A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service

Audrey Occello[1], Anne-Marie Dery-Pinna[1], Michel Riveill[1]

I3S Laboratory, France
{occello, pinna, riveill}@polytech.unice.fr

Abstract. Dynamic software adaptations may lead application execution to unsafe states. Then, the detection of adaptation-related errors is needed. We propose a model-based detection in order to define a solution that can be used in multiple platforms. This paper is intended to show how a runtime model for monitoring adaptation safety may look like and how it may be concretized in order to interact with real applications.

Keywords. Runtime models, adaptation safety, monitoring, service.

1 Introduction

Software systems are becoming more complex as they are composed of distributed components using a variety of devices and platforms to deliver services to mobile end-users. Such complexity also increases the need of adapting these systems to feet the changing environment. As these adaptations involve changing the structure or the behavior of applications, they may lead application execution to unsafe states. For example, a functionality may be removed accidentally when removing a component or undesired cycles may be introduced in new interactions between components. The detection of adaptation-related errors is needed to control adaptation safety.

As well as Medicine can be preventive or curative, we can choose to repair or anticipate adaptation-related errors. Repairing an executing system is difficult because it assumes that we can resume to an earlier state of the application and often force to freeze a part or even the whole application execution. We believe that freezing the application while it is adapted in order to detect errors is not a good remedy for highly available applications. We prefer to monitor the applications and prevent unsafe adaptations to occur.

Models are usually used at design time to capture architectural decisions and to ensure a common understanding. Such information is generally not available at runtime because it is lost when the model elements are transformed into runtime artifacts. However, runtime models have been used for decades in the meta-programming research community [1] in the form of meta-object protocols [2] enabling systems to control themselves during their execution. As Muller and Barais said, "automatic or even self-adaptability of the running system may be achieved by taking decisions based on monitoring information captured by runtime models" [3]. This can be applied to our problem: a system can take the decision of accepting or discarding an adaptation request by monitoring the history of adaptations that have been already performed on it.

We adopt a Model Driven Engineering [4] approach in order to define a monitoring solution that can be used in multiple platforms and whose abstraction makes it possible to reason about the solution correctness as a one-time cost. In previous work, we detailed what kind of adaptation-related verifications are carried out [5], [6], how we modeled adaptation safety [7] and how we established the correctness of our modeling [8]. In this paper, we show how a runtime model for monitoring adaptation safety may look like and how it may be concretized in order to interact with real applications.

2 How does a runtime model for monitoring adaptation safety look like?

The main elements of our runtime model, called Satin are depicted in Figure 1. To illustrate these elements, we will take, as a running example in the next section, an application made of diaries.

2.1 Software entities to be monitored

Components

In Satin, a **component** represents any software instance (unit of execution) composing the application and that is capable of exhibiting its interface(s): object, component, service, etc. Only the information about the identity and implemented interfaces of such software instance is reified (we do not deal with the software instances' state).

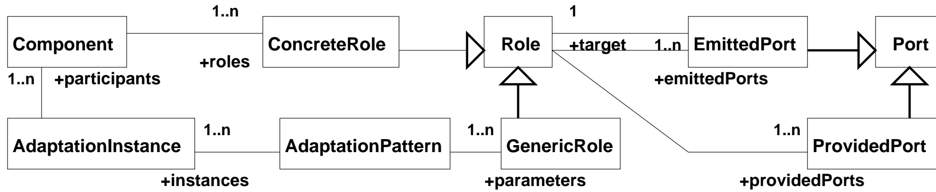

Fig. 1. Satin runtime model overview

Consider *lizDiary* and *johnDiary*, two diary instances of our running example, implementing a same interface with the add, remove, get and query meeting functionalities. At the Satin model level, *lizDiary* and *johnDiary* are two instances of the **Component** class.

Component roles

A Satin component is associated with **roles** (see the `ConcreteRole` class in Figure 1) which reify information about the software instance's implemented interfaces. Roles are composed of **ports** which are abstractions of operations provided by the component (see the `ProvidedPort` class in Figure 1) or required by one of the component adaptations (see the `EmittedPort` class in Figure 1). The interface implemented by the diary components is represented by the *BasicDiary* role. *BasicDiary* provided ports are: *addMeeting*, *removeMeeting*, *getMeeting*, *isFree* corresponding to functionalities that are present in the component interface at the platform level.

At any time, the roles of a component reflect its interactions with the environment. At the beginning, components' roles have only provided ports. When a component is adapted, its roles evolve and emitted ports are added to these roles. Roles are not shared between components. *lizDiary* and *johnDiary* have their own *BasicDiary* role and they evolve independently.

