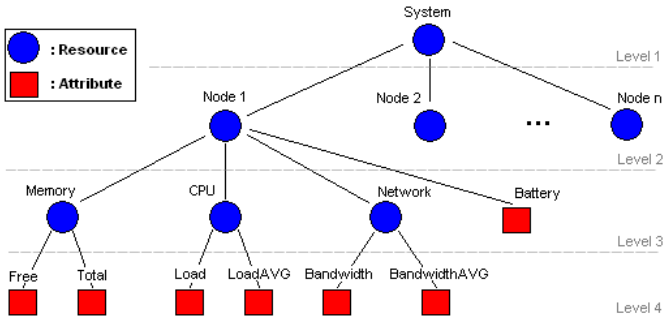


Fig. 3. Wildcat Runtime Hierarchy example

The system (Root resource) contains several nodes deployed in a distributed environment. Each node is represented by a basic resource (Second level). The resource of each node (*Memory*, *CPU* and *Network*) is represented as a sub-resource (Third level). Finally, each resource has some attributes describing its current status. For the Memory resource, the *Total* and *Free* attributes hold respectively the total and the free amount of available memory. The *CPU* resource is associated with the basic attribute *Load* which holds the instant CPU load (automatically updated) and the query attribute *LoadAVG* which holds the CPU load average for the last 15 seconds. The following code snippet shows how we create a basic attribute using the Context API:

```
1  /*
2   * Creating a context named "System" through the
        ContextFactory
3   */
4  Context ctx = ContextFactory.getDefaultFactory().
       createContext("System");
5  /*
6   * Creating the "Total" attribute of the Memory
        resource
7   */
8  ctx.createAttribute("self://Node1/Memory#Total");
```

Note that every missing resource in the current hierarchy is automatically added by the framework: the last code line implicitly creates the Node1 resource, the Memory sub-resource and the Total attribute. Query attributes such as LoadAVG and BandwidthAVG are defined as queries to the Esper framework [10], which is the WildCAT backend. Esper is an open-source Complex Event Processing engine providing interesting features such as events pattern matching, events correlation, sliding windows and a SQL-like language to write queries over triggered events. The code snippet below shows how query attributes are created and attached to the hierarchy:

```
1  /*
2   * Creating a query attribute holding the CPU load
        average for the last 15 seconds
3   */
4  QueryAttribute LoadAVG = new QueryAttribute("select avg
       (value.load) as loadAVG from
5  WAttributeEvent(source = 'self://Node1/CPU#Load').win:
       time(15s)");
6  /*
7   * Attaching the query attribute "LoadAVG" to the
        hierarchy
8  */
9  ctx.attachAttribute("self://Node1/CPU#LoadAVG", LoadAVG
       );
```

Finally, information such as total memory, CPU load or Network available bandwidth is easily retrieved thanks to WildCAT sensors. WildCAT sensors are designed to be generic probes since they are POJO (Plain Old Java Object) Attributes. This powerful WildCAT feature allows developers to design their own probes as Java classes and associate them to information source regardless of their nature (Network device, Camera, Thermometer, etc).

C. WildCAT at design-time

To both use design and runtime events in an homogeneous way, we extended WildCAT to monitor design evolution in the Eclipse Modeling Framework (EMF). EMF is an open source framework targeting Model-Driven Architecture development and can be considered the standard in the Model-Driven Engineering community.

Two problems must be solved in order to monitor the evolution of a EMF model with WildCAT:

- Create listeners to capture the evolution of the design model;
- Create a WildCAT hierarchy which only reflects the relevant changes that can affect the design model.

An interesting feature of EMF is the notification framework. EMF automatically provides notification functionality to the model in case of changes in the model. It is possible to register observers / listeners which are notified when the model is modified. There are six types of event: ADD, $REMOVE$, $ADDMANY$, $REMOVEMANY$, SET, $UNSET$ which are respectively raised when an element is added or removed in

