JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 8, No. 2, March–Aprile 2009

# Safety as a Service

**Audrey Occello**, **Anne-Marie Dery-Pinna**, **Michel Riveill**, Engineering School of Technology of the University of Nice, France

Application adaptations can involve changing the stucture or the behavior of applications. When performed at runtime, such adaptations may lead application execution to unsafe states. It arises in component and service-oriented platforms as well as aspect-oriented frameworks that support run-time adaptation. However such platforms hardly manage adaptation-related errors. In this paper we propose a generic safety service that can be used with different platforms as the need to determine safety of run-time adaptation is independent of the underlying technology.

## 1 INTRODUCTION

Software application complexity induces important risks of bugs or unpredicted behaviors resulting from interactions between application subsystems. Static analysis techniques are used to verify that no bug or undesired behavior will occur at runtime. Type-checking [12] detects invalid code. Most of Architecture Description Languages (ADLs) [24] ensure assembly soundness for initial constructions of the application architecture. Model-checking [22] ensures that "bad things do not happen on all executions of a system and that good things eventually happen on all executions of a system" [21]. Theorem proving [9] verifies invariant preservation to check, for example, whether an operation behaves as predicted.

Component and service-oriented platforms [10, 34, 27, 31] permit to change, add or remove components in assemblies. Aspect programming [32, 8, 16] allows for changing the set of actions associated with a functionality through the weaving and unweaving of aspects. We call *runtime adaptation* such kind of dynamic modification of applications. Runtime adaptations may generate bugs and unpredicted interactions leading the application execution to an unsafe state. For example, a functionality may be removed accidentally when removing a component or undesired cycles may be introduced in new interactions between components. Then static program verifications are not sufficient, new checks need to be performed at runtime to control adaptation safety. This can be done, for example, by application developers for each adaptation of their application. This is error-prone as each adaptation has to be managed on a case by case basis.

The design of distributed applications converges towards the use of middleware platforms that manages application complexity. Such platforms offer a separation between functional and extra functional aspects named services. For example, EJB-based platforms [35] such as Jonas [28] and CCM-based ones [17] such as OpenCCM

[27] provide transaction, security and persistence services. In the same way, platforms with adaptation capabilities [27, 32, 10, 8, 34, 31, 33] can manage adaptation safety in the middleware implementation of non functional features in a more systematic way. Adaptation safety checking can be reused for any application that runs on a given platform. Though, most of these platforms handle only a subset of adaptation safety errors and still lack of formal support to control such adaptations.

Even if adaptations are implemented differently in each platform, adaptation safety checking can be abstracted away and can be modeled independently of execution supports. In this paper, we propose to handle adaptation safety as a platform-independent "service" in the context of a Model Driven Engineering (MDE) [37] approach as it has been done for trader [23], transactions [36], persistence [40] and deployment [15] services.

The remainder of this paper is organized as follows. Section 2 describes some kinds of errors related to adaptations and how they are handled in middleware platforms. Section 3 presents the architecture of the service and its protocol. Section 4 shows a service implementation and explains how to configure the service to use it with two different platforms. Section 5 concludes and identifies future work.

## 2 ADAPTATION SAFETY

This section presents some middleware platforms offering dynamic adaptation capabilities. It also presents the errors that can arise when using such capabilities.

### Adaptation handling in middleware platforms

This section presents middleware platforms that are quite well-known and that handle adaptations dynamically. There are three more frequently encountered kinds of adaptations: 1) interface evolution (adding or removing functionalities to the set of functionalities that a component handles), 2) behavioral composition (modifying the behavior of a functionality by composing new actions with the existing ones) and 3) assembly modification (adding or removing bindings between components or replacing components by other ones in assemblies).

**CCM/OpenCCM:** The CORBA Component Model [17] is an evolution of the CORBA object model [38] dedicated to the design, produce, deploy, and run distributed heterogeneous component based applications. According to the CCM specification, it is dynamically possible to create CORBA component instances and to interconnect these instances by means of assemblies. Assembly information is defined to be static: the Component Assembly Descriptor determines which component types to use for interconnections. However, it is possible to change CORBA component instances to use for a given connection dynamically as in the first CCM implementation, OpenCCM [27]. OpenCCM supports only *assembly modification.*

**Fractal/Julia:** Fractal [11] is a programming language-agnostic component model. Various programming languages can be used to design, implement, deploy and reconfigure components dynamically without affecting the philosophy of the model. Julia [10] is the reference implementation of Fractal. A Fractal component is made of two parts: a content that manages the functional concerns, and a membrane of built-in and user-defined controllers, that manages non functional concerns (security, transactions, . . . ). As the model is hierarchical, the content part of a Fractal component can be made of other components and can be nested at arbitrary levels. The membrane part of a Fractal component contains specific built-in controllers that provides the API to adapt Fractal components. Using Julia, content and binding controllers make it possible to perform *assembly modification*.

**SOFA/DCUP:** Software Appliances [34] is a component model which allows an application to be composed of a set of dynamically updatable components. As Fractal, SOFA is a hierarchical model. The strength of the SOFA model is in the use of behavioral protocols [1] describing component usage. The behavior of a SOFA component corresponds to the set of all traces that can be produced by the component. DCUP (Dynamic Component UPdating) is the SOFA layer allowing for dynamic component replacement at runtime which is part of the *assembly modification* kind of adaptations.

