

Safe Runtime Adaptations of Components: a UML Metamodel with OCL Constraints

Audrey Occello¹ and Anne Marie Dery-Pinna²

*Laboratoire I3S - Bat. ESSI
930, Route des Colles, B.P. 145
06903 SOPHIA-ANTIPOLIS Cedex, France*

Abstract

With the increasing use and 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. We define a metamodel for adaptable components on which we describe safety properties independently from component platforms.

Key words: adaptation safety, components, metamodeling.

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 needs 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 these models are emerging to support adaptability. By comparing some component models [22], [14], [6], [19], [16], [4], [23], we can see that they do not provide the same kinds of adaptation (add/remove functionality, change behavior of

¹ Email: occello@essi.fr

² Email: pinna@essi.fr

a functionality, alter component assembly, etc). Moreover, same adaptations may be implemented differently in distinct component models. For example, CCM [14] bases its assembly adaptation on binding modifications, SOFA [19] and Fractal [6] on containment compositions, ACEEL [23] on automaton compositions, and Noah [4] on interaction pattern applications. One problem with these approaches is that each adaptation process implies modifying components. Although runtime adaptations often lead the application to an unsafe state, the problem of determining the safety of component dynamic adaptations has received, in our opinion, little attention of needed.

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. We have chosen to work at a meta level: we define a metamodel for adaptable components on which we describe safety properties independently from component platforms. This generic approach is usable by any existing component platform to make dynamic adaptations safer.

Section 2 presents how the safety of dynamic adaptations is handled in existing component models. Section 3 describes a metamodel, which highlights the key elements involved in component adaptations. Section 4 presents safety properties. 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. 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 [14] 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 Assembly 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. Nevertheless, the component model

does not provide means to check that the behavior of the new component conforms the behavior of the one replaced.

SOFA. SOFA [19] 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 [1]. Behavioral protocols describe how components can be used: the behavior of a SOFA component is the set of all traces, which can be produced by the component. SOFA allows one kind of adaptation: assembly modification. Through 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 protocol 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. Thus, SOFA components do not need to be stopped, while being adapted, in contrast to ACEEL components (see description in the next paragraph). 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 [23] 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 [6], 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.

Noah. Noah [4] is a framework providing a dynamic adaptation layer over the EJB [22]. The framework allows the programmer to express the interactions between components declaratively and externally to components via the ISL language [18]. By describing interaction rules 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 to a component. 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.

Synthesis. In conclusion, each component model checks the safety of some adaptations. But the validations are not systematically performed, 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 everywhere to determine the safety of the three kinds of adaptations:

- Type evolution consistency: if we consider that the component type corresponds to the set of functionalities it offers, then we can compare this need to a change of the component type during its life cycle. However, work on typing as defined in Object Oriented applications [8] are not sufficient to validate type evolutions during execution.
- 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 metamodelling [12] or in AOP [16] should be extended to take into account behavior composition coherence.
- 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 [10] 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 [14], Fractal [6], SOFA [19] and ACEEL [23]. 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 [23] and Fractal [6] require components to be stopped during adaptation. SOFA [19] proposes to determine a priori the moments of adaptation. But these techniques are not appropriate: the first worsen performance of the system and is too drastic, the second is too constraining and not enough flexible.

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 these questions can be answered by determining “safety properties” (section 4.1). Then, our approach is based on a metamodel for adaptable components (section 3) using UML [15], which allows us to set the vocabulary. In addition, we define OCL [24] constraints, which are checked on class instances (section 4.2) to ensure the safety properties.

3 A Metamodel for Adaptable Components

An informal definition of each element of the metamodel is given in this section. To illustrate the metamodel, we will consider, as a running example, 1) diary components which provide functionalities to add, remove, and consult appointments, 2) database components which provide functionalities to memorize and restore states and to synchronize data.