Adaptation patterns

In platforms, runtime adaptations can be structural and behavioral and consists of a set of elementary actions, which can express: 1) addition, withdrawal or replacement of components within an assembly, 2) addition and withdrawal of component ports, or 3) behavior modification of component ports.

In Satin, an **adaptation pattern** represents the unit of application and reuse of platform adaptations. This concept is used to reproduce at the model level the modifications performed for an adaptation at the platform level. For example, the code below depicts an adaptation pattern that contains an elementary action modifying the behavior associated with an *addMeeting* port in a diary application.

```
adaptationPattern Synchronization(SynchronizingDiary d1,
                                  SynchronizedDiary d2) {
   modifyPort d1.addMeeting(Meeting m)
     -> if (d2.isFree(m)) then
           d1._call(m); d2.addMeeting(m)
        else
           d1._call(m); d2.printError('Not synchronized')
        endif
}
```

The *Synchronization* adaptation pattern expresses that any addition of a meeting to a synchronizing diary also involves an addition of the meeting to the diary to be synchronized. If this is not possible because the time slot is not free, a synchronization conflict is reported. Note that the *_call* expression refers to a built-in delegation (like *proceed* in AspectJ [9]) that invokes the prior, non-adapted version of *addMeeting*.

Adaptation pattern roles

An adaptation pattern is defined on a set of roles. An **adaptation pattern role** (see the `GenericRole` class in Figure 1) specifies the ports that a component must provide (see the `ProvidedPort` class in Figure 1) or requires (see the `EmittedPort` class in Figure 1) to play this role in an adaptation. For instance, in the previous example, the *Synchronization* adaptation pattern expects two parameters with two different roles: The first parameter must conform to the *SynchronizingDiary* role by providing at least a port that conforms to *addMeeting*. The second parameter must conform to the *SynchronizedDiary* role by providing at least ports that conform to *addMeeting*, *isFree* and *printError*.

2.2 The monitoring criteria: Safety properties

Now that the software entities to monitor are well defined, we need to specify what to monitor. In our case, the monitoring depends on the criteria of safe adaptations. For that, we have identified a set of *safety properties*, which cover a large range of errors: from "assembly inconsistencies" [10], "message not understood" [11] local errors up to more global errors such as "adaptation composition conflicts" [12], "synchronization" and "divergence" [13]. Each safety property is guaranteed by a set of OCL (Object Constraint Language) [14] constraints (operation preconditions) attached to the Satin runtime model.

We suppose that the application initial state is safe (each component and the initial assemblies of these components are safe). If the OCL constraints are checked regarding the adaptation to perform then the adaptation can proceed and we guarantee that the application state remains safe after the adaptation.

Next section explains how to use the Satin runtime model in platforms offering adaptation facilities.

3 Making the model available at runtime

Two approaches can be considered to concretize the model. A first approach consists in adding safety code corresponding to the safety properties in a platform by extension of its model. This is done in two steps: 1) we must map the elements of the Satin model to elements of the target platform, 2) once the target element is identified, we must generate the safety properties's constraints on them [5]. This solution has several drawbacks. First, there is not always a one-to-one correspondence between Satin model element and platform elements. In this case, the generation of the safety properties's constraints is more complicated. Secondly, the minimization of software defects being an important issue in critical application areas, we have used validation and verification techniques to ensure the model correctness with a formal foundation [8]. However, such verifications and validations are lost with this approach since the constraints are regenerated at the platform level: mapping to N platforms implies N revalidations.

Another approach consists in populating the Satin model with platform-dependent application information first and then checking for adaptation safety at the Satin model level. For this, the model is made available at runtime as a service. A service protocol formalizes the way the service can be used and how the service communicates with the platforms. The safety service tells whether an adaptation of an application A is safe or not according to the operations offered by the components and according to the previous adaptations of A, stored as one goes along. This helps to prevent adaptations that would lead the application to an unsafe state. The difficulty with this approach is that we have to manage platform diversity as there is no mapping step. But the gain is that one implementation of the service makes it possible to use the model with several platforms. Moreover, the results of the verification and validation step can be preserved or need to be performed again only once at most.

Section 3.1 describes the service built on top of the Satin model. Section 3.2 explains how we take into account platform diversity.

3.1 Service description

The safety service can be used according to the following process. At each step, a subset of OCL constraints is checked in order to detect if a platform adaptation operation violates one of the safety properties. Figure 2 presents the overall service architecture.