**OSGi:** The Open Service Gateway initiative [31] is a service-oriented specification dedicated to the construction of Service Oriented Architectures (SOA) [26]. OSGi services are delivered and deployed in units called bundles. A bundle is either a service provider or service consumer and corresponds to an OSGi component. The specification defines administration mechanisms (installation, activation, deactivation, update and uninstallation of components). For example, the Wire Admin Service is responsible for the connection (wire) between providers and consumers. OSGi components can directly react on the appearance and disappearance of services by dynamically discovering each others. Then OSGi supports *assembly modification*.

**Noah:** Noah [8] is a framework allowing programmers to express the interactions between heterogeneous software entities (Java, EJB [35] and .NET [25]) declaratively and externally via the ISL language [5]. By describing interaction rules between software entities, programmers can modify their behavior dynamically. A merging operation based on ISL operators is used to compose a set of rules applied to a software entity. Then, Noah enables *behavior composition*.

**Composition filters/Compose*:** The composition filter model (CF) [6] is an extension of the object-oriented model. Some implementations of the model offer opportunities in terms of dynamic adaptations such as Sina [20] (on top of Smalltalk) and Compose* [16] (on top of .NET). CF's central construction, the filter, permits the modification of object structure and behavior. Each object includes two ordered collections of filters, one for received messages and the other for sent messages. All incoming and outgoing messages are intercepted and submitted to the filters before being possibly processed by the object itself. The weaving mechanism (superimposition) for filter composition with the application locates the point in the application where the filters will be added. Depending on the filter type, the associated action to perform varies: the addition (via the filter Substitution, Dispatch or Meta) and withdrawal (via the filter Error) of functionalities corresponding to *interface evolution* and the modification of existing functionality behavior (via any combination of filters of all kinds) corresponding to *behavior composition*.

**JAC:** Java Aspect Components [32] is a framework dedicated to the development of aspect oriented applications. It implements the API (Application Programming Interface) of AOP Alliance [3] and supports the composition of aspects statically as well as dynamically. Wrappers are used to change the behavior of the functionalities of an object and to make the object acts as if it accepts calls of new functionalities (not implemented in the object but in the wrapper itself). Then two kinds of adaptations can be performed in JAC: *interface evolution* and *behavior composition*.

**FAC/Julius:** Fractal Aspect Component [33] is a model that combines the possibilities of component platforms and aspect oriented frameworks. In the Julius implementation of the FAC model, a new kind of component (aspect component) and binding allows for the implementation of aspect oriented concepts: an aspect component can be connected to a Fractal component in order to modify the behavior of the latter. This acts as if an aspect has been woven on it. When an aspect component is bound to Fractal component, the former intercepts calls addressed to the latter in order to change its behavior. Then FAC/Julius supports *assembly modification* and *behavior composition*.

Table 1 sums up the kinds of adaptations supported by the studied approaches. Looking at the different approaches, we can notice that none of them implements the three kinds of adaptations we have identified. In the component approaches, the emphasis is on the capacity to modify component assemblies. While in aspect oriented approaches, the priority is to provide the ability to change the behavior of objects and the set of functionalities that they are able to handle. We can also point out the fact that adaptations are not implemented in the same way, even for a given kind of adaptations. In the next section, we will also see that the platforms do not handle adaptation errors in the same way.

| | **Interface evolution** | **Behavior composition** | **Assembly modification** |
|---|---|---|---|
| JAC | Functionality addition | Wrapper chaining | |
| CF/ Compose* | Functionality addition/ withdraw | Filter Superimposition | |
| Noah | | Interaction rule merging | |
| CCM/ OpenCCM | | | Binding addition/withdraw |
| Fractal/Julia | | | Binding addition/withdraw |
| Sofa/DCUP | | | Component replacement (implantation level) |
| OSGi | | | Binding addition/withdraw |
| FAC/Julius | | Special bindings | Binding addition/withdraw |

Table 1: Kinds of adaptations supported by the studied approaches

## Erroneous adaptations

This section lists some errors occurring during runtime adaptations. This helps to classify the safety properties we want to guarantee. We illustrate the reading with an application for the management of the diary of a project team, diaries of each team members and the diary of the project manager. Diary components implement a same interface including *add*, *remove* and *retrieve* meeting functionalities.

### Assembly inconsistencies

*Problem statement:* To communicate with each other, components are connected using typed bindings. Communication is unidirectional: the sender's side of the binding is a required interface and the receiver's side of the binding is a provided interface. Then the operations of a required interface must conform to operations of the provided interface it is bound to. Adamek et al. [2] has pointed out the occurrence of errors related to unbound interfaces or wrong bindings. One problem occurs when leaving some of a component's required interfaces unbound for reuse purpose (only part of the component's functionality is reused). If a required interface is unbound, the component should call no methods on it. If a component calls an operation on a required interface that is not tied to a provided one (the sender is not connected to a receiver), then the call cannot be achieved and an error occurs. Another problem arises when a component calls an operation on a required interface that is tied to a "wrong" provided one. In this case, the call cannot be handled (if the receiver has no operation that conforms the called one) or it results in an unexpected behavior (if the receiver cannot be used in the way the sender expects it).

*Example:* Consider a diary component that is bound to a database and to a printer to perform persistence and printing operations as shown in Figure 1. Suppose that during an assembly modification adaptation, the database component is removed and the print component is replaced by another one. In such a case, if we call persistence operations from the diary, il will result in a unbound interface error because the call cannot be handled. In the same way, if the component that replaces the print component does not offer printing operations (or with different parameters), there will be a wrong binding error.