A **role** (figure 1) has a name and describes component properties. There are two kinds of roles. A **generic role** expresses properties that the components participating in a collaboration must exhibit as for UML interfaces [15] or role modeling [3], [21]. For example, *AttributeManager* is a generic role that offers generic getters and setters functionalities. A **concrete role** is associated with a particular component in contrast to generic roles. It describes what is provided and required as well as the adaptations done on the component. It must not be confused with classical typing approaches [8] because it supports dynamic features additionally and it may be seen as a metaobject controlling the behavior of its component like Peschanski roles [17]. Examples of concrete roles will be done in the remainder of this section. Note that generic roles can be seen as formal parameters (of methods) and concrete roles as parameter value.

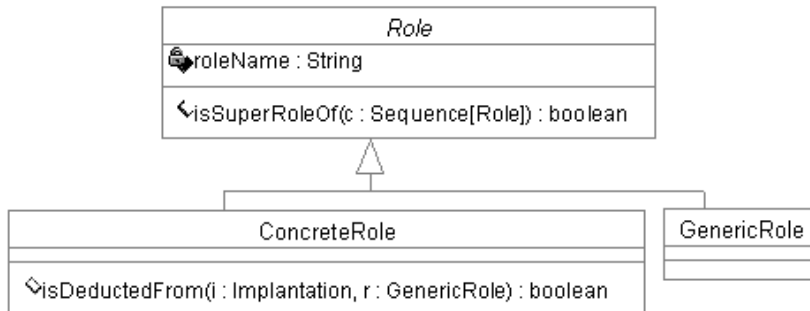


Fig. 1. UML representation of roles

Each **port** (figure 2) is determined by a signature: an expression to name the port, a set of parameter roles and possibly a return role. We distinguish two types of ports. A port with a **name** captures and handles a particular functionality addressed to the component. A port with an **expression** can only be defined in a generic role and may correspond to a set of functionalities.

For example, the *AttributeManager* generic role has two provided ports: $get^*(Any)$ and $set^*(Any)$ where get^* (resp. set^*) is an expression represent-

ing getters (resp. setters) functionalities and *Any* is a special generic role representing all possible roles. The *BasicDiary* concrete role, associated with diary components, has three provided ports: *addMeeting(Meeting)*, *removeMeeting()* and *searchFor(Date): Meeting*. The *DataBase* concrete role, associated with database components, has four provided ports: *load(): BasicDiary*, *store(BasicDiary)*, *lock(BasicDiary)* and *unlock(BasicDiary)*.

Note that *BasicDiary* (resp. *DataBase*) can be both a generic and concrete role : it will be 1) a generic role when determining the role of a parameter (with provided/required features), and 2) a concrete role when associated with a component (with not only provided/required features but also adaptation features) but, in fact, it will not be the same role.

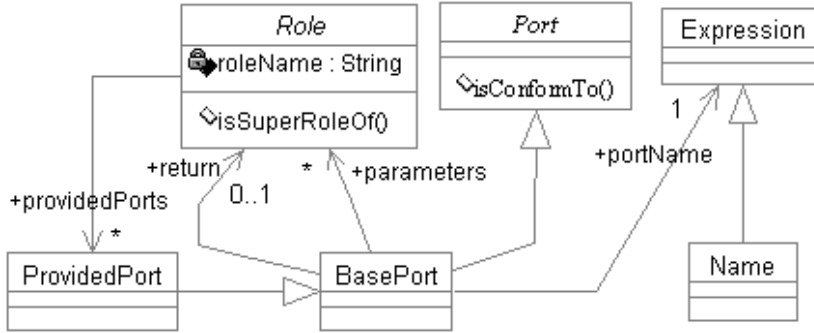


Fig. 2. UML representation of ports

A **template** (figure 3) defines a category of components and instantiates components. An **implantation** (figure 3) is the representation, at the meta-model 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.

