LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

SAFETY OF COMPONENT ADAPTATIONS: ELEMENTS OF FORMALIZATION

Audrey Occello, Anne Marie Dery-Pinna

Projet RAINBOW

Rapport de recherche
ISRN I3S/RR–2004-04–FR

Janvier 2004

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

MOTS CLÉS :

ABSTRACT:

Development by assembly has supplanted development by programming, thereby making applications more reusable and flexible. However, systems need to be evermore dynamically adaptable to meet the demands of new kinds of application domains. A number of component models and component platforms based on those models are emerging to support adaptability. Although runtime adaptations can lead the application to an unsafe state, this problem has received little attention of needed. We will therefore propose a methodology for any existing component platform to benefit from our results, reuse our concepts and make dynamic adaptations safer.

KEY WORDS :

safety, runtime adaptations, components, metamodeling, validation

Safety of Component Adaptations: Elements of Formalization

Audrey Occello[1] and Anne Marie Dery-Pinna[1]

[1] Laboratoire I3S - Bâtiment ESSI
650, Route des Colles, B.P. 145
06903 SOPHIA-ANTIPOLIS Cedex, France
{occello, pinna}@essi.fr
http://rainbow.essi.fr

Abstract. Development by assembly has supplanted development by programming, thereby making applications more reusable and flexible. However, systems need to be evermore dynamically adaptable to meet the demands of new kinds of application domains. A number of component models and component platforms based on those models are emerging to support adaptability. Although runtime adaptations can lead the application to an unsafe state, this problem has received little attention of needed. We will therefore propose a methodology for any existing component platform to benefit from our results, reuse our concepts and make dynamic adaptations safer.

Key words: safety, runtime adaptations, components, metamodeling, validation by simulation.

1 Introduction

To improve software productivity and quality, and to reduce complexity, skill requirements and development costs, development by assembly has supplanted development by programming. A solution to support rapid software evolution is to construct software systems from reusable components. Through this approach, the architecture of a system is described as component assemblies along with the interactions among these components. However, application evolution is often unforeseen. Hence systems also need to be evermore dynamically adaptable to meet the demands of new kinds of application domains such as mobility whose execution need to take into account runtime application aspects (variation in the use context of an application, platform connectivity, available resources, user localization, and so on).

A number of component models and component platforms based on those models are emerging to support adaptability. By comparing some component models [37, 24, 5, 31, 28, 32, 36], we can see that they do not provide the same kinds of adaptation (add/remove functionality, change behavior of a functionality, alter component assembly, …). Moreover, each kind of adaptation is implemented differently in component models. For example, Noah [32] bases its assembly adaptation on interaction

pattern applications, SOFA [31] and Fractal [5] on containment compositions, CCM [24] on binding modifications, and ACEEL [36] on automaton compositions. One problem with these approaches is that each adaptation process implies modifying components. Thus, runtime adaptations often lead the application to an unsafe state. However, the problem of determining the safety of component dynamic adaptations has received, in our opinion, little attention.

With the increasing use of software components and the multiplication of component platforms, the methods for safe adaptation of components will become of greater importance in the future. Facing this problem, we propose to ensure adaptation safety by abstracting away properties. Rather than designing a safety tool for one component platform, we will propose a methodology for any existing component platform to benefit from our results, reuse our concepts and make dynamic adaptations safer. So as to propose a general methodology for adaptation safety, we have chosen to work at a meta level: we define a metamodel for adaptable components on which we describe and prove safety properties independently from component platforms.

Section 2 presents how safety of dynamic adaptation is handled in existing component models. Section 3 describes the basis of our proposal for safe component adaptations: our metamodel highlights the key elements involved in the process of component adaptation independently from specific component models and safety properties have been identified and expressed through our metamodel. Section 4 shows how to validate our approach. First, we will see how the safety properties can be proven once and for all via the metamodel. Secondly, we will see how the safety properties proofs can be projected in specific component models. The last section concludes and presents future work.

2 Safety of Dynamic Adaptation in Component Models

Existing component models, which allow dynamic adaptations, highlight various kinds of adaptations. As the set of adaptation cases is too large, our proposal focuses on the more frequently encountered component adaptations: adding or removing a functionality to a component, composing new actions with the previous ones associated to a component functionality, and modifying a component assembly.

Component models allowing dynamic adaptation are not limited to those presented in this section. We have selected component models that we believe to be representative of the kinds of adaptations we are interested in and which lack adaptation safety. To conclude the section, we will see what can be made safer according to this state of the art analysis.

CCM. The CCM model [24] defines a component framework to design, produce, deploy, and run distributed heterogeneous component based applications. It provides new solutions to exhibit component interconnections, to separate functional and non-functional aspects, and to deploy components. According to the CCM specification, it is dynamically possible to load components into generic container servers, to create instances and then to interconnect these instances by means of an assembly. This assembly information is defined in the CCM to be static. Indeed, the Component As-

sembly Descriptor defines which components to use and how to interconnect them. However, it is possible to change connections during the life time of a component instance as far as it respects connection conformity[1]. Although we can replace a component by another in an assembly, the component model does not provide means to check that the behavior of the new component conforms the behavior of the one replaced.

Fractal. The Fractal model [22, 5] is a dynamic and recursive component model which allows structural composition based on containment and binding relationships between components. In this paper, we will speak about the Julia framework [5], which is the Java standard implementation of Fractal. Julia allows two kinds of adaptation: assembly modification and functionality composition. The first kind of adaptation refers to the fact that composite components can change their structure over time. The second one refers to the fact that controllers can change the behavior of components. Typing rules on binding ensure that assembly modifications are valid but as for the CCM, Fractal does not check that the behavior of the new component conforms the behavior of the one replaced. A life cycle service is used to manage synchronization when reconfiguring several components: components are stopped when adapted and are restarted only when the needed modifications have been carried out. This is time consuming. Moreover, the set of components that has to be stopped to make the adaptation is not computed automatically. The programmer has to stop the components that are supposed to be concerned by the needed modifications manually. This is not sufficient to guarantee that all the modifications have been carried out properly. On the other hand, the modifications implied by a given adaptation are not explicitly described but are carried out through the programming: adaptation can only be validated after applying the modifications. Consequently, components may be restarted introducing errors.

SOFA. SOFA [31] is a component model which allows an application to be composed of a set of dynamically hierarchical updatable components. The strength of the SOFA model is in the use of behavioral protocols [2]. Behavioral protocols describe how components can be used: the behavior of a SOFA component is the set of all traces[2], which can be produced by the component. SOFA allows one kind of adaptation: assembly modification. Though protocols, when replacing a component, we can check that the new one can be used in the same manner as the one replaced. Another characteristic of the model is that adaptation events are also included in behavior specification of components to indicate precise moments where any adaptation can occur. The begin event and the end event of potential adaptations are integrated into protocols traces so that they do not follow a request event on the component being adapted and so that there can be no other event between them. Since no other calls can be handled until the end of the adaptation, SOFA components do not need to be stopped, while being adapted, in contrast to ACEEL components (see description in

[1] For two component types to be connectable, it is required that a port provided (a facet) by one of them has to be compatible with a port used (a receptacle) by the other. The facet type must be the same or a subtype (interface inheritance) of the receptacle type

[2] A trace is formed of a sequence of tokens denoting events (requests or responses of method calls) occurring on a component.

the next paragraph) and Fractal components. Nevertheless, the programmer has in charge to define the "right points" of adaptations in protocols, and no tool is provided to check that the adaptation method calls are in the "right place". Moreover, no distinction can be made between different kinds of adaptation: the moments allowing an adaptation to occur are the same for all adaptations.