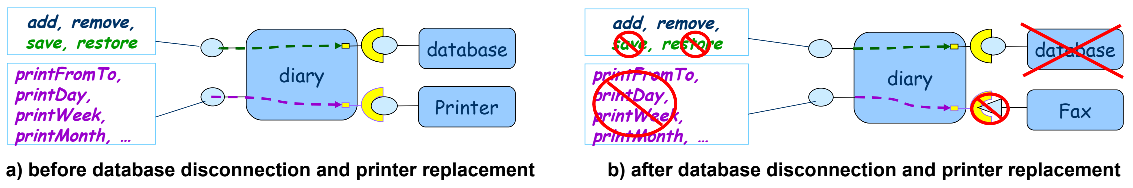

Figure 1: Assembly inconsistency in the diary example

*Assembly inconsistency handling in middleware platforms:* Wrong bindings are more or less handled in function of the platform. In the OpenCCM implementation [27], component replacement can be done as far as it respects connection type (provided and required interfaces must have the same type). In Fractal/Julia [11], component replacement is allowed only if the new component is a subtype of the replaced one. In these platforms, type conformity is only syntactical. Then, neither OpenCCM, nor Fractal/Julia, check that the behavior of the new component conforms the behavior of the one replaced. In contrast, SOFA/DCUP [34] type conformity is also done at a semantical level. When replacing a Sofa component, the system checks that the new one can be used in the same manner as the one replaced thanks to behavioral protocols. To prevent unbound interfaces errors, Fractal/Julia forbids to keep mandatory interfaces unbound. Then only unbound optional interfaces may generate errors. OpenCCM does not manage unbound interfaces errors. In OSGi, unbound interfaces errors cannot occur since an unbound component has to wait until another component providing the required service appears.

### Message-not-understood error

*Problem statement:* A method call is unknown when there is no applicable method in the receiver. This kind of errors is detected statically by compilers using type checking [12]. Dynamic operation calls such as permitted with Java reflect API or CORBA dynamic invocation services [38] are not checked by compilers. In such a case, dynamic method calls that are addressed to a wrong receiver trigger a message-not-understood error [13]. This error is only detected when the call is received by the receiver which can be too late to manage the error. A second problem related to dynamic calls is how to manage the modifications of the set of functionalities supported by a software entity. In this particular case of adaptation, new functionalities supported by a software entity can appear or disappear making difficult to locate wrong calls. If we consider that a type corresponds to the set of functionalities offered by a software entity, then we can compare the dynamic addition and withdraw of functionality handling to a change of the type during its life cycle.

*Example:* A static call to the *updateMeeting* operation on a diary component, with a *String* parameter while the diary waits a *Date*, is detected by the compiler (Figure 2 lines 1 to 5). A dynamic call to the operation with the same parameter using reflexivity is not detected by the compiler (line 7). At runtime, the reception of the call by the diary will throw an exception (lines 8 and 9).

```
1  C:\>javac StaticCallDiary.java
2  StaticCallDiary.java:11: cannot resolve symbol
3  symbol  : method updateMeeting (java.lang.String)?
4  location: class StaticCallDiary
5          d. updateMeeting("");
6
7  C:\>javac DynamicCallDiary.java
8  C:\>java DynamicCallDiary
9  java.lang.IllegalArgumentException: argument type mismatch
```

Figure 2: Message not understood error in the diary example

*Message-not-understood error handling in middleware platforms:* In Noah [8], software entities participating in an interaction rule are defined by type names but no type checking is carried out. Hence, if a functionality required in an interaction rule is missing, no warning is given until the functionality is called because such calls are dynamically interpreted. The same holds for JAC [32] and Compose* [16]. JAC and Compose* also support the modification of the set of functionalities accepted by a software entity. But these modifications are not handled at the type level because types are static in JAC and Compose*. Each call is intercepted before its reception by the receiver. The interceptor maintains a list of functionalities added at runtime and the delegees that implement each of them. If the call is related to one of these functionalities, the call is forwarded to the corresponding delegee otherwise the call is forwarded to the original receiver. In such a case, a message-not-understood error may still occur since the call can correspond to a functionality unknown to the receiver nor to the delegee.

### Adaptation composition conflicts

*Problem statement:* As stated by Hanneman [18], aspects composition, in particular when using same pointcuts (i.e. : aspects that will be woven in the same points of an application), may lead to undesirable interactions. A composition conflict is a situation where a combination of adaptations reduces or alters the functionality of themselves or of the system or engenders contradictory or non deterministic execution. The conflicts may occur only in one specific weaving order. The conflicts also tend to happen with aspects that engender side-effects on the adapted application.

*Example:* Suppose that two persons want to monitor the wrong accesses to the team's diary component by raising an exception. Both of them define an aspect to be woven on the *addMeeting* and *removeMeeting* operations of the component. Each aspect reads an environment variable that tells whether the user of the diary is authentified or not. As the two persons don't know that each other weaves a similar aspect, they probably use a distinct exception type in their aspect definition. In such a case, identical calls to *addMeeting* and *removeMeeting* operations of the team's diary component will throw two different exceptions after the weaving of the two aspects. This may lead to conflicts as in most of runtime environments only one exception can be thrown at one time.