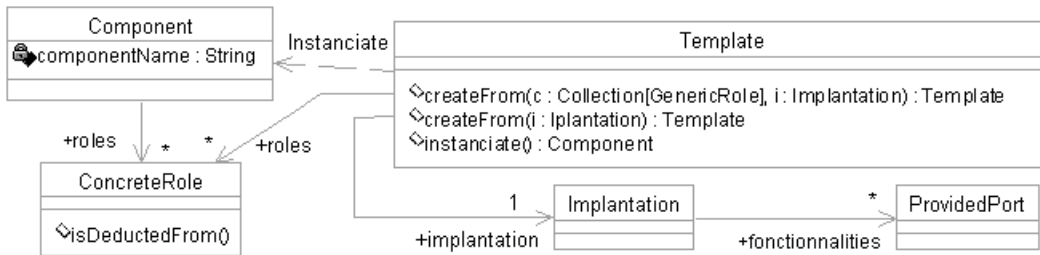


Fig. 3. UML representation of template, implantation and component

A **component** (figure 3) is an instance of a template, a unit of execution. A component is associated with concrete roles (*myDiary* is associated with a *BasicDiary* role and *myDB* with a *DataBase* role), which can be seen as view points similar to RM-ODP [9] and CCM [14]. In contrast with these work, we can use a component according to a subset of its roles simultaneously. At any

time, the roles of a component reflect its interactions with the environment. The roles of a component 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 evolve independently.

We believe that describing adaptations is better than programming because it checks for safety before performing the modifications associated with adaptations. Then, an **adaptation pattern** (figure 4) 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 adaptation pattern has a name and defines a set of adaptation rules (see the definition below) on generic roles (its parameters).

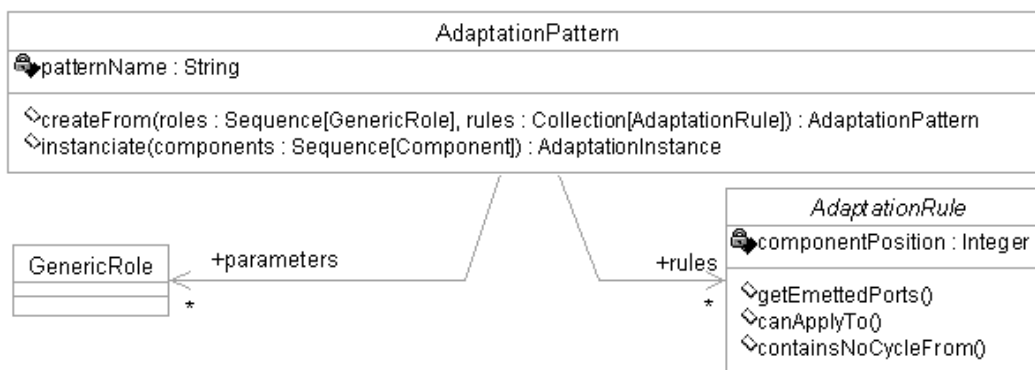


Fig. 4. UML representation of adaptation pattern

For example, the way to make a component persistent is to define an adaptation pattern (see the code, written in an ISL-like style [18], below). The PersistencePattern³ 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)*. Then, we can use this adaptation pattern with diary and database components: diaries notify a data base of actions performed Note 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 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)
    