[2]Graphical Modeling Framework [8]
[3]Java Emitter Template

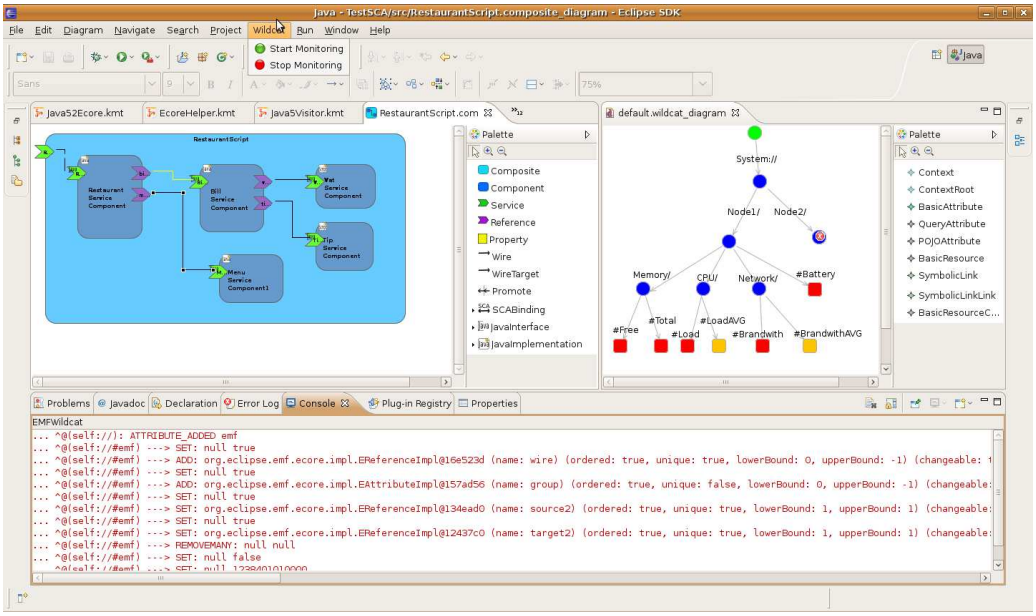


Fig. 2. WildCAT Designer Snapshot

the model, when a collection of elements is added or removed in the model or when an object property is set or unset.

A WildCAT POJO Attribute implements the EMF Observer interface and is registered to all the resources (models) of the opened editors in the Eclipse workbench. This way, the observer is automatically notified when a designer modifies a design model.

WildCAT hierarchies are built according to the concept of a domain meta-model. In our case, we are interested in software architectures made of components types with ports, component instances with bindings, etc. A fragment of this hierarchy is illustrated in Figure 4. Each meta-class of the domain meta-model becomes a node in the WildCAT hierarchy. Each node contains two sub-nodes: $FACTORY$ and SET. The $FACTORY$ node contains two attributes specifying when an instance of the corresponding meta-class is created or deleted. The SET node contains as many attributes as there are properties or references in the meta-class.

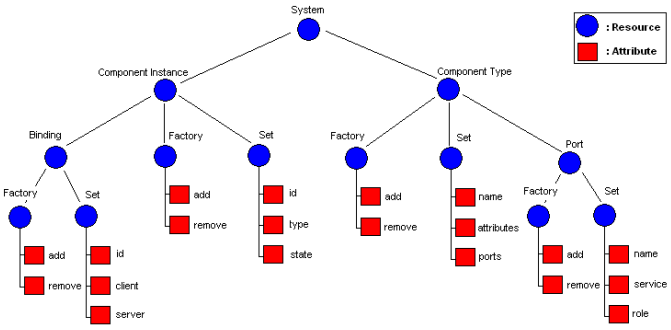


Fig. 4. System Architecture meta-model

D. Discussion

In this section, we have presented the meta-model of WildCAT. This meta-model allows designer to specify monitoring hierarchies. We illustrated how to define such a hierarchy of sensors for monitoring runtime events, in the context of self-adaptive systems. Based on an Event-Condition-Action framework like Safran [11], these runtime events can trigger reconfiguration scripts [6] in order to adapt the running system depending on the context. We explained how we can monitor architectural models with another WildCAT hierarchy, by leveraging the mechanisms provided by EMF. We rely on an existing "models@runtime" approach [5], [12], [13] to reflect design evolution by reconfiguring the running system. The way the connection between the design model and the running system is realized is beyond the scope of this paper.

In the next section, we will use the runtime and the design hierarchies in concert, in order to control when (*i.e.* in which runtime context) design evolutions can actually be applied to the running system. Using WildCAT, it is now possible to make both activities interact in a seamless way thanks to a unified monitoring.

IV. Validation on the Flood Prediction System

In this section, we show how we aggregate runtime (Section III-B) and design (Section III-C) events during some design evolutions. Our approach is applied on the real-life Flood Prediction System (FPS) [14] in order to specify when design evolutions can actually be reflected to the runtime system. Since human lives depend on this system, it should only be adapted when really needed or when no flood is likely to appear.

In the first subsection, we briefly introduce the system. In the second subsection, we detail several examples of design

evolution that use design and runtime events to specify when these evolutions can actually be applied to the running system.

A. An Overview of the Flood Prediction System

The FPS is currently deployed on the Ribble River (Yorkshire, England) in order to carry out flood predictions and notify local authorities from possible dangers. The FPS is composed of a set of nodes deployed along the river. Each node contains some sensors (*e.g.* cameras to evaluate water speed and depth, etc.), a battery, solar panels, as well as communication equipments (*e.g.* WiFi and/or Bluetooth antenna). All these equipments are controlled by software components, which compute flood prediction level according to the values provided by sensors, communicate the predictions to a central node or to a nearby node, etc.