*Behavioral composition conflicts handling in middleware platforms:* In Noah [8], since the merging operation is commutative, applying a set of rules to a component always results in equivalent behavior and does not depend on a particular order. In JAC [32], FAC/Julius [33] and Compose* [16], the resulting behavior depends on the composition order. SECRET (Semantic Reasoning Tool) can be used on top of Compose* to perform an impact analysis of each filter on resources (conditions, global variables, message deadline and so on). Then SECRET focuses only on the detection of errors related to the superimposition of a filter with an application not on the composition conflicts between several filters. To prevent composition conflicts between aspects, JAC proposes to specify use-defined policies that express aspect dependencies and incompatibilities. Then it is powerful to detect local errors between a few number of aspects that we have anticipated but is useless for indirect errors with a great amount of aspects that have been woven at different times.

## Synchronization

*Problem statement:* Synchronization problems come along with concurrent programming [4] and can be detected using model-checking techniques [22]. Simultaneous calls on components that share a component can generate ambiguity in the shared component behavior. In the situation where the shared component does not support concurrent calls, a synchronization error occurs.

*Example:* Suppose that the diary of a project team notifies incoming events to the diaries of each team member and that the team members' diaries use a buffer to display some information. Since we cannot make any hypothesis on the execution order of the personal diaries notifications, the buffer may be used incorrectly.

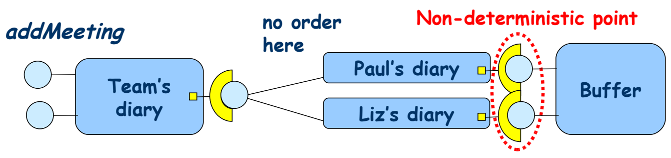

Figure 3: Synchronization error in the diary example

*Synchronization error handling in middleware platforms:* Sofa [34] detects only the particular synchronization situations where a deadlock occurs. The other approaches do not handle this error.

## Divergence

*Problem statement:* Divergence is a liveness property of model checking that occurs when computation/communication never stops [22]. Some loops are needed to implement recursive behavior, other ones are due to multiple adaptations for which the global network interaction has not been considered. The use of unbounded pointcuts (wildcards for example) may also lead to circular dependencies between classes [18].

*Example:* Suppose that a team is composed for a new project. Each team member has his/her own diary that he/she can parametrize dynamically for its own need. Joe and Liz work on a same activity and need to communicate quickly. Liz and Paul work on another common activity and Paul and Joe also share an activity. As Joe is really busy, he decides to notify Liz each time he updates his diary so that they can organize quick meetings when they are both free. As the team-work method works well, Liz decides to do the same with Paul because she knows he works part time, and Paul with Joe. In such a case, a divergence occurs on the *addMeeting* functionality. The circular notifications between the three persons will never end.

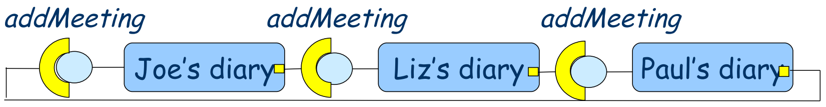

Figure 4: Divergence in the diary example

*Divergence error handling in middleware platforms:* Divergence errors occur in each of the studied approaches except Sofa [34] in which the use of behavior protocols makes it possible to detect quickly divergence.

**Synthesis**

To ensure adaptation safety, the studied approaches perform some checking. For component platforms, in which assembly modifications are predominant and made explicit, checking focuses on assembly consistency. For aspect-oriented frameworks such as JAC, Noah and Compose*, in which composition of behavior is the explicit way to adapt applications, checking emphasizes on behavioral composition coherence. We can notice that the weaknesses in terms of verification of component platforms are the strengths of aspect oriented frameworks and vice versa. However, none of the studied approach except Sofa handles errors related to the global interaction graph: divergence and synchronization.

Even if component platforms do not provide explicit constructs to compose behavior, modifying component assemblies also impacts the way the application behaves. And even if aspect-oriented frameworks do not provide explicit constructs to modify the software entity interaction graph, modifying the behavior of these entity often engender new interactions thus new implicit assemblies. Then, each platform is concerned by almost all kind of adaptation errors.

There is no approach that makes all necessary checking. The checking offered by one platform cannot be reused in other platforms as adaptation implementations are too different. By comparing the platforms, we conclude that existing solutions cannot be reused "as-is" everywhere to determine the safety of the three kinds of adaptations (type evolution, behavioral composition, assembly modification). In addition, most of the platforms detect errors only after the adaptation has occurred. Then we loose a degree in the application safety as it is not always possible to revert to previous application states.

Our goal is to handle the adaptation-related errors independently of the technological details of the platforms. We also want to check the adaptation safeness a priori, that is just before the adaptation is performed in order to prevent errors instead of having to recover them.

## 3 ARCHITECTURE OF THE SAFETY SERVICE

The safety service that we developed, called Satin, can be queried by technological platforms to determine whether a runtime adaptation is safe or not. The safety service makes it possible to validate an adaptation of an application $A$ according to the operations offered by the components and to the previous adaptations of $A$. The service prevents the execution of adaptations that would lead the application to an unsafe state.

In this section, we present the architecture of the service. The architecture has been designed as a platform-independent model that describes the core of the service. The entry points to component and aspect platforms are modeled as plugins. The service protocol formalizes the way the service can be used and how the service communicates with the platforms.

### Core of the service

Figure 5 depicts the core model of the service: **adaptation patterns** reify the concept of adaptation and **roles** reify component typing, taking into account possible evolution of types by adaptation. Adaptations affect the **ports** of a component, which are abstractions of operations provided or required by the component.