```

³ 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** (figure 5) that has a name, comprises components and provides navigability 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/roles, suppression of the controls applied to adaptation ports, destruction of component assemblies, ...).

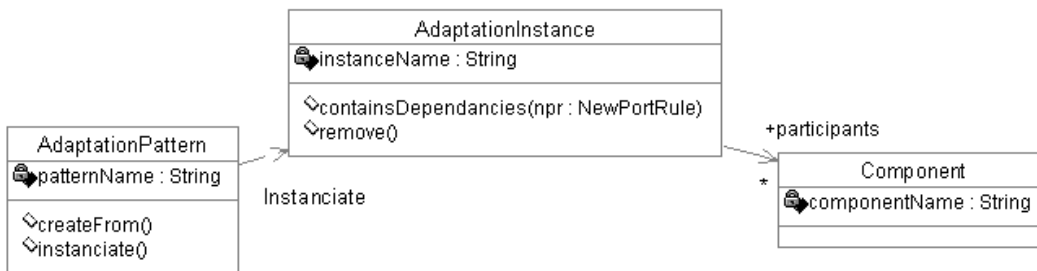


Fig. 5. UML representation of adaptation instance

An **adaptation rule** (figure 6) describes explicitly what modifications on components are expected and the link between components. Adaptation rules can be seen as an abstraction of declarative adaptation techniques such as [2], [5] or [4]. According to the Medvidovic software connector classification [11], they can also be understood as implicit connectors with ADL connection ends (generic roles) that are attached to ADL ports/interfaces (concrete roles). A rule is defined on a particular parameter of the adaptation pattern. Rules of adaptation can be of three categories: 1) a **control rule** consists in modifying the behavior of component ports, 2) a **new port rule** consists in adding a provided port to component roles, and 3) a **new role rule** consists in adding a new role to components.

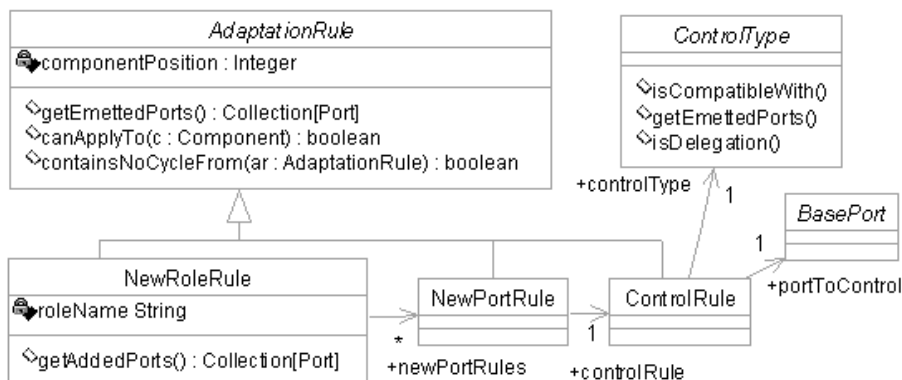


Fig. 6. UML representation of adaptation rules

For example, *PersistencePattern* (see the code above) 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 *PersistencePattern* to *myDiary* as *ComponentToPersist* and to *myDB* as *PersistenceSupport*. Then the *BasicDiary* role of *myDiary* is modified and new features appear in it.

Up to now, we have only seen one kind of port that represent provided features. A set of **provided ports** can be seen as an UML 2.0 provided port or a CCM facet. Adaptation rules involve two new kind of ports: **emitted ports** and **adaptation ports** (figure 7). An emitted port appears in a role when a component requires this port to perform an action. But, whereas an UML 2.0 required port and a CCM receptacle reflects the required functionalities of a component at design time, emitted ports only deal with requirements at adaptation time. 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 **required component roles** (figure 7) of r' . An adaptation port is a provided or an emitted port whose behavior has been altered using **controls** (figure 8).

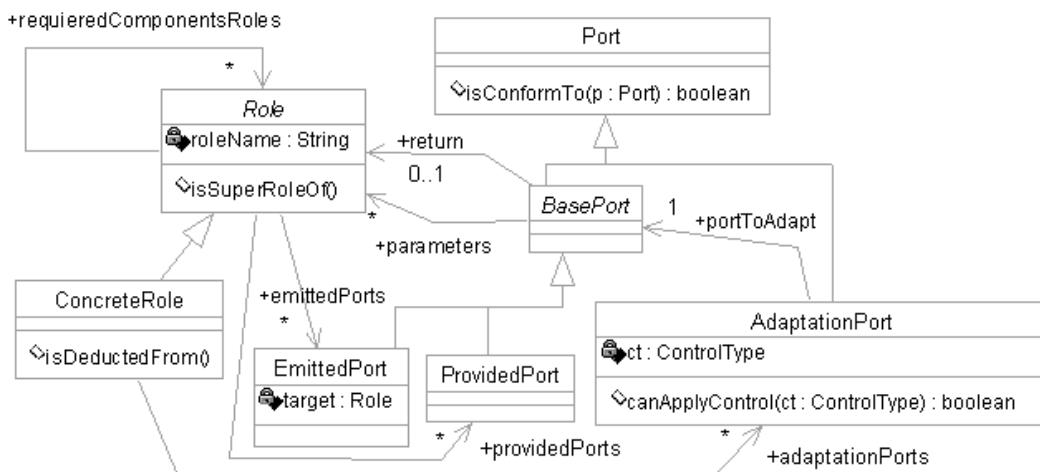


Fig. 7. UML representation of the different kinds of ports

Each kind of adaptation rule expresses modifications on components. A **control rule** alters the behavior associated with a port. The component roles will be modified as follows: 1) addition of an adaptation port, 2) addition of the emitted ports associated with the adaptation port, 3) addition of the required components roles.