ACEEL. ACEEL [36] is a framework allowing components to be dynamically adapted to variations of mobile environments. ACEEL components reify their adaptation needs like an automaton whose states represent execution conditions and whose transitions represent a significant variation in the environment and determine the adaptation to be performed when the variation occurs. Two types of adaptation may be performed. The first one consists in changing the value of some elements of the internal state of the component. The second one consists in replacing the implementation of the component. ACEEL takes into account synchronization when reconfiguring several components and consistency among automata of components. However, the composition of behaviors remains difficult and error prone since the composition must be managed ad-hoc by the programmer at the level of the automata. On the other hand, as for the CCM model and Fractal, replacing an implementation by another one does not guarantee that the behavior described in the new component implementation will conform the behavior described in the one replaced. The ACEEL framework implements a mechanism to enforce waiting until the end of ongoing requests before adapting components. This is not adequate because a method may never end.

JAC. JAC [28] is a Java framework for AOP applications. Wrappers are used to intercept messages before reception by the component. Wrappers can be added, removed and rescheduled dynamically. Wrappers are used to perform two kinds of adaptation: adding functionalities to a component and modifying the behavior of a component. On one hand, adding a functionality does not extend the component type. The disadvantage is that calls to such added functionalities must be carried out explicitly using a specific API of the framework instead of applying the functionality directly to the receptor component. Even if the API can be specialized by the framework user to perform verifications, the process remains ad-hoc and is optional. On the other hand, an aspect compositor schedules actions associated with the aspects by defining a policy of dependence and incompatibility rules between aspects. The policy is then used to manage conflicts of control composition. But the validation of the control composition must be carried out by the programmer. Then, the validation of the control composition remains informal and is error prone.

Noah. Noah [32, 3] is a framework providing a dynamic adaptation layer over the EJB [37]. The framework allows the programmer to express the interactions between components declaratively and externally to components via the ISL language. By describing interaction rules (rewriting rules on functionalities) between components, the programmer can modify the behavior of components dynamically. Components participating in an interaction rule are defined by type names but no type checking is carried out. Hence, if a functionality that is required in an interaction rule is missing, no warning is given until the functionality is called. A merging operation based on ISL operators is used to compose a set of rules applied on a component [30]. Since the

merging operation is commutative, applying a set of rules applied to a component always results in equivalent behavior and does not depend on a particular order.

Synthesis. In conclusion, each component model checks the safety of some adaptations. But the validations performed are not systematically taken into account, and the techniques used to validate the modifications are often ad-hoc or are in charge of the programmer. By comparing such component models, we conclude that existing solutions cannot be used to determine the safety of the three kinds of adaptations:

1. Type evolution consistency: if we consider that the component type corresponds to the set of functionalities it offers, then we can compare our need to a change of the component type during its life cycle. However, work on typing as defined in Object Oriented applications [7, 10] are not sufficient to validate type evolution during execution.
2. Behavior composition coherence: associating new behavior to a functionality should not imply contradictory or non deterministic execution of the functionality. Mechanisms used for functionality composition in metamodeling [21] or in AOP [28] should be extended to take into account behavior composition coherence.
3. Assembly soundness: is the connectivity correct and complete? Is it possible to call an unknown functionality? Is a component missing in an assembly or of an inadequate type? ADLs [19] are insufficient to guarantee that dynamic changes will be applied to the executing system in a safety manner. They ensure assembly validity only for initial constructions.

In addition, adaptation-related modifications are often carried out programmatically as for the CCM [24], Fractal [5], SOFA [31] and ACEEL [36]. Hence, we can only detect errors after the modifications have been performed and we lose a degree in the application safety. On the other hand, no effort has been done to support safety when performing adaptation-related modifications.

Some component models try to determine when it is safe to adapt components. ACEEL [36] and Fractal [5] require components to be stopped during adaptation. SOFA [2] proposes to determine a priori the moments of adaptation. But those techniques are not appropriate: the first worsen performance of the system and is too drastic, the second is too constraining and not enough flexible.

3 The Basis of our Approach

Our proposal to making the adaptation process safer is to identify: 1) *what* are the criteria of safe adaptations, 2) *when* do adaptations have to be carried out and 3) *how* do the modifications associated to a given adaptation have to be performed. We believe that those questions can be answered by determining what we call "safety properties" (see section 3.2) that components and adaptations must fulfill. Our proposal is similar to approaches in formal specification and coherence validation at runtime by abstracting away properties [9] (for example, Briais [4] defines properties to be checked when objects move on network). Our approach is based on a metamodel for adaptable components (see section 3.1) using UML [27], which allows us to set the

vocabulary. In addition, we define OCL [38] constraints, which are checked on class instances (see section 3.2) in order to ensure the safety properties.

To illustrate our proposal, we will use in next sections, as running example, two component types. A *BasicDiary* component type will provide functionalities which correspond to adding, removing, and consulting meetings. A *DataBase* component type will provide functionalities to memorize and restore states and to synchronize its data.

3.1 A Metamodel for Adaptable Components

Whereas CodA [17] defines a metamodel to reify the process of communication between actors, we define a metamodel, based on the MOF [26], whose elements involve dynamic adaptations of components. Our approach follows the MDA recommendation [25], in the same way as [14, 23].

The first central element of our metamodel refers to a typing notion allowing specification evolutions. To avoid any confusion with classical typing approaches [7, 10], our typing notion will be referenced as "role". In addition to a typing notion, we will see in this section that our role definition also includes other role definitions. The second central element is the adaptation pattern, which can be understood as a dynamic reconfiguration protocol [29]. We believe that describing adaptations is better than programming because it checks for safety before performing the modifications associated with adaptations. Then, the adaptation pattern describes explicitly what an adaptation is: it allows to express the effect of an adaptation on the structure and on the behavior of components. An informal definition of each element of our metamodel is given below.

A **`role`** (see figure 1) has a name and describes component properties: not only syntactic definitions of ports but also behavioral features associated with these ports. There are two kinds of roles. A **`simple role`** has provided ports determined by a simple name. Simple roles may be seen as metaobjects controlling the behavior of the components they are associated with like Peschanski roles [29]. An example of a simple role can be the *BasicDiary* role and the *DataBase* role. **A `generic role`** has provided ports determined by regular expressions and express properties that the components participating in a collaboration must exhibit, but do not necessarily determine a specific “type” in particular as for UML interfaces[3] [27]. An example of a generic role can be the *AttributeManager* role, which offers generic getters and setters functionalities.

[3] UML 2.0 interfaces subsume the UML 1.x concepts of classifier role.

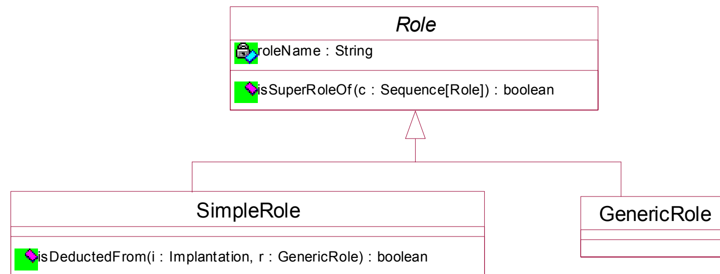

Fig. 1. UML representation of roles

Each **`port`** (see figure 2) is determined by a signature: a designation, a set of parameter roles and possibly a return role. We distinguish two types of ports. A port whose designation is a name captures and handles a particular functionality addressed to the component. A port whose designation is a regular expression corresponds to a set of functionalities.

For example, suppose we have the definition of a role *Meeting* and of a role *Date*. Then the *BasicDiary* simple role has three provided ports: addMeeting(Meeting), removeMeeting() and searchFor(Date): Meeting. The *DataBase* simple role has four provided ports: load(): BasicDiary, store(BasicDiary), lock(BasicDiary) and unlock(BasicDiary). And the AttributManager generic role has two provided ports: get*(Any) and set*(Any) where get* (resp. set*) is a regular expression denoting any name starting with *get* (resp. set) and representing getters (resp. setters) functionalities.