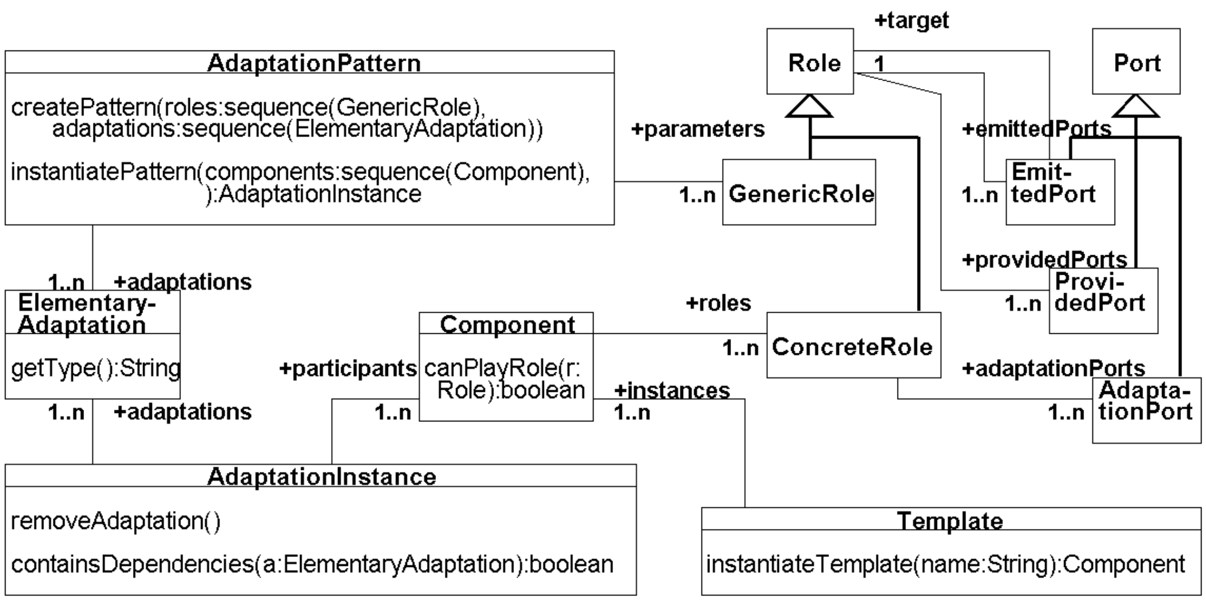

Figure 5: Core model overview

#### Adaptation patterns

An *adaptation pattern* describes the structural and behavioral modifications involved in a dynamic adaptation. It consists of a set of elementary adaptations, 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.

An adaptation pattern represents the unit of application and reuse of elementary adaptations. For example, figure 6 depicts an adaptation pattern (written in an ISL-like style for easy reading) that contains an elementary adaptation modifying the behavior associated with an *addMeeting* port in a diary application. It 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 *super* in Java or *proceed* in AspectJ [19]) that invokes the prior, non-adapted version of *addMeeting*.

```
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
}
```

Figure 6: *Synchronization* adaptation pattern

## Roles

An adaptation pattern is defined on a set of roles. An **adaptation pattern role** specifies the ports that a component must provide or require 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*.

Adaptation patten roles are complemented by component roles. A **component role** describes the ports offered by the component and the ports required by the component. Note that, unlike in conventional type systems, the role of a component can evolve by adaptation. Each component role can evolve independently of the roles of other components.

## Safety properties

The third keystone of the model is the criteria of safe adaptations. We suppose applications to be fault-free (each individual component are safe and the initial assemblies of these components are safe too). If the application initial state is safe, we guarantee that the application state remains safe after being adapted. For that, we have identified a set of *safety properties*, which correspond to the criteria to be satisfied in order to adapt safely components. Each safety property is guaranteed by a set of OCL [39] constraints (operation preconditions) which must be true for the adaptation to occur.

The "assembly inconsistencies" errors described in section 2 are handled by two safety properties: 1) **use context conservation** prevents interchanging two software entities whose roles are not substitutable according to the rules of the target platform, and 2) **binding consistency** guarantees that for each operation required by a component a conforming operation is effectively offered by another component. The "message not understood" errors are handled by two safety properties: 1) **guaranteed message consumption** prevents calls to unknown operations, and 2) **visibility guarantee** prevents calls or adaptation of operations that are not visible to the adaptation process. The "adaptation composition conflicts" and "synchronization" errors are handled by the following safety property: **behavioral determinism** identifies potentially unpredictable behaviors. The "divergence" errors are handled by the following safety property: **cycle detection** identifies loops in the operation call flow.

Note that the above list of safety criteria is full of terms that have very different definitions in different platforms (substitutability, conformance, visibility, . . . ). Thus the above criteria have different interpretations depending on the target platform. For this reason, the core of the service that is platform-independent need to rely on plugins that help to take into account platform specificities and to locate them.

## Plugins

The main advantage in the definition of Satin is to capture the overall interactions that a safety service for run-time adaptation must support, in spite of the variation of the central notions of "typing" and "adaptation" in the different platforms.

### Type checking plugin

Before accepting the application of an adaptation pattern to a set of components, the **binding consistency** safety property has to check whether each component conforms to the role it has to play in the adaptation. This check must take into account for various definitions of "conformance" adopted in different platforms. According to the classification of Beugnard et al. [7] conformance can be based on syntactic, behavioral, synchronization or even quality of service criteria. Moreover, for each "conformance model", a platform can choose a particular concretization. For example, syntactic conformance is interpreted differently according to the platforms: no subtyping, inheritance conformance or inclusion polymorphism (with different variance rules). Behavioral conformance can be based on pre and postconditions. The challenge for a general safety service is that there is no universal model for conformance and platforms typically implement only one of many conformance models cited above.