According to rules 1, 2 and 3 of *PersistencePattern*, three adaptation ports are added to the *BasicDiary* role of *myDiary*: (*addMeeting*, *self*, *ct1*), (*removeMeeting*, *self*, *ct2*), (*getMeetings*, *self*, *ct3*) where *self* represents the currently considered role of the component (see the value of *ct1* in figure 9). Two emitted ports are also added to *BasicDiary*: (*store*, *DataBase*) and (*load*, *DataBase*). The role *DataBase* of the required component *myDB* is also added.

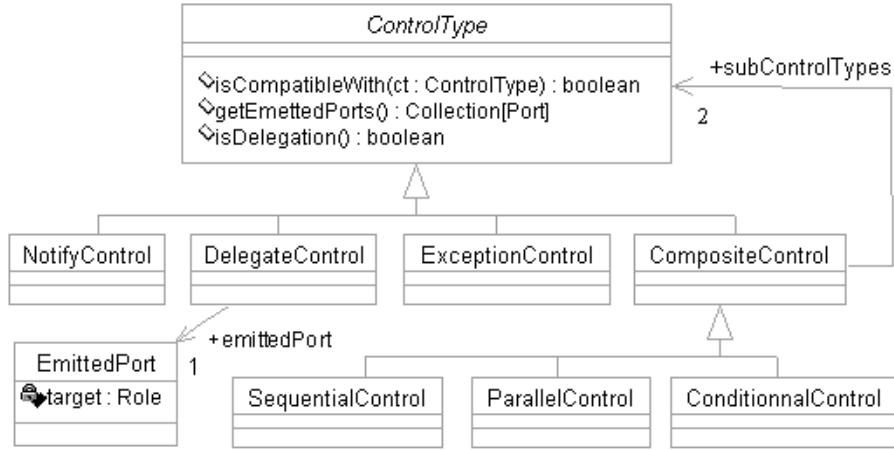


Fig. 8. UML representation of the different kinds of controls

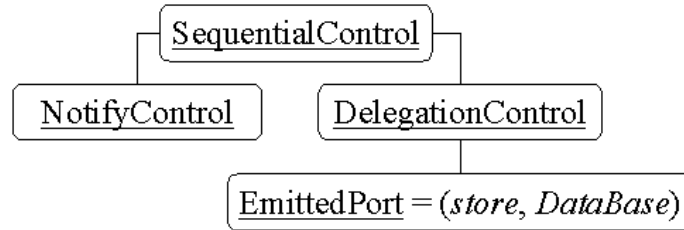


Fig. 9. Example of control tree: *ct1*

A **new port rule** adds a port to a component. The component roles will be modified as follows: 1) addition of a provided port to the set of provided ports. 2) addition of an adaptation port to the set of adaptation ports in order to give a code definition for this port. A **new role rule** adds a role to a component. The component will be modified as follows: addition of a new 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 *myDiary*'s *BasicDiary*: *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 *myDB* does not need to be added as it has already been added. After applying *PersistencePattern* to *myDiary* and to *myDB*, each *addMeeting* (or *removeMeeting*) call on *myDiary* is preceded by a *store* call on *myDB*.

4 Safety Properties

Section 4.1 informally outlines the expected properties to answer the three safety questions: *What*, *When* and *How* (see section 2). Section 4.2 describes how to use OCL [24] to express the safety properties on the metamodel classes.

4.1 Informal Definitions

“What” properties: Here, we detail five safety properties, which have to be guaranteed before an adaptation can be performed. These properties can be understood as criteria that determine if a given adaptation is safe or not.

- P_1 : The adaptation **conserves initial roles** of a component. 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 from these 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 two components offer structurally-equivalent ports, they may not be interchangeable in term of use. *A log component cannot be used where a database component is required (even if they offer the same “store” port).*
- P_4 : The adaptation does not introduce functionality conflicts: **adaptation rule composition coherence**. *Multiple adaptations of a component may lead to delegate the responsibility of handling a functionality to different entities, which may introduce an incompatibility.*
- P_5 : The adaptation **warns about cycle** detection. *Let consider three components a , b and c . a has been adapted to delegate the responsibility of handling the m functionality to b . b has been adapted to notify c (that is: b calls n on c) every time m is called on b . Now suppose that c is adapted to delegate the responsibility of handling the n functionality to a . Then a cycle is introduced by propagation indirectly.*