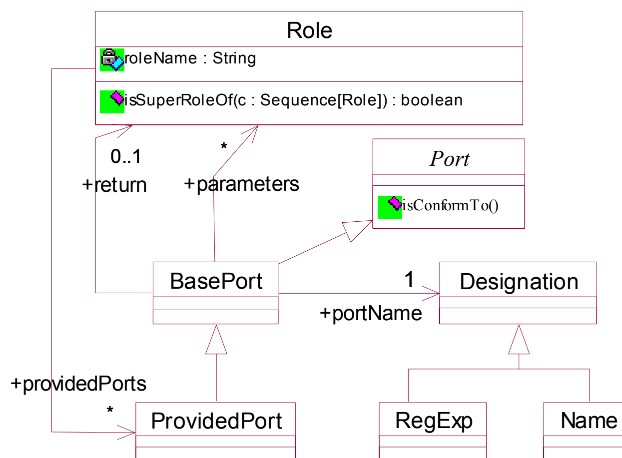

Fig. 2. UML representation of port

A **`template`** (see figure 3) defines a category of components. Template instantiates components. A template is composed of simple roles and an implementation.

An **implantation** (see figure 3) has only ports determined by a simple name and is the representation, at the metamodel level, of the component code (classes, descriptors, …) described in a specific component model. Details regarding the implementation are abstracted away, hiding internal activity. We only focus on the adaptability part of a component: implantation only holds ports that can be adapted and called within an adaptation.

A **component** (see figure 3) is an instance of a template, a unit of execution. A component has only simple roles, which can be seen as view points similar to work as RM-ODP [12] and CCM [8]. But in contrast with those work, we can use a component according to a subset of its roles by recovering the union of the desired roles simultaneously. At any time, the roles of a component reflect its interactions with the environment. The roles of a component can evolve in time by adaptation. All the components instantiated from a given template initially have the same associated roles but the roles of two components can evolve in time independently.

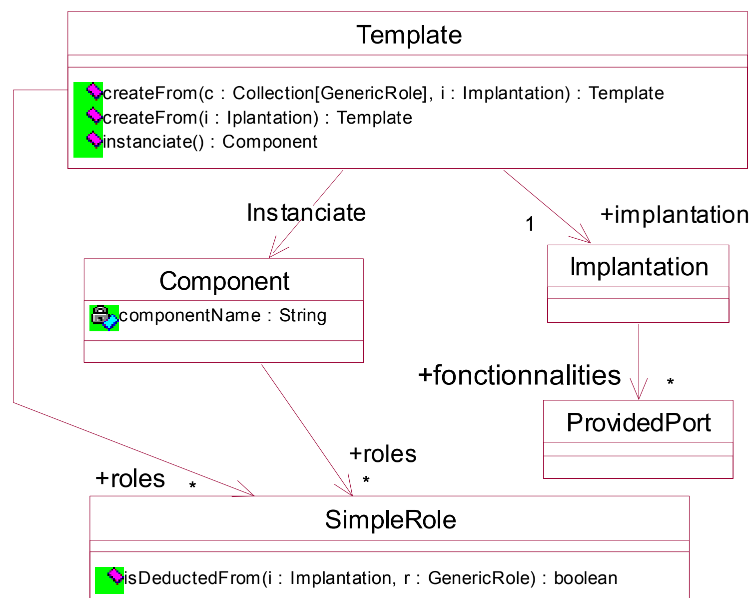

Fig. 3. UML representation of template, implantation and component

An **adaptation pattern** (see figure 4) has a name and defines a set of adaptation rules (see the definition below) on generic roles (its parameters).

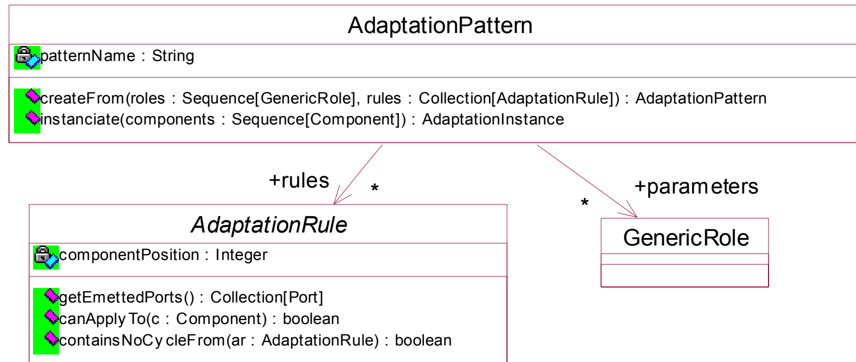

Fig. 4. UML representation of adaptation pattern

Suppose we now want to make our diary component persistent. A way to achieve this is that the diary notifies a data base of actions performed. Then we define an adaptation pattern specifying how to make a component persist (see figure 5). This pattern takes two parameters: the role of components we want to make persistent and the role of components providing the persistent storage. The *ComponentToPersist* generic role has three provided ports: add*(Any), remove*(Any) and search(Any): Any. The *PersistentSupport* generic role has four provided ports: load(): Any, store(Any), lock(Any) and unlock(Any). Note here that any component providing a *searchFor* port and two ports whose name begins with *add* and *remove* can be made persistent using this pattern. Similarly, any kind of persistence support (data base, file system, …) can be used providing that they offer *load* and *store* ports.

```
PersistencePattern(ComponentToPersist ctp,
                   PersistenceSupport ps)

1. ctp.add*(Any a) -> ctp.add*(a); ps.store(ctp)
2. ctp.remove*(Any a) -> ctp.remove*(a); ps.store(ctp)
3. ctp.searchFor(Any a) : (Any aa) -> ps.load(ctp);
                                      ctp.searchFor(a)
4. new ctp.lock() -> ps.lock(ctp)
5. new ctp.unlock() -> ps.unlock(ctp)
```

Fig. 5. Example of adaptation pattern composed of five rules: the PersistencePattern. "Any" is a special generic role representing all possible roles. The ";" operator stands for port sequencing

The operation of adaptation consists in applying an adaptation pattern to a set of components. The operation modifies a subset of the roles of the participating components and may add required components to some of the participating components (in this case an assembly is created). The operation also creates an **`adaptation instance`** (see figure 6) that has a name, comprises components and provides naviga-

bility functionalities between participants. To apply an adaptation pattern is a reversible action. Thus, we can undo the modifications carried out to the roles of the components (withdrawal of added ports or roles, suppression of the controls applied to some adapted ports, destruction of some component assemblies, etc).

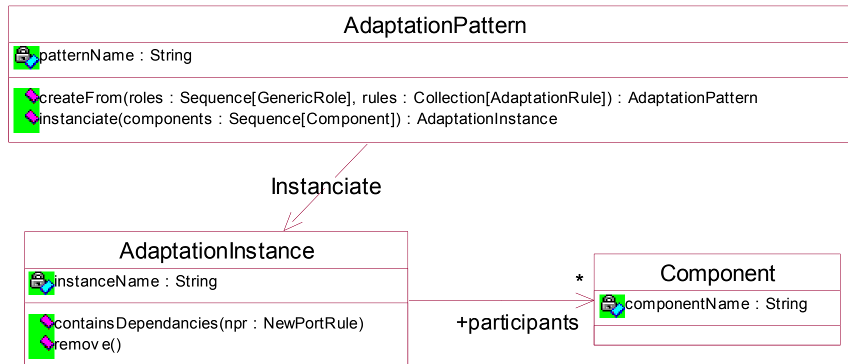

Fig. 6. UML representation of adaptation instance

An **`adaptation rule`** (see figure 7) describes explicitly what modifications on components are expected and the link between components. According to the Medvidovic software connector classification [20], our adaptation rules can be understood as implicit connectors. A rule is defined on a particular parameter of the adaptation pattern. Rules of adaptation can be of three categories:

- control rule consists in modifying the behavior of component ports,
- a new port rule consists in adding a provided port to component roles,
- new role rule consists in adding a new role to components.