The *isSubRoleOf* operation of the *Role* class of the core model refer to the conformance relationship of the implementation platform through the *conforms* operation of the type checking plugin, which is responsible for checking that types are compatible according to the rules laid down at concretization time. Depending on the concretization, any of the conformance definitions discussed above could be used.

**Adaptation introspection plugin**

The types of elementary adaptations differ from a platform to another. The ways in which the elementary adaptations are expressed and implemented also vary according to the platforms. These variations are captured in the adaptation introspection plugin. Thus, adaptation patterns handle adaptations without knowing their representation.

Applying an adaptation pattern to a set of components can be done only if the elementary adaptations to be applied to components are compatibles (*ElementaryAdaptation.isCompatibleWith* operation of the adaptation introspection plugin) with the adaptations already applied to those components in the same pointcut (*PointCut.match* operation of the adaptation introspection plugin).

**Data extraction plugin**

To populate the Satin model with information about the application to be monitored, some data need to be extracted. The first time a component is involved in an adaptation, a Satin component with its name is created. Then, the initial roles of the component need to be deduced from application types. The ways in which the application can be introspected depends of the platform and language. These variations are captured in the data extraction plugin.

Section 4 explains how to configure the plugins for a given platform.

## Service protocol

Figure 7 presents the overall service architecture with the diary example and FAC/ Julius [33] as the service client. Message exchanges between the service and the platform are bi-directional. It means not only that the platform queries the server to check an adaptation, but also that the server interact with the platform. The entry points for message exchanges are formalized by four interfaces. The *Server* interface allows the platform to use the service without knowing its internal mechanisms thoroughly. Each operation of the *Server* interface is associated with a step of the validation process of adaptations described above ((un)registration of components, creation/destruction of adaptation patterns, (un)application of adaptation patterns, the replacement components). The *Extraction*, *Contract* and *Adaptation* interfaces correspond to the service plugins and are used by the server to communicate with the platform. The *Extraction* interface is used by the server to retrieve information on the application in order to create roles from the component types of the application.

The *Contract* interface is used by the server in order to determine if two roles conform according to the typing rules of the platform. The *Adaptation* interface is used by the server for adaptation introspection.

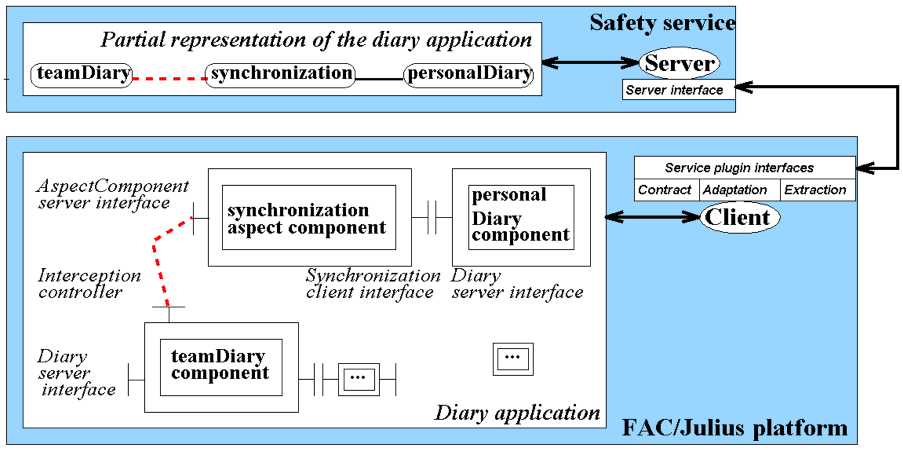

Figure 7: Safety service architecture : component adaptation and service query

The safety service can be used according to the following adaptation process.

Step 1. Components can be registered to the server. This step is important to have an partial representation of the application to be adapted. However, the step can be delayed to the first time a component is being adapted (see step 3). Hence, only the components of the application that are adapted need to be represented in the service state. At this step, the server queries the platform to retrieve data on the components through the data extraction plugin.

Step 2. The description of the adaptation to be performed is registered to the safety server. The modifications involved by the adaptation description are checked in order to assert if the safety properties are preserved. At this step, the server queries the platform to get adaptation-relative information through the adaptation introspection plugin.

Step 3. The client asks if a given list of components can be adapted using a description of adaptation registered previously. Components to be adapted have to be registered to the server if not already done. At this step, the server queries the platform in order to check component types conformity according to the platform typing rules through the type checking plugin.

Step 4. As some adaptations may not be wished anymore in the future, the modifications applied on the components may have to be undone. At this step, the server checks if the modifications involved by an adaptation can be undone regarding previously adapted components.

Steps 3 and 4 modify the state of the service to memorize the adaptation of components. Then, the platform and the service must be synchronized so that the state of the application from the point of view of the adaptations is equivalent in the service and in the platform.

# 4 SERVICE IMPLEMENTATION AND PLATFORM INTEGRATION

A prototype of the service has been realized and has been tested with Noah [8] and the Julia implementation framework of the Fractal model [11]. As we have chosen two targets that support different adaptation capabilities (see section 2), the flexibility of the service and its safety properties could be evaluated.

This section describes the implementation choices of the service and how to configure the service prototype for Noah and Julia.