1. Components are registered to the service in order to have a partial representation of the application to monitor. However, the step can be delayed to the first time a component is being adapted (step 3): only the components implied in an adaptation need to be represented in the service state (at the Satin runtime model level).
2. The description of the adaptation to perform at the platform level is registered to the service as an adaptation pattern.
3. An adaptation pattern is applied to a list of components in order to perform at the model level the adaptation triggered at the platform level. The components are registered if not already done.
4. The adaptation pattern of step 3 is unapplied in order for the modifications applied to components to be undone if requested at the platform level. This step is optional.

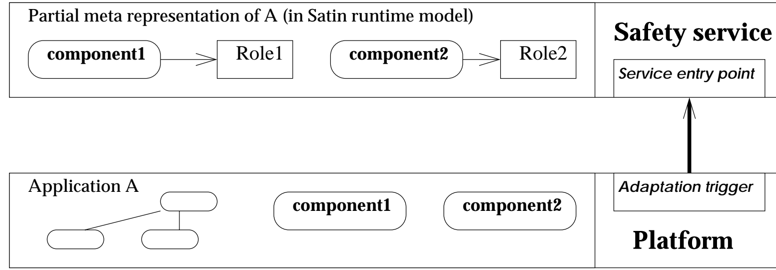

Fig. 2. Safety service architecture : component adaptation and service query

Combining step 3 and step 4 makes it possible to replace components in assemblies. Steps 3 and 4 modify the state of the service to memorize the adaptation of components. These steps are necessary to synchronize the platform and the service so that the history of adaptations is equivalent in the service and in the platform.

The service entry point is described by an IDL Corba interface that allows a platform to use the service without thoroughly knowing its internal mechanisms. Each interface operation is associated with a step of the process: (un)registration of components, creation/destruction of adaptation patterns, (un)application of adaptation patterns, the replacement of components in assemblies.

3.2 Parametrization of service

To evaluate the OCL constraints preserving the safety properties, the service needs information about the monitored application. The way of retrieving and interpreting such information varies from one platform to another one. To take into account platform specificities, the service needs to be parametrized.

Data extraction parametrization point. To populate the model with information about the application, some data need to be extracted. The first time a component is involved in an adaptation at the platform level, a corresponding component is created at the model level (step 1). The initial component roles are deduced from application types using introspection mechanisms which depends on the platform and language.

Type checking parametrization point. Before accepting the application of an adaptation pattern to a set of components (step 3),

each component must conform to the role it has to play in the adaptation. Conformance can be based on different syntactic, behavioral or quality of service criteria [15]. The supported conformance model depends on the platform and language.

Adaptation introspection parametrization point. Applying an adaptation pattern to components (step 3) can be done only if the elementary actions to apply are compatible with the adaptations already applied to those components. The types of elementary actions supported differ from a platform to another. The ways in which the elementary actions are expressed and implemented also vary according to the platforms.

These three parametrization points correspond to the entry points for message exchanges with the platforms and are used by the service to retrieve platform specific information. As for the service entry point, these entry points are formalized by IDL Corba interfaces. Each platform has to provide implementation for these interfaces. Configuring the service for a specific platform consists in specifying which implementation objects the service must use.

A prototype has been implemented in Java. The OCL constraints are not translated into Java code but are interpreted using the Dresden-OCL toolkit [16]. Then a new validation and verification step is not needed provided that the toolkit interpreter preserves the semantics of OCL. Note that even if the prototype is currently used with Java platforms, it can be easily exposed as a web service to interact with non-Java platforms.

4 Discution and future work

The Satin model for adaptation safety monitoring is made available to adaptive platforms as a generic service. Then, the detection of adaptation-related errors is externalized and consists in querying the service to check whether an adaptation request will break the application execution or not. However, the detection of adaptation-related errors can also be done at two other levels. This can be put in place by application developers for each adaptation of their application. This is not a fair solution as the developer is not an expert of the domain. It is also error-prone as each adaptation has to be managed on a case by case basis. It can also be put in place by

the platform that provides the adaptation facilities such as Fractal [17], Sofa [18], JAC [19], Compose* [20] or Noah [21]. This task can then be delegated to an expert of the domain. Though, most of such platforms still lack of formal foundations or force to freeze a part of the application making the hosted applications unavailable to their users. Lastly, these solutions cannot be reused in other platforms.

