

Software interactions

Mireille Blay-Fornarino, Anis Charfi, David Emsellem, Anne-Marie Pinna-Dery, Michel Riveill, Laboratoire I3S, Bâtiment ESSI, BP 145, 06903 Sophia Antipolis CEDEX, France

This paper proposes the usage of a dedicated Interaction Specification Language (ISL) to express interactions between software components in a component-based application. This approach brings three major benefits: First, it allows component interactions to be expressed explicitly as first-class entities. Second, it enables the expression of the interactions independently of any specific programming languages or component models. This is especially important if we consider the variety of components specifications and their heterogeneity. Third, our approach permits the dynamic adaptation of the application by defining/removing interactions at runtime.

To this end, Interaction patterns are specified in ISL. They represent models of future interactions that connect some component instances. An Interaction Server is in charge of managing the life cycle of interactions (pattern registration, instantiation, destruction, interaction merging). The Interaction service allows the creation of interactions connecting heterogeneous components. Noah is an implementation of the Interaction Service. It can be thought of as a dynamic aspect repository with a weaver that uses a commutative and associative aspect composition mechanism.

1 INTRODUCTION

The goal of the interaction service is to enable the dynamic adaptation of component-based applications. This service is based on the interaction model. In this model, interactions are described in a meta-language independently of the component implementation language. Component interactions are the basis for application connectivity. Interaction patterns define one or more interaction rules. They can be registered on a specific server and then instantiated on component instances. This approach allows to dynamically establish links between components in order to adapt their behaviour to the environment. This is reached by creating/destroying instances of an interaction pattern at runtime. In the following, we explain what interactions are, then we introduce the interaction model through an example and after that we describe the ISL language for interaction pattern definition. Finally, we will discuss the merging mechanism which is required when at least two overlapping interactions are applied to the same component instance. The merging mechanism is based on the ISL language and it ensures commutativity and transitivity. This is very important to insure the consistent adaptation of an application by several users.

2 INTERACTIONS

”The architecture of a software system defines that system in terms of components and interactions among those components” [6].

Interactions can be found almost everywhere in the real world. We meet interactions more or less clearly expressed along the software lifecycle.

From Components to Interactions

A component is defined at least by the specification of its provided services and required services. The execution of a component-based application can be seen as reacting to messages (probably events) sent by some component instances to other component instances. The behaviour of a component instance is characterized by the observable external semantics of its methods when it receives a message. This semantics is expressed by the methods return values and the messages sent to other component instances. From our point of view, adapting the component instances can be attained by modifying their behaviours in order for example to throw exceptions, send new messages, change the return values, etc.

Now, let us define what is an interaction in our terminology. An interaction between instances of components specifies how the behaviour of these instances should change so that the interaction semantics is enforced. Thus, a notification interaction between an Agenda component and a Display component can modify the behaviour of the Agenda in such a way that the Display shows an appropriate message, whenever a new meeting is added or removed from the Agenda.

An interaction pattern is an abstraction of the interaction concept. It is defined on the component class level and can be instantiated. It is the counterpart to classes in object-oriented languages whereas interaction instances are the counterpart to objects.

Interactions in the Analysis Phase

During the analysis phase in the software lifecycle, interactions appear at the modeling level. This can be observed on several UML [1] diagrams. When the presence of interactions has a structural impact, they are expressed by associations in the class diagrams. Some interactions have an additional behavioural impact. Therefore, they appear as events and conditions in the state diagrams. However, in UML it is very difficult to express new interactions that may appear at execution time such as the availability or absence of some components in the execution environment.



Interactions and Software Architecture

Interactions also appear in the context of Architecture Description Languages (ADL). The fundamental concepts of ADL are components and connectors assembled with help of configurations [12]. A component is specified within ADL by the provided/required services. These services are expressed like method signatures, messages or variables.

”Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems.” [12]

ADL configuration languages use the architecture description in order to partially generate the component’s implementation or to improve the application deployment. The newest configuration language is the CORBA Component Assembly Descriptor [17], which automates deployment. These descriptors characterize key component deployment information, such as assembly instructions and interconnection topology. Some of the configuration languages have a limited dynamic dimension that enables us to specify possible evolutions of the application as stated by Medvidodic:

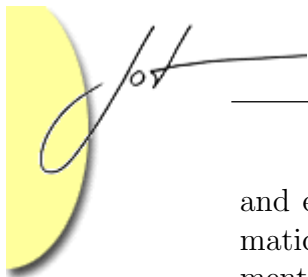
”Explicit modelling of architectures is intended to support development and evolution of large and potentially long-runtime systems. It may be necessary to evolve such systems during execution. Configurations exhibit dynamism by allowing replication, insertion, removal and reconnection of architectural elements or sub-architectures.”