## Concretization of the service

To concretize the service, we must concretize the core model and the platform/service exchanges. The core model structure is implemented as Java classes. The OCL constraints are injected in the Java code using the Dresden-OCL toolkit [14]. Message exchange to use the service are described by an IDL Corba that defines the *Server*, *Extraction*, *Contract* and *Adaptation* interfaces. Note that the adaptation process described in section 3 corresponds exactly to the application protocol described by the *Server* interface.

```
interface Server  {
  void registerComponent(in string componentName, in sequence<string> interfaceNames,
                         in sequence<string> implantationClassName)
       raises (AlreadyDefinedException, SafetyPropertyViolationException);

  void unregisterComponent(in string componentName)
       raises (LifeCycleException, NotDefinedException);

  string createPattern(in string adaptationDescription)
       raises (AlreadyDefinedException, SafetyPropertyViolationException);

  void removeAdaptationPattern(in string patternName)
       raises (LifeCycleException, NotDefinedException);

  string instantiatePattern(in string patternName, in sequence<string> participantNames)
       raises (AlreadyDefinedException, NotDefinedException, SafetyPropertyViolationException);

  void removeAdaptationInstance(in string instanceName)
       raises (NotDefinedException, SafetyPropertyViolationException);

  void replaceComponent(in string oldComponent, in string newComponent)
       raises (AlreadyDefinedException, SafetyPropertyViolationException);
};

interface Extraction {
    Classifier System.getClassifier(in string name) raises (ReflectionException);
    sequence<Feature> getFeatures(in Classifier c) raises (ReflectionException);
    string getName(in Feature f) raises (ReflectionException);
    sequence<Classifier> getArguments(in Feature f) raises (ReflectionException);
    Classifier getReturn(in Feature f) raises (ReflectionException);
};

interface Contract {
  boolean conforms(in Classifier c1, in Classifier c2);
};

interface Adaptation {
    boolean match(in String s1, in String s2);
    boolean isCompatibleWith(in ElementaryAdaptation ea1, in ElementaryAdaptation ea2);
};
```

### Configuration of the plugins

To configure the service so that it can be used with a specific platform, it is necessary to implement the *Extraction*, *Contract* and *Adaptation* interfaces. Tables 2, 3 and 4 give correspondences for each plugin operations for Noah and Julia. The service calls the different operations of the IDL CORBA interfaces on implementation objects. The objects used depend on a property file that specifies if the service should use the interface implementations of Noah or those of Julia for example.

As Julia and Noah are Java platforms, the Java reflect API is used to introspect the hosted application in both cases.

| Extraction plugin operations | Julia & Noah |
|---|---|
| System.getClassifier(String name) | java.lang.Class.forName(String) |
| Classifier.getFeatures() | java.lang.Class.getDeclaredMethods() |
| Feature.getName() | java.lang.reflect.Method.getName() |
| Feature.getArguments() | java.lang.reflect.Method.getParameterTypes() |
| Feature.getReturn() | java.lang.reflect.Method.getReturnType() |

Table 2: Configuration of the data extraction plugin for Julia and Noah

The two platforms are based on syntactically conformance. Although in the case of Julia, we have to check that the component belongs to a hierarchy and in Noah we only use Java class substituability principle. With these two examples, we see that the implementations of type checking differ for the two platforms. The implementation of the *Contract* interface can be delegated to the *org.objectweb.fractal.api.Type* class for Julia and to Java.lang.Class class for Noah

| Contract plugin operations | Julia | Noah |
|---|---|---|
| conforms(Classifier, Classifier) | Type.isFcSubTypeOf(Type) | Class.isAssignableFrom(Class) |

Table 3: Configuration of the type checking plugin for Julia and Noah

The two platforms are based on different types of adaptations: Noah is close to an aspect oriented platform and Julia is a component platform. In Noah, elementary adaptations are described explicitly by interactions rules and consist in modifying the behavior of functionalities. These rules are reified and can be introspected to find the needed information on compatibility. In Julia, elementary adaptations are programmed using an API that enables modifying component assemblies (bindFc, unBindFc, addFcSubComponent, removeFcSubComponent). Content controllers check the consistancy of the components' bindings and can be queried to access this information. Here again, we see that the implementations of adaptation introspection differ for the two platforms.

| Adaptation plugin operations | Julia | Noah |
|---|---|---|
| match(String, String) | ContentController.checkFC | String.substring(String) |
| isCompatibleWith(in ElementaryAdaptation ea1, in ElementaryAdaptation ea2) | BindingController.lookupFc() | Rule.merge(Rule) |

Table 4: Configuration of the adaptation introspection plugin for Julia and Noah

## 5 CONCLUSION

With the design of the safety service for software adaptations, we have come to appreciate the benefits of MDE [37]. We have been able to formalize the adaptation checking at the model using OCL [39]. The minimization of software defects is an important issue in application areas such as safety, security, cost and legally, where the residual defect ratio is known to be particularly high. Hence, it would make no sense to consider the problematic of adaptation safety without a formal foundation. The abstraction level of the service design has allowed us to verify the service correctness [30].

The use of the service implies for each platform to give plug-ins implementation. This plug-in solution has the advantage to locate target code for the concretization step of the MDE process. The core of the service can be extended in adding new OCL constraints and new plug-ins at the service level. These extensions would permit to manage new kind of adaptations and new safety properties. We have experimented this aspect for mobile platforms [29]. For each new adaptation such as communication mode switching. Extending the service means to identify new safety properties and their OCL formalization, to validate them and to express new dependencies towards platform related information through new plug-ins.