For example, the *PersistencePattern* adaptation pattern (see figure 5) defines five rules: three control rules (1 to 3) and two new port rules (4 and 5). In this example, all rules are defined on the componentToPersist role parameter. Suppose we apply the *PersistencePattern* to a component *myDiary* exhibiting the role *BasicDiary* and to another component *myBD* exhibiting the role *DataBase*. Then the *BasicDiary* role of *myDiary* is modified and new features appear in it.

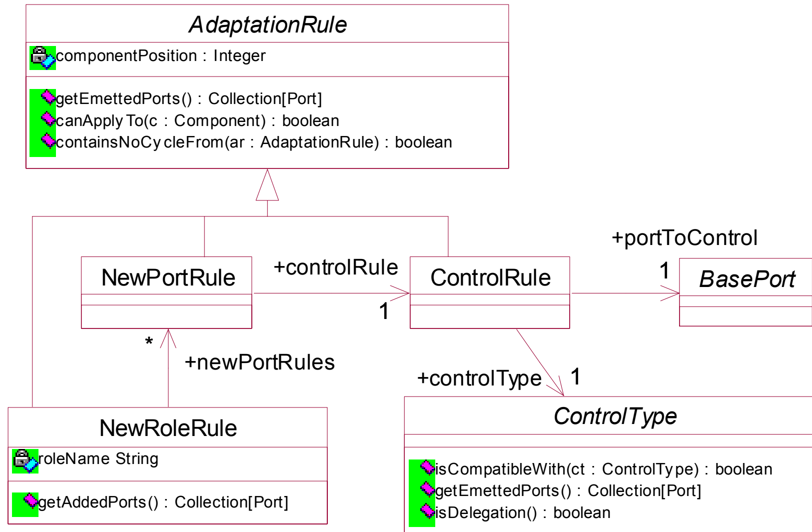

Fig. 7. UML representation of adaptation rules

Up to now, we have only seen one kind of port: provided ports. But adaptation rules involve two new kind of ports: emitted ports[4] and adaptation ports[5] (see figure 8). A set of **`emitted ports`** can be seen as a CCM receptacle [24] (whereas a set of **`provided ports`** can be seen as a CCM facet [24]). An emitted port appears in a role when a component requires this port to perform an action. Note that when an emitted port (*p*, *r*) is added in a role *r'* then the role of the receiver r of p is also added to the set of roles of required components of *r'*. An **`adapted port`** is a provided or an emitted port that is being adapted and whose behavior is being altered using **`controls`** (see figure 9).

[4] An emitted port is a pair characterized by a port *p* and the role of the receiver *r* (*p*, *r*).
[5] An adapted port is a triplet characterized by a port *p*, the role of the receiver *r*, the tree *ct* of controls applied to *p* and containing emitted ports (*p*, *r*, *ct*).

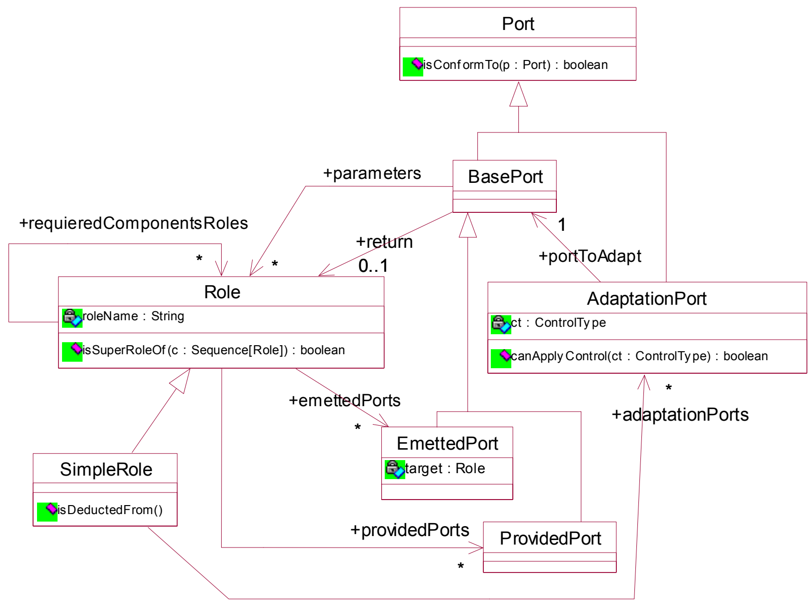

Fig. 8. UML representation of the different kinds of ports

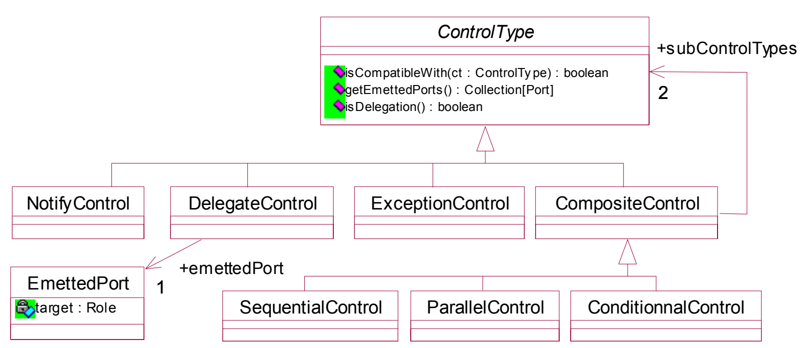

Fig. 9. UML representation of the different kinds of controls

Now that the different kinds of ports have been explained, we can express the effect of each kind of adaptation rule more formally.

A **`control rule`** alters the behavior associated with a port. The component roles concerned will be modified as follows: 1) addition of an adapted port, 2) addition of the emitted ports required by the adapted port, 3) addition of the roles of the required components (see figure 8).

If we consider our running example and according to rules 1, 2 and 3, three adapted ports are added to the role *BasicDiary* of *myDiary*: (*addMeeting, self, ct1*), (*remove-*

Meeting, self, ct2), (*getMeetings, self, ct3*) where *self* represents the currently considered role of the component (see the value of *ct1* in figure 10). Two emitted ports are also added to BasicDiary: (*store*, *DataBase*) and (*load*, *DataBase*). The role *DataBase* of the required component myDB is also added.

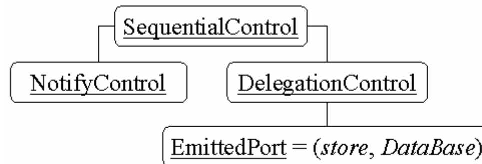

Fig. 10. Example of control tree: *ct1*

A **`new port rule`** adds a port to a component. The component roles concerned will be modified as follows: 1) addition of a provided port to the set of provided ports. 2) addition of an adapted port[6] to the set of adapted ports in order to give a code definition for this port.

A **`new role rule`** adds a role to a component. The component roles concerned will be modified as follows: addition of a role (comprising new provided ports and their associated adaptation ports) to the set of roles of the component.

According to rules 4 and 5, two new provided ports are added to the role BasicDiary of myDiary: *lock* and *unlock*. Two adaptation ports are added: (*lock, self, ct4*) and (*lock, self, ct5*), two emitted ports are added: (*lock*, *DataBase*) and (*unlock*, *DataBase*). This time, the role *DataBase* of the required component *myDB* does not need to be added as it has already been added via rules 1, 2 and 3. After applying *PersistencePattern* to *myDiary* and to *myDB*, each "*addMeeting*" (or "*removeMeeting*") call on *myDiary* is preceded by a "*store*" call on *myDB*.