The objective of ADL languages is to describe the communication between the required services and provided services at different stages of the software lifecycle. None of them is explicitly geared to the dynamic management of interactions.

Interactions and Component Models

Standard component platforms such as EJB [9], CCM [14] or .NET [16] compel the developer to define interactions a priori. This is usually done at the level of component interface or business logic. In CCM, components can interact with external entities, such as the services provided by the ORB, other components or clients via a set of interfaces called ports, which define the standard mechanisms to modify the component configurations.

The OMG IDL (Interface Definition Language) has been extended to express component interconnections. A component can offer multiple interfaces, each one defining a particular point of view to interact with the component. The four kinds of component interfaces in CCM are facets, receptacles, event sources/sinks and attributes. They can be used for component configuration. [17]. Two interaction modes are provided: facets for synchronous invocations, and event sinks for asynchronous notifications. Moreover, a component can define its required interfaces, which define how the component interacts with others: receptacles for synchronous invocations,



and event sources for asynchronous notifications. Components are installed automatically when they are installed on a component server. The port mechanisms mentioned above provide interfaces to configure the components i.e. set up the object connections, subscribe/publish events, etc. It is also possible to use the container programming model to implement interactions. The Container API simplifies the task of developing and configuring CORBA applications by providing an adaptation layer for commonly used services such as Transaction, Notification, Persistence and Security. However, the application developer still needs to statically express the component configuration i.e., specify how the component reacts to a given event, which event pertains to which sink, etc. As a result, the connectivity and the control induced by interactions are always platform-dependent. Interaction patterns are scattered among the components and the entities needed for their deployment like stubs and proxies.

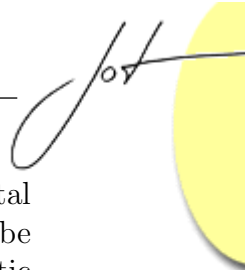
Interactions, AOP and Meta Programming

Aspect Oriented Programming

In an object-oriented application classes collaborate to achieve the application's goal. However, there are some concerns that cannot be viewed as being the responsibility of only one class, they cut across the class hierarchy and affect parts of many classes. Examples might be logging method calls, authorization or exception handling. Of course, the code that handles these parts can be added to each class separately, but that would violate the principle that each class has well-defined responsibilities. This is where Aspect Oriented Programming (AOP) [10] comes into play: AOP defines a new program construct, called aspect, which is used to capture crosscutting concerns in separate and well-modularized program entities. The application classes keep their well-defined responsibilities and each aspect captures cross-cutting behaviour.

The power of AOP lies in its ability to recognize patterns in pre-existing code and change them in multiple places (this property is called quantification) with minimal work from the programmer, without changing the source code of the original program (this property is called obliviousness). As an Aspect Oriented Programmer, you simply specify where (pointcut) you want crosscutting functionality (advice) to be executed. The weaver is responsible for integrating the aspect code with the base application. We differentiate static AOP approaches where aspect weaving is performed at compile-time and dynamic AOP approaches where weaving happens at runtime. Aspects can be applied to a number of classes, and can therefore add capability without forcing a class to extend or implement anything to obtain it.

Aspects in dynamic AOP languages are quite similar to interactions our interaction model but interactions are more powerful in explicitly modeling behavioural dependencies between components. In fact, aspects are basically language constructs to modularize crosscutting concerns. They are not appropriate to model the interactions between classes. In the following sections we will show a group interaction,



which can not be modeled as an aspect. Moreover, in AOP there is a fundamental problem when it comes to aspect composition. Aspect composition must always be handled by the programmer. In contrast, our interaction model provides automatic composition of interactions. Another advantage of the interaction model, is the use of ISL language that allows the definition of interactions independently of any programming language and across heterogeneous components. AOP however uses the same language for aspect and base code and therefore it does not support interoperability. In addition, the ISL language is more powerful and expressive than common aspect languages. In fact, ISL provides more operators such for concurrency and waiting, which are very useful in distributed applications. Shortly put, AOP is good at modularizing crosscutting concerns whilst interactions are good at modularizing component interactions.