## REFERENCES

[1] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of USE*, University of Warsaw, Poland, 2003.

[2] Jirí Adámek and Frantisek Plasil. Partial bindings of components - any harm? In *11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 632–639. IEEE Computer Society, 2004.

[3] AOP Alliance. API AOP Alliance. http://aopalliance.sourceforge.net/, 2005.

[4] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[5] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modle MICADO.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 2001.

[6] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. Technical report, TRESE project, University of Twente, Centre for Telematics and Information Technology, Enschede, The Netherlands, 2001.

[7] A. Beugnard, J.-M., N. Plouzeau, and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, 1999.

[8] M. Blay-Fornarino, A. Charfi, D. Emsellem, A.-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 10(10), 2004.

[9] R. S. Boyer and J. S. Moore. Proof-checking: Theorem-proving and program verification. In W. W. Bledsoe and D. W. Loveland, editors, *Contemporary Mathematics: Automated Theorem Proving - After 25 Years*, pages 119–132. American Mathematical Society, Providence, RI, 1984.

[10] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[11] E. Bruneton, T. Coupaye, and J.-B. Stefani. The fractal component model. http://fractal.objectweb.org/, 2004.

[12] L. Cardelli. Type systems. *ACM Computing Surveys*, pages 263–264, 1996.

[13] P. Costanza. Dynamic vs. static typing - a pattern-based analysis.

[14] Dresden university. OCL2 toolkit. Internet: http://dresden-ocl.sourceforge.net/.

[15] A. Flissi and P. Merle. Une démarche dirigée par les modèles pour construire les machines de déploiement des intergiciels à composants. *Conférence Langages et Modèles à Objets (LM0 2005)*, 11, 2005.

[16] C. Francisco N. Garcia. Compose*: A runtime for the .Net platform. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, August 2003.

[17] The Object Managemant Group. CORBA Component Model Specification, 4.0 edition. OMG Document formal/2006-04-01, 2006.

[18] J. Hanneman, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software (aaos) workshop report. Technical report, University of California, Darmstadt, Germany, 2003.

[19] G. Kiczales and J. Lamping. Aspectj home page. http://eclipse.org/aspectj, 2001.

[20] P. Koopmans. On the design and implementation of the sina/st language. Master's thesis, Dept. of Computer Science, University of Twente, August 1995.

[21] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[22] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symp. Principles of Programming Languages (POPL'85)*, pages 97–107, New Orleans, LA, USA, 1985.

[23] R. Marvie, P. Merle, J.-M. Geib, and S. Leblanc. Torba: Trading proxies for corba. In *Proceedings of the Sixth USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, 2001.

[24] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 1997.

[25] Microsoft. Microsoft .NET platform. http://www.microsoft.com/net/, February 2001.

[26] Y. V. Natis. Service-oriented architecture scenario. Gartner, Inc, 2003.

[27] Objectweb Consortium. Open CCM. http://openccm.objectweb.org/.

[28] Objectweb Consortium. JOnAS : Java (TM) Open Application Server. http://jonas.objectweb.org/, 2005.

[29] A. Occello and A.-M. Pinna-Déry. Sûreté de fonctionnement d'applications nomades construites par assemblage de composants. In *2nd French-speaking conference on Mobility and Ubiquity Computing*, pages 73–80. ACM Press, 2005.

[30] A. Occello, A.-M. Pinna-Déry, and M. Riveill. 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: Integrating Verification and Validation in MDE*, Lillehammer, Norway, 2008. IEEE Digital Library.

[31] Open Services Gateway initiative. OSGi service platform (3d release). http://www.osgi.org/, 2003.

[32] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, volume 2192 of *LNCS*, pages 1–24. Springer-Verlag, 2001.

[33] N. Pessemier, L. Seinturier, and L. Duchien. Components, adl and aop: Towards a common approach. In *In Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04)*, 2004.

[34] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998.

[35] E. Roman, S. W. Ambler, and T. Jewell. *Mastering Enterprise Java Beans II and the Java 2 Platform.* John-Wiley & Sons Inc., enterprise edition, 2002.

[36] R. Rouvoy and P. Merle. Towards a model driven approach to build component-based adaptable middleware. In *Proceedings of the 3rd International Middleware Workshop on Reflective and Adaptive Middleware (RAM'04)*, Toronto, Ontario, Canada, 2004.

[37] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–32, 2006.

[38] The Object Managemant Group. Common Object Request Broker Architecture: Core specification. OMG TC Document formal/04-03-01, 2004.

[39] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 1999.

[40] W. Witthawaskul and R. Johnson. Specifying persistence in platform independent models. In *Proceedings of the 2nd Workshop in Software Model Engineering at the Sixth International Conference on the Unified Modeling Language, UML 2003*, San Francisco, California, USA, 2003.

## ABOUT THE AUTHORS

**Audrey Occello** is a lecturer and research assistant in Computer Science at the Engineering School of Technology of the University of Nice, France. She can be reached at occello@polytech.unice.fr. See also `http://www.polytech.unice.fr/~occello/`.

**Anne-Marie Dery-Pinna** is an Assistant Professor in Computer Science at the Engineering School of Technology of the University of Nice, France. She can be reached at pinna@polytech.unice.fr. See also `http://www.polytech.unice.fr/~pinna/`.

**Michel Riveill** is a Professor in Computer Science at the Engineering School of Technology of the University of Nice, France. He can be reached at riveill@polytech.unice.fr. See also `http://www.polytech.unice.fr/~riveill/`.