3.2 Safety Properties

Section 3.2.1 informally outlines the expected properties to answer the three safety questions: *what* criteria determine the safe adaptations, *when* do safe adaptations have to be carried out and *how* do the modifications associated to a given safe adaptation have to be performed. Section 3.2.2 describes how to use OCL [38] to express our safety properties on the metamodel classes.

3.2.1 Informal Definitions

"What" properties

Currently we have listed five safety properties, which have to be guaranteed before an adaptation can be performed. Those properties can be understood as criteria that de-

[6] Emitted ports and roles of required components associated to the given adapted port are also added.

termine if a given adaptation is safe or not. Hence we allow an adaptation to be performed if, and only if, the following properties are fulfilled:

P_1: The adaptation conserves initial roles of a component: The initial roles of a component must be maintained during the component life cycle. This property is important to avoid errors due to a call to an unknown functionality.
If the BasicDiary role initially provides three ports corresponding to functionalities of adding, removing and consulting meetings then a component which initially presents the BasicDiary role will always be able to answer requests to those three functionalities.

P_2: The adaptation creates only valid assemblies: each required functionality is offered.

P_3: The adaptation takes into account component's semantic of use: Even if, structurally, two components offer the same ports, they may not be interchangeable in term of use.
A log component cannot be used where a database component is required (even if structurally they offer the same "store" port).

P_4: The adaptation does not introduce conflicts in functionality use. Several adaptations of the same port must be consistent: Delegating the responsibility of handling a functionality to different components, at the same time, may introduce an incompatibility, which we would like to warn of or even forbid.
Delegating the responsibility of handling the "store" functionality for diaries to different entities (a database and a file system for example) may mean that there is an incompatibility.

P_5: The adaptation accepts only cycles and non-deterministic points voluntarily defined by users.
Figure 11 shows an example of adaptation graph that introduces a non-deterministic point.

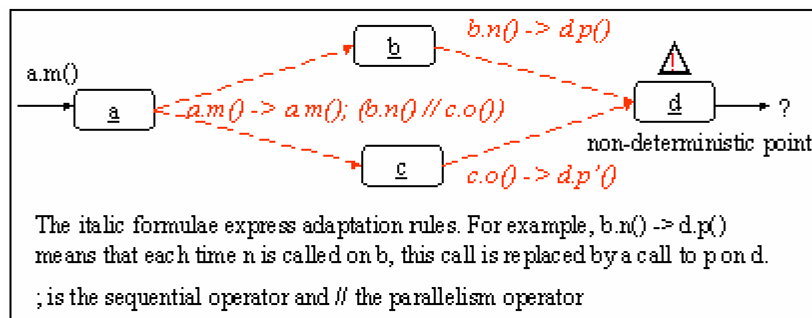

Fig. 11. Example of non-deterministic point. if d is a one-place buffer component with read (p) and write (p') functionalities, then there will be a conflict between read and write because they can be called in any order

"When" properties

After determining that a given adaptation is safe, the next step is to detect when it is safe to actually carry out the adaptation. A major criterion identifying a potentially "proper" moment is that there should be no communication between the components to be adapted and their environment. Otherwise *Atomicity* cannot be ensured (see the "*How"* properties in the following paragraph). But there is always an operation in execution. Then this criterion implies requiring the termination of the currently executing operation before handling an adaptation. And, this is neither pertinent as it may take quite a long time, nor sufficient as the modifications should not break the execution of one contiguous unit of work.

"How" properties

As for data base applications, we can reuse the ACID properties to perform adaptation-related modifications on components safely.

Atomicity guarantees that modifications performed within an adaptation undergo an all-or-nothing paradigm. In fact, this property ensures that actions performed during applying or unapplying a pattern appear as one contiguous unit of work. Consequently, this property guarantees that the restoration of the previous component state (the state just before the adaptation) is always possible.

If a diary is connected to a database and if we want to manage a disconnected mode between them, the adaptation has to be atomic. Otherwise, supposing only the diary has been adapted, any time a disconnection occurs, if the diary synchronizes the database then the database may be partially inaccessible.

Consistency guarantees that an adaptation will leave the system's state to be consistent after completion. In our approach the five « *What* » properties listed above are a set of rules that define a consistent system state. During the course of an adaptation, these rules may be violated, resulting in a temporarily inconsistent state. And when the adaptation completes, the state is consistent once more. As an adaptation executes atomically (from the Atomicity property above), the system will always appear to be consistent to a third party.

Isolation protects concurrently executing adaptations from seeing each other's incomplete results. Without isolation, the *consistency* property cannot be ensured. During an adaptation, locks on components must be assigned as necessary. The locks guarantee that, while an adaptation is occurring, no other concurrent updates can interfere. This allows many users to modify the same set of components simultaneously without concern for interleaving of adaptation operations.

Durability guarantees that updates survive failures (hard disks crashing, power failures, etc). In database applications, recoverable resources keep a log in order to be reconstructed by reapplying the steps in the log. In the context of adaptation processes, component state and component interconnections can be recovered from adaptation rules.

3.2.2 A Step towards Formalization

We have defined our metamodel using UML [27] and our approach to validating properties over the metamodel is to describe "behavioral" constraints in OCL [38], which must be satisfied by class instances. We have also defined "structural" constraints to specify, for example, that components, adaptation patterns and adaptation instances are identified by their names and that one cannot use the same role as participant in a given adaptation pattern twice. Currently, we have formalize all the "what" properties. Thus we have defined 16 structural invariants, two structural preconditions, 12 behavioral preconditions and five behavioral postconditions.

The constraints validating the "what" properties are listed below. Table 1 summarizes how each property is ensured and on which elements of our metamodel. Some OCL expressions are given in Figures 12 and 13 and in the appendix.

A_1: Any component required in a given role must always correspond to this role.
A_2: Any template connects a set of roles to an implantation so that each port of the roles is associated to an implantation port (that is a definition code).
A_3: Two structurally equivalent ports can belong to different roles of a component provided that they have an equivalent semantic of use.
A_4: Any component « can always answer » to functionalities associated with its roles.
A_5: To each port emitted in direction of a given required component role *r*, corresponds a provided port of *r*.
A_6: Each rule of an adaptation pattern does not generate any cycle.
A_7: Before applying an adaptation pattern to a sequence of components, each component involved in the adaptation must provide a role that conforms the parameter role of the pattern that the component is associated with.
A_8: For each control rule of an adaptation pattern, controls applied on the port to be adapted of the concerned component are not incompatible with new control to be applied.
A_9: For each rule of an adaptation pattern, its application to a component does not generate any cycle.
A_{10}: A pattern with a role or port addition rule cannot be unapplied to a set of components if, at least, another pattern instance adapts or uses the given added ports.