Meta-Level Programming

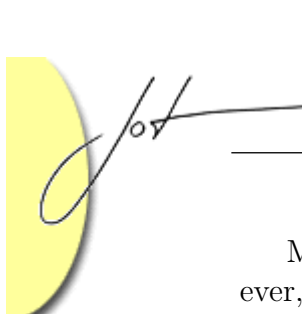
The technique of meta-programming goes back to dynamic languages like CLOS and Smalltalk. Meta-level programming techniques are key technologies to develop adaptive and adaptable software systems.

- The Meta Program is a program that manipulates other programs or itself.
- The Meta Object Protocol (MOP) defines the interface between the meta level and the base level.

AOP has a lot in common with Meta Programming [3]. Both capture crosscutting aspects of a software system in clean, controlled ways. One of the most fundamental properties of meta-level programming is that the programmer has access to the structures that represent a program, i.e. a program written in a specific language is represented at runtime in this very same language. The most popular language that implements meta-level programming concepts is CLOS, the Common Lisp Object System [3]. Its implementations are based on the MOP. The MOP can be seen as a standard interface to the CLOS interpreter. With the help of the MOP it is possible to modify the behaviour of the interpreter in a controlled way. This can be used to dynamically adapt component's behaviour to its changing environment.

CLOS provides a feature called method combination. For every method it is possible to define a method that is executed immediately before the primary method is executed (called the method's before-method), and a method that is executed after the primary method (the after-method). These methods can also be defined in subclasses.

Central to CLOS' Meta Object Protocol is the concept of meta class. The meta class is responsible for implementing the class's protocol e.g., the method call mechanism, the object creation process, etc. Each class in a system has its own meta class. Meta classes can be subclassed to create custom behaviour just like any other class.



Meta-level programming can be used for dynamic component adaptation. However, it is a quite complex approach that requires expert knowledge. It is in general complexer and slower than the AOP approaches. Moreover, we think that interactions between components should be expressed at the application level rather than at the meta level. Expressing interactions at the meta level in the form of message reception/sending is not intuitive.

3 THE INTERACTION MODEL

The interaction model allows the expression and definition of interactions between components as first-class entities. It also enables the adaptation of the application's behaviour according to the active interactions within the system. The interaction model fulfills the following requirements:

- Avoid inconsistencies that may be entailed by many adaptations
- Manage the composition of interactions automatically
- Interoperability of interactions across heterogeneous components
- Enable direct communication between the interacting components without a centralized interaction management point

Interaction properties

Interactions are the basic elements in the interaction model with the following properties:

- An interaction pattern defines the behavioural dependencies between the component classes that it connects. The interaction instances of this interaction pattern preserve this coherence locally.
- An interaction pattern is implementation-independent and an interaction instance can connect heterogeneous components across different platforms. For instance, an interaction can connect a Java component with a .NET component.
- Only the component interface (in particular the provided services) may be used to describe an interaction pattern.
- Interactions can not control properties that do not belong to the component's interface. Thus, encapsulation is not broken. The interface of an interaction-bound component is not modified even though its behaviour is modified.



- Interactions and interaction patterns can be dynamically created and destroyed at application runtime.
- The interaction management is based on a composition mechanism that ensures commutativity and associativity.

Implementation of the Interaction Service

In order to support these properties, we provided an implementation called "Noah" of the interaction model (available on the website <http://noah.essi.fr>). This implementation offers an interaction service and consists of the following parts:

The Interaction Server The interaction server manages interaction patterns and interaction instances. It enables the dynamic definition/removal of component interactions by instantiating interaction patterns. It also provides methods to traverse the interaction graph. Moreover, it acts as the central repository for interaction patterns. By Noah Server we refer to the Java Interaction Server. Noah Server is implemented as a Java RMI Server but it is also exposed as web service to be accessible from other platforms such as .NET.

Interacting Components These are components that have been prepared to manage and interpret interactions. We use code instrumentation at load time or compile time to modify the component class files. We provide instrumentation tools for Java based components such as EJB and RMI and for .NET components (local components and remote components published using the HTTP or TCP channels). The behaviour of interacting components can be dynamically modified by adding/removing new interactions on them. So, an interacting component is a dynamically adaptable component. To make a component interacting the programmer can use the tools JavaGenInt for Java and MSILGenInt for .NET.