“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 environment⁴. But there is always an operation in execution. Then this criterion implies requiring the termination of the currently executing operation before handling an adaptation. 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, an adaptation has to be performed as an ACID transaction.

4.2 A Step towards Formalization

Our approach to guaranteeing the properties over the metamodel is to describe constraints in OCL [24], which must be satisfied by class instances. The OCL constraints that formalize the five “*What*” *properties* listed in section 4.1, correspond to a set of preconditions, postconditions and invariants on methods and classes of the metamodel. All the OCL expressions corresponding to these

⁴ Otherwise *Atomicity* cannot be ensured (see the “*How properties*” in the following paragraph).

constraints can be found in [13]. Next table summarizes how each property is ensured and on which elements of the metamodel.

Property	Constraints	Elements concerned	Functionalities used
P_1	A_1, A_2, A_4, A_7	Template, Component, AdaptationPattern	IsSuperRoleOf
P_2	A_5, A_{10}	AdaptationPattern, AdaptationInstance	containsDependancies
P_3	A_3	Template, Role, Component, Implantation	IsSuperRoleOf
P_4	A_8	AdaptationPattern	CanApplyTo
P_5	A_6, A_9	AdaptationPattern	containsNoCycleFrom

In this paper, we describe the OCL constraints that validate properties “initial role conservation” p_1 and “valid assemblies construction” p_2 . A_1, A_2, A_4 and A_7 (resp. A_5 and A_{10}) are the translation in “natural language” of the OCL constraints that guarantee p_1 (resp. p_2).

- A_1 : Any component required in a given role must always correspond to this role. This constraint can evolve in time.
- A_2 : Any template connects a set of roles to an implantation so that each port of the roles is associated to a definition code.
- A_4 : Any component “can always answer” to functionalities associated with its roles.
- A_7 : Before applying an adaptation pattern to a sequence of components, the role of the involved components must conform to the parameter roles of the pattern.
- A_5 : To each port emitted in direction of a given required component role r , corresponds a provided port of r .
- 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 of the same component.

In constraint A_5 (see the corresponding OCL expression below), the *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 A_{5a} guarantees that for all rules, for each port emitted 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.getEmittedPorts()->forall(ep |
  ep.oclAsType(EmittedPort).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(EmittedPort).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 : ...