Table 1. "What" Properties synthesis

Property	Constraints	Elements concerned
Components initial roles conservation: P_1	A_1, A_2, A_4, A_7	Template, component, adaptation pattern *Uses*: IsSuperRoleOf
Valid assemblies construction: P_2	A_5, A_{10}	Adaptation pattern, adaptation instance *Uses*: containsDependancies
Component's semantic of use preservation: P_3	A_3	Template, role, component, implantation *Uses*: IsSuperRoleOf of Role
Conflicts in functionality use: P_4	A_8	Adaptation pattern *Uses*: CanApplyTo
Cycles and non-deterministic points: P_5	A_6, A_9	Adaptation pattern *Uses*: containsNoCycleFrom

In constraint A_5, `conforms` operation checks that a port *p* conforms another port *p'*. For a space preoccupation, we do not give OCL expression for operations substitutions expression here but, informally, conformity between ports corresponds to: name matching, for example, getX matches the get* regular expression; contravariance on parameters roles; and covariance on return role if defined. Conformity between roles (`isSuperRoleOf` operation) refers to a typing relation, which is satisfied when: for each provided port *p* of super role *r*, there is a provided port *p'* of sub role *r'* and *p'* conforms *p* (and if *r* or *r'* is a generic port: for each emitted port *p* of *r*, there is an emitted port *p'* of *r'* and *p'* conforms *p*). Then, the OCL precondition *constraint5* guarantees that for all rules, for each port emetted *ep* in direction of a role *r*, there is a provided port of *r* that conforms *ep* or *ep* is a port added to *r* within the current pattern.

```
context AdaptationPattern::createFrom(roles :
Sequence(GenericRole), rules : Collection(AdaptationRule)) :
AdaptationPattern

    pre A5a:
rules->forAll(ru | ru.getEmettedPorts()->forAll(ep |
ep.oclAsType(EmettedPort).target.providedPorts->exists(pp |
pp.conforms(ep)) xor rules->select(ru | not
ru.oclIsTypeOf(ControlRule) and ru.componentPosition =
Sequence{1..roles->size()}->select(i | roles->at(i).roleName =
ep.oclAsType(EmettedPort).target.roleName)->at(1))->iterate(
ru ; accu : Set(Port) = Set{} |
if ru.oclIsTypeOf(NewRoleRule) then
   accu->union(ru.oclAsType(NewRoleRule).getAddedPorts())
else
   accu->including( ru.oclAsType(NewPortRule).controlRule.port)
endif)->exists(ap | ap.conforms(ep))))


    pre A5b :
rules->select(ru | ru.oclIsTypeOf(ControlRule))->forAll(ru |
if ru.oclAsType(ControlRule).port.oclIsTypeOf(ProvidedPort) then
   roles->at(ru.componentPosition).providedPorts->exists(p |
   p.conforms(ru.oclAsType(ControlRule).port))
else roles->at(ru.componentPosition).emettedPorts->exists(p |
   p.conforms(ru.oclAsType(ControlRule).port) and
   p.oclAsType(EmettedPort).target.roleName =
   ru.oclAsType(ControlRule).port.oclAsType(EmettedPort).
                                             target.roleName)
endif)
```

Fig. 12. OCL Expression for A_5

In constraint A_{10}, `containsDependancies` operation checks if an added port of the adaptation instance is used or adapted in another adaptation instance. We cannot remove the adaptation instance until there are no more dependencies.

```
context AdaptationInstance::remove()
    pre A10:
self.pattern.rules->forAll(ru |
if ru.oclIsTypeOf(NewPortRule) then not
self.containsDependancies(ru.oclAsType(NewPortRule))
else if ru.oclIsTypeOf(NewRoleRule) then
ru.oclAsType(NewRoleRule).newPortRules->forAll(npr | not
self.containsDependancies(npr))
else true endif endif)
```

Fig. 13. OCL Expression for A_{10}

The aim of the approach is to simplify safety property proofs for component adaptation by reducing them to proofs of the sampler assumptions described by OCL constraints. The validity of the approach is also investigated by instantiating it to two component frameworks. We will take advantage of the work aroud transformation rules [15, 16, 18] to automatically translate our UML/OCL specification into a B specification.

4 Validating our Approach

Once that constraints have been defined formally using OCL, the next step is to check that they are syntactically correct. We also have to prove that they are consistent with respect to the property they have to guarantee[7]. Another step is to validate that the abstraction level of our metamodel is correct. For this, we have to verify that the safety properties proofs can be reused in specific component models. In this section, we show how we use a simulation technique [33] to validate our constraints (section 4.1) and how to integrate our safety properties proofs in two component models (section 4.2).

4.1 Validation Based on Simulation and on Proof Obligations

We need to validate the UML model and the OCL constraints not only syntactically to verify the specification against OCL grammar and type checking but also "semantically" to be sure that constraints cover exactly the property that they have to guarantee. Open-source tools for validating UML models are proposed in [33, 35]. Objecteering [35] provides a framework to dynamically test integrity constraints but implies defining constraints in a specific input language J. Gogolla et al [33] developed a tool called USE (UML-based Specification Environment) [34] to validate UML and OCL constraints from standard definition.

[7] There must be a bijection between constraints value domain and property value domain.

To validate a property by simulation with the USE tool, we check the property-related constraints. The property-related constraints are checked explicitly constructing snapshots representing system states. A snapshot is a set of objects with attribute values and links to other objects. For example, the snapshots we have tested against constraint A_2 are around a system of 64 states. Another point is that if any of the constraints is evaluated to "false" or has an undefined result, the system state is considered illegal and the snapshot is rejected. Then, the second step is to test known contra-examples "by hand". We isolate significant contra examples to verify that our properties recognize erroneous adaptations. We have tested if a template can be created from an implantation and from a set of roles that have some ports not corresponding to those of the implantation. We have tested this contra example on several port values and none of the snapshots check our constraint A_2 as we expected.

Nevertheless, to rapidly validate a large number of snapshots, we would like to construct the snapshots in a more declarative way for a systematic state space exploration. For this means, we use the language ASSL (A Snapshot Sequence Language) [13] on top of USE. ASSL serves to construct snapshots by specifying properties the desired snapshot has to fulfill instead of giving a explicit sequence of commands. Then, USE generates objects with randomly chosen attribute values and all possible link combinations between objects until a valid state space is found. As ASSL forbids side-effects, only invariants can be checked. Then we encountered some difficulties to translate pre and post condition into invariants. Preconditions are often defined on operation parameters. If operation parameters of an object *o* correspond to objects associated to *o*, there is no problem.

For example, the preconditions corresponding to constraint A_5 are applied on operation `create` of `AdaptationPattern` and deals with rules and parameters roles. It is really easy to transform these preconditions into invariants since roles and rules belong to the state of adaptation patterns. One only has to be careful to insert a supplementary condition: adaptation pattern links to its rules and to its roles must be defined. Otherwise, the first snapshot (where there are no links between objects) will always be chosen and the state space exploration will end. It is more difficult to translate postconditions since the result of the operation has to be taken into account. Instead of using post condition on instantiate operations, we move the constraint application onto the instantiated object. For example, post condition on `instantiate` operation of `AdaptationPattern` is moved into an invariant on `AdaptationInstance`.

By analyzing rejected and accepted system states, we have improved our specification. Desired snapshots that do not fulfill a constraint indicate too strong constraint. We corrected it by relaxing the constraints to include these cases. On the other hand, too weak constraints can accept undesirable system states. In this case, constraints are changed to be more restrictive. In fact, we cannot certify that a property is covered by a set of constraints that have to guarantee it and vice versa. Even if invalid snapshots are detected and valid snapshots pass all constraints, it cannot be generalized, and we cannot conclude the correctness of a specification in a formal sense.

Simulation has help us to find bugs and to gain confidence in our model. However, it only says that the model is correct with respect to the analyzed system states. Since one has to check that all accepted snapshots are really those we expected, it is a man-

ual process, which is subject to human limitations. Then there is always an error rate that we have to consider. Even if verification by theorem proving required more skill to use, the level of confidence obtained by proof verification is higher than with simulation techniques [33]. Using a formal method is essential to be more rigorous. Therefore, complementary to Use, we plan to use B formal method [1] in order to analyze and verify consistency and coherence of our UML/OCL specification.

4.2 Validation Based on Projection

The aim is not to modify existing component models to provide all the possible properties proofs but to help control adaptations allowed by each component model. This section presents two examples of projections: in the Julia implementation framework [5] of the Fractal model [22] and in the Noah framework [3, 32]. We have chosen Noah because it allows us to integrate and reuse a maximum of properties proofs. As the structure of the Julia model differs on many points from ours, we have chosen Julia to determine to what extent our metamodel has the right level of abstraction and is reusable.