Use Case

In order to illustrate the interaction model and its implementation, we take a simple agenda application as an example. This application is made up of several component classes: a Display component class which displays messages, a Security component class which authorizes method calls on a given component and an Agenda component class which stores meetings.

At runtime the component instances will be connected/disconnected by interactions. We define an interaction pattern team between two Agenda components and a Display component, an interaction pattern notification between a Display component and an Agenda component and an interaction pattern security that associates

an Agenda component with a security component. These patterns are shown in the listing below.

The creation of instances of these interaction patterns leads to changes in the component behaviour. Thus, the security component checks each method call on the Agenda instance *MichelAgenda* before this latter responds to that call. Only authorized method calls are processed by the Agenda component. We notice thereby a change in the behaviour of the Agenda: the base behaviour was to execute all method calls, now it executes only calls authorized by the security manager.

The interaction patterns security and notification address so-called non-functional concerns, whereas the interaction pattern team addresses the functional part or business logic. The interaction security can be implemented as an aspect in AOP whereas the interaction team can not be modelled as an aspect.

Figure 1 shows one possible graph of components and interactions during the execution of the agenda application.

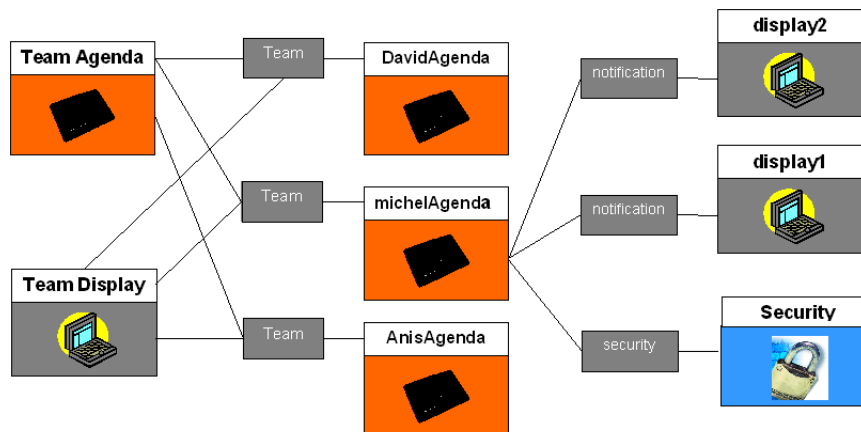
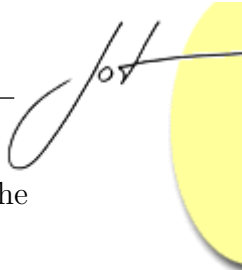


Figure 1: Interaction between components

Definition of Interaction Patterns

In the agenda example, the programmer can define an interaction pattern that associates an agenda component instance with a display component instance at runtime, so that an appropriate message is displayed whenever a meeting is added or removed from the agenda. The programmer defines interaction patterns and deliver them together with the application. The interaction patterns are adaptation models that the final user can apply to the application components at runtime in order to reconfigure the application. The interaction patterns are defined at the level of component classes whereas the interactions instances (or shortly interactions) affect instances of the component classes. The final user can in his turn create additional interaction patterns e.g., he can define an interaction pattern persistence



to connect an Agenda component to a database component in order to store the agenda meetings in the database.

Interaction patterns are specified in the Interaction Specification Language (ISL). This language is described in the next section. An interaction pattern defines at least one interaction rule. Interaction rules express the control that should be executed on the connected components. An interaction rule consists of two parts: the left side is the notifying message and the right side is the action. The semantics of an interaction rule is to rewrite method code i.e., instead of executing the default method (default behaviour), the interaction runtime should execute the rule's action. This applies to all component methods that match with the rule notifying message. "Match" means in this context, the same component class and the same method signature. The following listing shows the interaction patterns we mentioned so far.

```
interaction notification( Object obj, Display display) {
    obj.* -> obj._call // display.notify(_call)
}
```

```
interaction team( Agenda group, Agenda member, Display display) {
    group.addMeeting(String title) ->
        group._call ; member.addMeeting(title)
    member.addMeeting(String title) ->
        member._call; display.notify(_call)
}
```

```
interaction security(Object obj, SecurityService security) {
    obj.* -> if security.check(_call)
        then obj._call
        else exception "unauthorized user";
    endif
}
```

```
interaction persistence( Agenda agenda, Database database) {
    agenda.addMeeting(String title)-> agenda._call;
    database.store(agenda.getOwner(),agenda)
}
```

The interaction pattern notification can bind any Java component to a Display component. It declares only one interaction rule expressing that every message received by the component `obj` should be executed by `obj` and sent to the Display component concurrently.