```

In constraint A_{10} (see the corresponding OCL expression below), the *containsDependencies* 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.containsDependencies(ru.oclAsType(NewPortRule))
else if ru.oclIsTypeOf(NewRoleRule) then
ru.oclAsType(NewRoleRule).newPortRules->forall(npr | not
self.containsDependencies(npr))
else true endif endif)

```

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. Our approach is to define “safety properties” to be fulfilled by components and adaptations. Then, we have proposed a metamodel for adaptable components using UML [15], and “*What*” *properties* have been ensured using OCL [24] constraints attached to the metamodel classes.

Independence from middleware allows us to prove the properties formally and then prevent from runtime adaptation problems in each component model where the properties are projected. The validation of the safety properties is reduced to the validation of the sampler assumptions described by OCL constraints. The validation of the OCL constraints consist in proving that they are consistent with respect to the property they have to guarantee⁵. It has been done using simulation techniques [20]. The validation of the metamodel itself has been investigated by instantiating it to two component frameworks: Fractal [6] and Noah [4]. This work has helped us to evaluate how much the

⁵ Constraint value domain and property value domain must be the same.

approach is reusable because we have chosen, as projection targets, two very different component models. For further details on these validations, see [13].

Future work will consist of the consolidation and the extension of the metamodel and its safety properties. Indeed, the metamodel should be easily extensible to new kinds of adaptation. In particular, we will take into account communication mode switching [7]. On the other hand, we will make a prototype of a generic safety service that ensures “*What*” and “*How*” properties.

References

- [1] Adamek, J. and F. Plasil, *Behavior protocols capturing errors and updates*, in: *Proceedings of USE*, University of Warsaw, Poland, 2003.
- [2] Aksit, M., K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, *Abstracting Object Interactions Using Composition Filters*, in: R. Guerraoui, O. Nierstrasz and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS **791** (1994), pp. 152–184.
- [3] Anderson, E. P. and T. Reenskaug, *System design by composing structures of interacting objects*, in: *Proceedings of the European Conference on Object-Oriented Programming*, LNCS **615** (1992), pp. 133–153.
- [4] Blay-Fornarino, M., A. Charfi, D. Emsellem, A.-M. Pinna-Dery and M. Riveill, *Software interaction*, *Journal of Object Technology* (2004).
- [5] Bosch, J., *Superimposition: A component adaptation technique*, *Information and Software Technology* **41** (1999), pp. 257–273.
- [6] Bruneton, E., T. Coupaye and J. Stefani, *Recursive and dynamic software composition with sharing*, in: *Proceedings of WCOP*, 2002.
- [7] Budau, V. and G. Bernard, *Synchronous/asynchronous switch for a dynamic choice of communication model in distributed systems*, in: *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, 2002.
- [8] Cardelli, L., *Type systems*, *ACM Computing Surveys* (1996), pp. 263–264.
- [9] ISO, *Open Distributed Processing Reference Model - parts 1-4*, International Standard Organization (1995), ISO 10746-1..4.
- [10] Medvidovic, N. and R. N. Taylor, *A classification and comparison framework for software architecture description languages*, *Software Engineering* **26** (1997), pp. 70–93.
- [11] Mehta, N. R., N. Medvidovic and S. Phadke, *Towards a taxonomy of software connectors*, in: *Proceedings of the 22th International Conference on Software Engineering*, Limerick, Ireland, 2000.
- [12] Mulet, P., J. Malenfant and P. Cointe, *Towards a methodology for explicit composition of metaobjects*, in: *Proceedings of OOPSLA '95*, Austin, Texas, 1995, pp. 316–330.

- [13] Occello, A. and A.-M. Dery-Pinna, *Safety of component adaptations: Elements of formalization*, Technical Report I3S/RR-2004-04-FR, I3S laboratory, Sophia-Antipolis, France (2004).
- [14] OMG, *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03.
- [15] OMG, *Unified Modeling Language Specification*, OMG TC Document formal/03-03-01 (2003).
- [16] Pawlak, R., 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*, LNCS **2192** (2001), pp. 1–24.
- [17] Peschanski, F., *A versatile event-based communication model for generic distributed interactions*, in: *ICDCS Workshops*, 2002, pp. 503–510.
- [18] Pinna-Dery, A., M. Blay-Fornarino, B. Arcier, L. Mule and S. Moisan, *Distributed access knowledge-based system: Reified interaction service for trace and control*, in: *Proceedings of the 3rd International Symposium on Distributed Object Applications (DOA '01)*, Rome, Italy, 2001, pp. 76–84.
- [19] Plasil, F., D. Balek and R. Janecek, *SOFA/DCUP: Architecture for component trading and dynamic updating*, in: *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998.
- [20] Richters, M. and M. Gogolla, *Validating UML models and OCL constraints*, in: A. Evans and S. Kent, editors, *Proceedings of 3rd Int. Conf. on the Unified Modeling Language*, LNCS **1939** (2000), pp. 265–277.
- [21] Riehle, D. and T. Gross, *Role model based framework design and integration*, in: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1998, ACM Press.
- [22] Roman, E., S. W. Ambler and T. Jewell, “Mastering Enterprise Java Beans II and the Java 2 Platform,” John-Wiley & Sons Inc., 2002.
- [23] Segarra, M. and F. André, *A framework for dynamic adaptation in wireless environments*, in: *Proceedings of TOOLS Europe 2000*, Mont St. Michel, St. Malo, France, 2000.
- [24] Warmer, J. and A. Kleppe, *OCL: The constraint language of the UML*, Journal of Object-Oriented Programming (1999).