4.2.1 Projection of the Elements of our Metamodel

The projections show that our metamodel is not a priori projected in totality in each component model, and each component model is not a fortiori completely abstracted. For each specific component model, only the elements corresponding to the adaptations provided in this specific component model can be projected (see Table 2).

Table 2. Projection of the metamodel elements synthesis. The × symbol means that there is no corresponding element in the concrete model for the considered metamodel element. The metamodel elements not presented in the table have no corresponding element neither in Noah nor in Julia

Metamodel element	Noah	Julia
Template	×	Template
Role	Name of Java interface	Interface type
Implantation	Public methods of Java class	Primitive component
Component	Noah object	Component
Provided port	Java method	Java method of server interface
Required component role	×	Client interface
Adaptation port	Interacting method	Interceptor/controller
Adaptation pattern	Interaction pattern	×
Adaptation instance	Interaction	×

4.2.2 Projection of the Safety Properties Proofs

This section deals with the projection of properties proofs in Julia and Noah frameworks. Table 3 summarizes which properties proofs are or may be projected and which projections are immediate as the property proofs are ensured in each framework. As for the projection of the metamodel elements, only the properties proofs

corresponding to the adaptations provided in that specific component model are projected.

Table 3. Projection of the safety properties proofs synthesis. The **!** symbol means that the projection is immediate as the property is completely ensured in the component model. The × symbol means that the adaptation corresponding to the property is not provided in the component model implying that the property does not have to be ensured. The **?** symbol means that we do not know how to project the proof of the property in the component model. The **o** symbol means that we know how to project the proof of the property

Safety properties	Noah	Julia
P_1	**o**	o for A_2 and A_4, × for A_1 and A_7
P_2	**!** for A_5, × for A_{10}	**!**
P_3	**!**	**!**
P_4	**!**	×
P_5	**o**	**o** for cycle, **?** for non deterministic points

Julia. Here is how the assertions corresponding to properties P_1 (A_1, A_2, A_4, A_7), P_2 (A_5, A_{10}) and P_3 (A_3) can be projected in Julia:

A_1: as Julia does not allow type evolution, this assertion does not have to be projected.
A_2: ensured by Julia template creation mechanism.
A_4: ensured by Julia template instantiation mechanism, which guarantees that components instantiated from a template *T* "can answer" to calls of operations of *T*.
A_7: as there is no equivalent element for the adaptation pattern in the Julia model, this assertion does not have to be projected.
A_5: ensured by typing rule on bindings. A client interface *It1* of a component *c* can be connected to a server interface *It2* of a component *s* if and only if *It2* is a subtype of *It1*.
A_{10}: ensured by the life cycle service. To alter the type of a component *s*, we have to stop all the client components of *s* simultaneously.
A_3: ensured by the Java type system. Common methods of several interfaces implemented by a class have automatically the same implementation because we cannot have several methods with the same signature. Then, common methods automatically have the same semantic of use automatically.

Property P_4 does not have to be ensured in Julia. The initial behavior of services associated to a given component can be modified by using controllers. Controllers desired for a given component are defined in a property file, which is read when the application is loaded. But at runtime, the set of controllers applied on a given component cannot be altered. Then, we do not have to manage service composition dynamically.

Property P_5 is partially ensured by containment cycle detection. Indeed, containment cycle detection eliminate the cycles in methods calls that are the result of cycles of containment but it does not eliminate all cycles in methods calls. However, no projection of this property is studied since Fractal philosophy states that cycles in sequence of method calls are perfectly legal and are consider to be voluntary defined by programmer. The programmer has then in charge to ensure that the introduced cycles

do not produce infinite loop. Currently, we do not know how to project the detection of non-deterministic points because parallelism is not standardized in Julia.

Noah. As Noah implements a lot of adaptations, all the properties can be, at least partially, projected. Here is how the 10 assertions can be projected in Noah:

A_1: ensured by typing rules on role when applying interaction patterns.
A_2: ensured by the "implement" mechanism verified by the Java compiler.
A_3: ensured by the Java type system. Common methods of several interfaces implemented by a class have automatically the same implementation because we cannot have several methods with the same signature. Then, common methods automatically have the same semantic of use.
A_4: ensured by the Java instantiation mechanism, which guarantees that objects instantiated from a class A "can answer" to calls of methods of A.
A_5: formal participants of an interaction are represented by a name. Implicitly, the role of a formal participant p is equal to the union of services addressed to p in left part of interaction rules and services addressed to p in right part of interaction rules. Then component assemblies are supposed to be correct provided that effective participants roles are conform to formal ones. The projection of the assertion is done by the intermediate of the projection of assertion A_7.
A_6: since interaction rules are navigable, the OCL assertion can be directly projected into a Java assertion.
A_7: since we can extrapolate the role of each participant of the interaction (see projection of assertion A_5), we project the assertion using the Java reflexivity to determine if a component (effective participant) is a good candidate to fulfill the role of a given participant.
A_8: ensured be the Noah merging mechanism, which composes behaviors.
A_9: since interaction rules and components are navigable, the OCL assertion can be directly projected into a Java assertion.
A_{10}: adding of functionality is not implemented in Noah. Then, the assertion does not have to be projected.

Synthesis. For the moment, all projections are performed manually. We have to insert code in some methods of some elements of each concrete model. Currently, we are not able to automate the projection process. However, the metamodel reifies the key elements of the process of adaptation, and we can determine how the metamodel elements are projected in concrete models. Then, we can locate precisely where the modifications have to be carried out. With the help of rewriting rules attached to some elements of the metamodel, we hope to be able, at least, to generate code automatically per target.

5 Conclusion and Future Work

In this paper, we have shown that to determine the safety of dynamic adaptations of components we have to answer three questions: 1) *what* are the criteria of safe adaptations, 2) *when* do adaptations have to be carried out and 3) *how* do the modifications

associated to a given adaptation have to be performed. To answer these questions, our approach is to define what we have called “safety properties”. Then, we have proposed a metamodel for adaptable components using UML [27], and safety properties have been ensured using OCL [38] constraints attached to our metamodel classes.

Due to the abstraction level provided by the metamodel, we have already validated the main properties proofs by simulation [33]. Indeed, by analyzing rejected and accepted system states, we have improved definitions of the OCL constraints [38] but since we cannot fairly check the states space entirely and as there is always an error rate to consider, we plan to use B formal method [1] in complement. The properties proofs validated on the metamodel ensure that no bugs will appear at runtime in each component model where the proved properties proofs are projected. Indeed, once a proof projection is proven consistent with respect to the property it guarantees, the given property is always preserved in the concerned component model independently from the application it hosts.

Our work around projections allows us to evaluate how far our approach is reusable because we have chosen, as projection targets, two very different component models. Noah [3, 32] defines component assemblies on interaction patterns and Julia [5, 22] on structural composition based on containment and bindings. Moreover, safety of adaptations is integrated into the targeted frameworks: we have shown how to project some properties proofs.

Our future work will consist of different issues, from the consolidation and extension of our metamodel and its safety properties to a general validation of our proposal mentioned above. We will strengthen the OCL [38] constraints associated to the property P_3 (component’s semantic of use conservation) by adding a behavioral subtyping relation [11] to our role definition. We will also go deeper into the validation of our OCL constraints using B formal method [1]. On the other hand, our metamodel should be easily extended to new kinds of adaptation. In particular, we will take into account communication mode switching [6]. Finally, we will automate the projection process for some component models implemented in Java by meta programming.