The requirements of the system [15], [14] specify three relevant contexts:

- **No flood**: No flood is predicted in a close future. Most of the sensors are disabled (only one camera is active), the system uses low power consuming network protocols (also less efficient), so that the solar panels can quickly reload the battery;
- **Flood predicted**: A flood is likely to appear. Some nodes indicates alarming values (*e.g.* water speed is increasing). Some sensors are activated in order to provided more accurate predictions;
- **Flood**: A flood is currently predicted. All the available sensors are activated and the system uses efficient (also power consuming) network protocols, so that local authorities can be notified quickly, with accurate data.

The probes for monitoring the river and computing predictions (with aggregation of runtime events) are organized into a WildCAT hierarchy. Based on an Event-Condition-Action rules engine like Safran [11], dynamic adaptations (activate new sensors, use efficient networking algorithms, etc.) are triggered by WildCAT events.

B. Design Evolution of the Flood Prediction System

In this subsection, we illustrate two design evolutions of the Flood Prediction System whose application to the running system is highly dependent on runtime events. The first one illustrates a corrective maintenance, which aims at improving the accuracy of the predictions. The second one is a design evolution realized after the integration of a new physical node (sensors, antenna, battery, solar panels, etc), in order to deploy all the software components needed to administrate the node (data processing, prediction algorithms, routing and networking algorithms, etc).

1) A corrective maintenance: From the model, it is possible to increase the sampling rate of the sensors (or decrease this rate) after the system has been deployed, in order to have more accurate predictions. The designer simply modifies the corresponding attributes in the model. Then, using an automatic causal connection [5], this change is automatically reflected to the running system. This evolution is very simples: no components are bound, added or removed, a set method is simply invoked.

However, a significant augmentation of the sampling rate will inevitably consume more energy and reduce the battery level. If the sampling rate is augmented, the energy consumption increases proportionally. This is why this design evolution (performed on the model) should not be committed via the causal connection if the battery level (monitored at runtime) is currently low.

To realize this constraint mixing design and runtime events, we define the following query in WildCAT:

```
1  select * from pattern[
2     every B=WAttributeEvent(source like 'FPS://Node%#
          Battery')
3     ->
4     every A=WAttributeEvent(source = 'System://
          ComponentInstance/Attributes#Set')
5  ]
```

A listener is registered on this query. Every time the battery status change (*e.g.*, from medium to low) in the 'FPS' run-time context, and every time an attribute is set in the 'System' design context, the listener is notified by a call its `update(EventBean[] newEvents, EventBean[] oldEvents)` method (an Esper API). Using the old and new events, the listeners checks if the augmentation of the rate attribute is limited (less then 5% for example) and if the battery level is not low. If this condition is true, then the sampling rate is actually augmented on the running system.

2) A new requirement: When a new node is physically deployed on the river, because another node has been destroyed, it is not yet controlled by the flood prediction software system. All the components needed to get the date from the sensors, to compute flood prediction, to send data to other nodes, etc. are uploaded later.

In a graphical editor, the designer simply creates a composite component in the architectural model, which contains all the software components for managing the physical sensors, performing local predictions, sending the result, etc. Such a design evolution has a great impact on the system: it introduces some components and bindings and could only be reflected to the running system if there is enough free memory, as explained earlier.

Moreover, the introduction of these software components would make the routing and networking algorithms to compute new paths including the new node, thus disturbing the whole system. Note that when a node is not yet controlled by software components, it cannot be "seen" by the other nodes. Such design evolutions should only be deployed when no flood is predicted (running the algorithms will not disturb the system), or when the accuracy of the prediction is low, so that the integration of the new node can improve the accuracy after routing paths have been established.

In order to check if this design evolution can be actually reflected to the running system, we define the following query in WildCAT:

```
1  select * from
2     WAttributeEvent( source='System://ComponentInstance/
          Factory#add') as addComponent ,
3     WAttributeEvent( source='FPS://Prediction#noFlood')
          as noFlood ,
4     WAttributeEvent( source='FPS://Accuracy#accuracyLow
          ') as accuracyLow
5  where
6  addComponent.value and (noFlood.value or accuracyLow.
       value) ;
```

V. Related Work

As stated in the introduction, our approach is based on models@runtime techniques such as Rainbow [2], ABCTool [3], Plastik [4] which aims at ensuring the synchronization between a running system and its related design models. Rainbow and Plastik focus on the development of SAS by ensuring that self-adaptation does not conflict with static constraints developed at design-time. Our tool is complementary: it checks that an evolution of the design does not conflict with current self-adaptation. The ABCTool enables design-evolution by synchronizing an architecture model and its JEE implementation but do not detect conflicts while applying design evolution on a SAS.