Self-adaptive systems such as COMPAA [22], PLASMA [23], RAINBOW [24] and SAFRAN [25] monitor and adapt themselves to system errors, changes in the environment or in user preferences. In such systems, adaptations are performed at a model level like in Satin. However, Satin is about introspection not intercession: Satin does not modify the behavior of the monitored application. Model-level adaptation is only done for synchronization purpose between the base level (platform hosting the application) and the model level (Satin safety service). Without the consistency between these two views, the service is not able to compute the adaptation safety anymore. Then Satin should not be seen as a self-adaptive system but as a mean for self-adaptive systems to prevent erroneous adaptations.

Self-adaptive systems detect "changes" and react to changes by "repairing" themselves. In such systems, repairing means adapting the application to the "new context". In our case, repairing only consists in warning the monitored application when an adaptation request is not safe but Satin does not correct the adaptation. In future work, we plan to contribute to decision-making and propose error-free adaptation alternatives when an adaptation request is rejected by the safety service.

References

1. Affer, J.M.: Meta-level programming with coda. In: 9th European Conference on Object-Oriented Programming, Springer-Verlag (1995) 190–214
2. Kickzales, G., deRivières, J., Bobrow, D.G.: The Art of Meta Object Protocol. MIT Press (1991)
3. Muller, P.A., Barais, O.: Control-theory and models at runtime. In: Proceedings of the Models Workshop on Models@Runtime, Nashville, USA (oct 2007)
4. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006) 25–32
5. Occello, A., Dery-Pinna, A.M.: An adaptation-safe model for component platforms. In: Proceedings of the 3rd International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04), France (2004) 169–174
6. Occello, A., Dery-Pinna, A.M., Riveill, M.: Safety as a service. Journal of Object Technology (2009) to appear in March/April issue.

7. Occello, A., Dery-Pinna, A.M.: Safe runtime adaptations of components: a UML metamodel with OCL constraints. In: First International Workshop on Foundations of Unanticipated Software Evolution, Barcelona, Spagna (2004)
8. Occello, A., Dery-Pinna, A.M., Riveill, M.: Validation and Verification of an UML/OCL Model with USE and B: Case Study and Lessons Learnt. In: Fifth International Workshop on Model Driven Engineering, Verification, and Validation, Lillehammer, Norway, IEEE Digital Library (2008)
9. Kiczales, G., Lamping, J.: Aspectj home page. http://eclipse.org/aspectj (2001)
10. Adámek, J., Plasil, F.: Partial bindings of components - any harm? In: 11th Asia-Pacific Software Engineering Conference, IEEE Computer Society (2004) 632–639
11. Cardelli, L.: Type systems. ACM Computing Surveys (1996) 263–264
12. Hanneman, J., Chitchyan, R., Rashid, A.: Analysis of aspect-oriented software workshop report. Technical report, University of California, Germany (2003)
13. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: 12th ACM Symp. Principles of Programming Languages, New Orleans, LA, USA (1985) 97–107
14. Warmer, J., Kleppe, A.: OCL: The constraint language of the UML. Journal of Object-Oriented Programming (1999)
15. Beugnard, A., J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. In: IEEE Software. (1999) 38–45
16. Wiebicke, R.: Utility support for checking ocl business rules in java programs. Master's thesis, TU-Dresden (December 2000)
17. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of WCOP. (2002)
18. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: Proceedings of ICCDS'98, USA (1998)
19. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible and efficient solution for aspect-oriented programming in java. In Yonezawa, A., Matsuoka, S., eds.: Reflection. Volume 2192 of LNCS., Springer-Verlag (2001) 1–24
20. Garcia, C.F.N.: Compose*: A runtime for the .Net platform. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands (2003)
21. Blay-Fornarino, M., Charfi, A., Emsellem, D., Pinna-Dery, A.M., Riveill, M.: Software interaction. Journal of Object Technology **10**(10) (2004)
22. Aniorté, P., Lacouture, J.: Compaa : A self-adaptable component model for open systems. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), Belfast, Northern Ireland, IEEE Computer Society (2008) 19–25
23. Layaïda, O., Hagimont, D.: Plasma : A component-based framework for building self-adaptive applications. In: SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications, San Jose, CA, USA (January 2005)
24. Cheng, S.W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. PhD thesis, Carnegie Mellon University (May 2008)
25. David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive fractal components. In Löwe, W., Südholt, M., eds.: International Workshop on Software Composition (SC). Volume 4089 of LNCS., Vienna, Austria, Springer Verlag (March 2006) 82–97