References

1. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press, ISBN 0-521-4961-5 (1996)
2. Adamek, J., Plasil, F.: Behavior Protocols Capturing Errors and Updates, Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS, published by University of Warsaw, Poland, Apr 2003
3. Blay-Fornarino, M., Ensellem, D., Occello, A., Pinna-Dery, A.M., Riveill, M., Fierstone, J., Nano, O., Chabert, G.: Un service d'interactions: principes et implémentation. In Journée composants: Systèmes à composants adaptables et extensibles, Grenoble, France (2002)
4. Briais, S.: Mobile Objects "Must" Move Safely. Uwe Nestmann (EPFL), FMOODS'02
5. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and Dynamic Software Composition with Sharing. In Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02) (2002)

6. Budau, V., Bernard, G.: Synchronous/Asynchronous Switch for a Dynamic Choice of Communication Model in Distributed Systems. In 9th International Conference on Parallel and Distributed Systems, Taiwan, ROC (2002)
7. Cardelli, L.: Type systems. ACM Computing Surveys (1996) 263-264
8. Caron, O., Carré, B., Debrauwer, L.: An original View mechanism for CORBA middleware. Int. Conf. Tools'2000, Mont St Michel, France (2000)
9. Corbett, J., Dwyer, M., Hatcliff, J., Robby: A Language Framework For Expressing Checkable Properties of Dynamic Software. Proceedings of the 2000 Spin Workshop. LNCS 1885 (2000) 205-223.
10. Ducournau, R.: Spécialisation et sous-typage : thème et variations. Technique et science informatiques (2002)
11. Findler, Latendresse, Felleisen: Behavioral Contracts and Behavioral Subtyping. FSE 2001
12. IS0: Open Distributed Processing Reference Model - parts 1-4. International StandardOrganization, ISO 10746-1..4 (1995)
13. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. 6th International Conference on the Unified Modeling Language, UML'03, San Francisco, United States of America (2003)
14. Leblanc, S. , Merle, P.: TORBA : vers des composants de courtage. In Sixth International Conference on les Langages et Modèles à Objets, L'Objet vol. 8, n° 1-2/2002, Montpellier, France (2002) 185-201
15. Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from UML Notations. In: Proceedings of the 15th Int. Conf. on Automated Software Engineering, ASE'2000 (2000)
16. Marcano, R., Levy, N.: Using B formal specifications for analysis and verification of UML/OCL models. Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language (2002)
17. Mc Affer, J.: Meta-level programming with CodA. In ECOOP'95. SpringerVerlag (1995)
18. Meyer, E., Souquières, J.: A systematic approach to transform OMT diagrams to a B specification. In: FM'99: World Congress on Formal Methods in the Development of Computing Systems (1999)
19. Medvidovic N., Richard N. Taylor R. N.: Classification and Comparison Framework for Software Architecture Description Languages. Software Engineering (1997)
20. Mehta N. R., Medvidovic, N.,Phadke S., "Towards a Taxonomy of Software Connectors", in Proceedings of the 22th International Conference on Software Engineering(ICSE2000), Limerick, Ireland, 2000
21. Mulet, P., Malenfant, J., Cointe, P.: Towards a methodology for explicit composition of metaobjects. In Proceedings of OOPSLA'95 (1995) 316-330
22. Objectweb: Objectweb component model. Internet: http://fractal.objectweb.org/
23. Nano, O., Blay-Fornarino; M., Dery, A-M., Riveill, M.: Using MDA to integrate services in component platforms. In Height International Workshop on Component-Oriented Programming in conjunction with ECOOP'2003 (to be published)
24. OMG: CORBA 3.0 New Components Chapters. OMG TC Document ptc/01-11-03
25. OMG: Model Driven Architecture. OMG TC Document ormsc/01-07-01
26. OMG: Meta Object Facility (MOF) Specification. OMG Document AD/97-08-14.
27. OMG: Unified Modeling Language Specification. OMG TC Document formal/03-03-01.
28. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A Flexible and Efficient Solution for Aspect-Oriented Programming in Java. Reflection, vol. LNCS 2192, Springer-Verlag (2001) 1-24
29. Peschanski, F.: A Versatile Event-Based Communication Model for Generic Distributed Interactions. ICDCS Workshops (2002) 503-510

30. Pinna-Dery, A.M., Blay-Fornarino, M., Arcier, B., Mule, L., Moisan, S.: Distributed access knowledge-based system: Reified interaction service for trace and control. In 3nd International Symposium on Distributed Object Applications - DOA 2001, Italy, 76-84
31. Plasil, F., Balek, D., Janecek, R., "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.
32. Rainbow team: Noah home page. Internet: http://noah.essi.fr/
33. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), Springer, Berlin, LNCS 1939 (2000) 265-277
34. Richters, M.: The USE tool: A UML-based specification environment. Internet: http://www.db.informatik.uni-bremen.de/projects/USE/
35. Softeam: the Objecteering/UML tool suite. Internet: http://www.objecteering.com/ (2002)
36. Segarra, M.T., André, F.: A Framework for Dynamic Adaptation in Wireless Environments. Proc. of TOOLS Europe 2000, Mont St. Michel, St. Malo, France (2000)
37. Sun: EJB Specification Version 1.1. Internet: http://java.sun.com/poducts/ejb/docs.html
38. Warmer, J., Kleppe, A.: OCL: The constraint language of the UML. Journal of Object-Oriented Programming (1999)

Appendix: OCL Description of Constraints

Note that in precondition *constraint7*, pattern instantiation is atomic since we check that *all* components (*forAll* statement) can conform its associated role of the pattern *before* creating the effective adaptation instance. Consequently, the *atomicity* property is also guaranteed by this constraint.

Constraint A1:

```
context Component::methodCall1(method : ProvidedPort,
effectiveParams : Sequence(Component))
    pre A1_1a:
method.return->isEmpty and self.roles->forAll(r |
r.providedPorts->exists(p | p.conforms(method))) and
effectiveParams->size() = method.parameters->size() and
Sequence{1..method.parameters->size()}->iterate(index : Integer;
accu : Boolean = true | accu and
method.parameters->at(index).isSuperRoleOf(
effectiveParams->at(index).roles->asSequence()))

context Component::methodCall2(method : ProvidedPort,
effectiveParams : Sequence(Component)) : Role
    pre A1_1b:
method.return->notEmpty and self.roles->forAll(r |
r.providedPorts->exists(p | p.conforms(method))) and
effectiveParams->size() = method.parameters->size() and
Sequence{1..method.parameters->size()}->iterate(index : Integer;
accu : Boolean = true | accu and
method.parameters->at(index).isSuperRoleOf(
effectiveParams->at(index).roles->asSequence()))
    post A1_2:
result.isSuperRoleOf(Sequence{method.return})
```

```
context Component::declaration_assignement(waitingRole : Role,
providedComponent : Component) : Component
    pre A1_3:
waitingRole.isSuperRoleOf(providedComponent.roles->asSequence())
    post A1_4: result.roles = providedComponent.roles
```

Constraint A2:

```
context Template::createFrom(i : Implantation) : Template
    post A2_1:
i.fonctionnalities->forAll(fct | result.roles->exists(r |
r.providedPorts->includes(fct)))

context Template::createFrom2(c : Collection(GenericRole), i :
Implantation) : Template
    pre A2_2a:
c->forAll(r | r.providedPorts->forAll(p1 |
i.fonctionnalities->exists( p2 | p1.conforms(p2) ))) and
c->forAll(r1, r2 | r1 <> r2 implies not (r1.roleName =
r2.roleName))
    post A2_2b:
result.roles->size() = c->size() and
result.roles->forAll(r1 | c->exists(r2 |
r1.isDeductedFrom(i, r2)))
```

Constraint A4:

```
context Template::instanciate(componentName : String) :
Component
    post A4: result.roles = self.roles
```

Constraint A7:

```
context AdaptationPattern::instanciate(components :
Sequence(Component)) : AdaptationInstance
    pre A7:
Sequence{1..components->size}->forAll( index : Integer |
self.parameters->at(index).isSuperRoleOf(
components->at(index).roles->asSequence()))
```