Several works [16], [17] propose to monitor heterogeneous contexts enabling the exchange of relevant contextual information across and between domains. However, our work differ in that we provide a complete monitoring framework combining several aspects such as CEP (Complex Event Processing).

VI. Conclusion

This paper describes how to unify design evolution and runtime adaption of dynamically adaptive systems in order to build consistent adaptation decisions. On the conceptual side, our approach combines Complex Event Processing, Model Driven Engineering and Context-aware systems. This results in a monitoring framework able to deal with both design evolution events and run-time platform events in a homogeneous manner. On the technical side, this framework alleviates the tedious and error-prone handmade developments needed to aggregate design and runtime events. The combination of Wildcat's probes with the Esper's queries and the EMF's design events provides an efficient way to produce events agregators. Applied on the flood prediction system deployed along the Ribble river, our framework succesfully allows designer to make consistent adaptation decisions.

In future works, we plan to extend our approach following two main axis. On one hand, making explict the development process using SPEM could help monitoring the various roles and tasks. On the other hand, to take the most of the resulting complex events, the monitoring framework must be connected to an automatic decision-engine to support the automatic detection of potential conflicts between runtime-adaptation and design evolution.

VII. Acknowledgment

This work was funded by the DiVA FP7 European project (STREP) on Dynamic Variability in Complex Adaptive Systems (See http://www.ict-diva.eu/), the S-Cube European Network Of Excellence on Software Services and Systems (See http://www.s-cube-network.eu/) and the GALAXY INRIA cross-project (See http://galaxy.gforge.inria.fr/).

References

[1] P. Oreizy, N. Medvidovic, and R. Taylor, "Architecture-Based Runtime Software Evolution," in *ICSE'98: 20th Int. Conf. on Software Engineering*, Washington, DC, USA, 1998.

[2] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[3] H. Mei, G. Huang, L. Lan, and J. Li, "A software architecture centric self-adaptation approach for Internetware," *Science in China Series F: Information Sciences*, vol. 51, no. 6, pp. 722–742, 2008.

[4] A. Joolia, T. Batista, G. Coulson, and A. T. A. Gomes, "Mapping adl specifications to an efficient and reconfigurable runtime component platform," in *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 131–140.

[5] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel, "Taming Dynamically Adaptive Systems with Models and Aspects," in *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.

[6] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration fractal architectures," *Annals of Telecommunications, edited by Springer-Verlag France*, vol. 64, no. 1-2, January-February 2009, special issue on Component-based architecture: the Fractal initiative.

[7] N. Bencomo, G. Blair, and R. France, "Models@run.time (at MoDELS) workshops," www.comp.lancs.ac.uk/ bencomo/MRT/.

[8] *Eclipse development using the graphical editing framework and the eclipse modeling framework*. Riverton, NJ, USA: IBM Corp., 2004.

[9] P. David and T. Ledoux, "WildCAT: a generic framework for context-aware applications," in *MPAC'05: 3rd Int. Workshop on Middleware for Pervasive and Ad-hoc Computing*. New York, NY, USA: ACM, 2005, pp. 1–7.

[10] T. Bernhardt and A. Vasseur, "Esper: Event Stream Processing and Correlation," *O'Reilly OnJava.com*, 2007, http://www.onjava.com/pub/a/onjava/2007/03/07/esper-event-stream-processing-and-correlation.html.

[11] P. David and T. Ledoux, "An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components," in *SC'06: 5th Int. Symposium on Software Composition*, ser. Lecture Notes in Computer Science, vol. 4089, Vienna, Austria, 2006, pp. 82–97.

[12] 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 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[13] B. Morin, O. Barais, and J.-M. Jezequel, "K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines," in *Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08*, Toulouse, France, oct 2008.

[14] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven, "Gridstix:: Supporting flood prediction using embedded hardware and next generation grid middleware," in *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[15] H. Goldsby, P. Sawyer, N. Bencomo, B. Cheng, and D. Hughes, "Goal-Based Modeling of Dynamically Adaptive System Requirements," in *ECBS'08: 15th IEEE International Conference on the Engineering of Computer Based Systems*. Belfast, Northern Ireland: IEEE Computer Society, 2008, pp. 36–45.

[16] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," *Hum.-Comput. Interact.*, vol. 16, no. 2, pp. 97–166, 2001.

[17] H. van Kranenburg and H. Eertink, "Processing heterogeneous context information," Jan. 2005, pp. 140–143.