The ISL keyword `_call` denotes the notifying message call (`obj._call`). It also represents the reified notifying message when it is used alone (`_call`) as a method parameter. The reified notifying message is an object that encapsulates the notifying method call and the call parameters.

The interaction pattern team can bind three components. It defines two interaction rules. The first interaction rule in this pattern states that the notifying message `addMeeting` to the Agenda instance group results in executing the message by the Agenda instance group itself and also by the Agenda instance member. This interaction pattern defines a collaboration relationship among the Agenda instances.

Once defined, the interaction pattern has to be registered on the Interaction Server. This latter provides a method `registerPattern(String islPattern)` that takes an interaction pattern as parameter. The Interaction Server acts as a repository for interaction patterns. Interaction patterns can be retrieved or modified with the method `getPatternCode(String patternName)`. We have also developed a graphical tool called Noah Editor, which facilitates writing interaction patterns and visualizes interaction instances and interacting components.

Creating and removing interactions

At runtime the programmer can bind or unbind component instances using one of the interaction patterns registered on the interaction server. For this purpose, the Server provides the method `instantiatePattern(String patternName, NoahProxy targets[])`.

The interaction server creates an RMI interaction object that represents the interaction, stores it and then passes it to the involved component instances. These components need to take into account the interaction rules expressed by the interaction object. When an interaction is added on a component instance, it first checks if any interaction rule with the same notifying message was already applied to it. If that is the case, it merges the new rule with the existing one. The merging process will be explained later, It generates one interaction rule which is semantically equivalent to the input rules. From the next notifying call on, the component's behaviour is changed. For example, if we successively instantiate the interaction team on the Agenda group and the Agenda instances David, Michel and Anis the merging mechanism will generate the rule shown below for the notifying message "group.addMeeting".

```
groupAgenda.addmeeting(String _var0) -> groupAgenda._call;
```



```
    davidAgenda.addMeeting(_var0)
    // michelAgenda.addMeeting1(_var0)
    // AnisAgenda.addMeeting(_var0)
```

The resulting rule means that each time a new meeting is added to the group agenda, the meeting must also be added to the agendas of the team members. The “;” is the notation of the ISL sequential operator while the symbol “//” denotes the concurrency operator. Accordingly, the actions of adding the new meeting (var_0) to the member agendas are performed in parallel.

Since interacting components are not necessarily Java components, special proxies are needed. The proxies are java objects that provide an identical interface to the interaction server. In fact, even in the Java world components may be quite different e.g., Enterprise Java Beans, local Java objects or remote Java objects etc. We can also have interactions that involve .NET components as well as Java components. For this reason, the method `instantiatePattern(String pattern, NoahProxy[] objects)` takes an array of proxies as second parameter. This proxy design pattern abstracts away from the technical component properties such as implementation language, platform, etc.

Alternatively, new interactions can be created with the help of the tool Noah Editor. The user first selects the registered interaction pattern he wants to instantiate (e.g. notification). A window pops up showing the types of components that can be connected by this pattern and all known running component instances of these types. After that, the user chooses the specific instances that should be connected by the interaction and the tool requests the interaction server to create the corresponding interaction objects.

The screenshot in Figure 2 shows Noah Editor. On the upper left corner, all registered interaction patterns are listed. The user can input a new interaction pattern. When the user selects one of the listed patterns, the tool shows the interacting components that are bound by this pattern in the “Registered object” window. The concrete interaction objects are RMI objects and they are shown in the window “Object’s interactions”. The tool also displays the business methods of an interacting component and the interaction rules that concern each of them. Business methods are those methods that realize the object’s business logic. Only these methods are relevant to interactions i.e., they may appear on the left side of an interaction rule.

When the user selects the business method `addMeeting(string meeting)` of the Agenda instance David, the Noah Editor displays the ISL interaction rule that pertain to the notifying message `addMeeting` on the component instance David.

To delete an interaction, the Interaction Server provides the method `removeInteraction(String identifier)`. All connected component instances (present on the left side of the interaction rule) will be requested to adapt their behaviour after the interaction (identified by the String identifier) has been destroyed. The destruction of an interaction can be performed with Noah Editor as well.

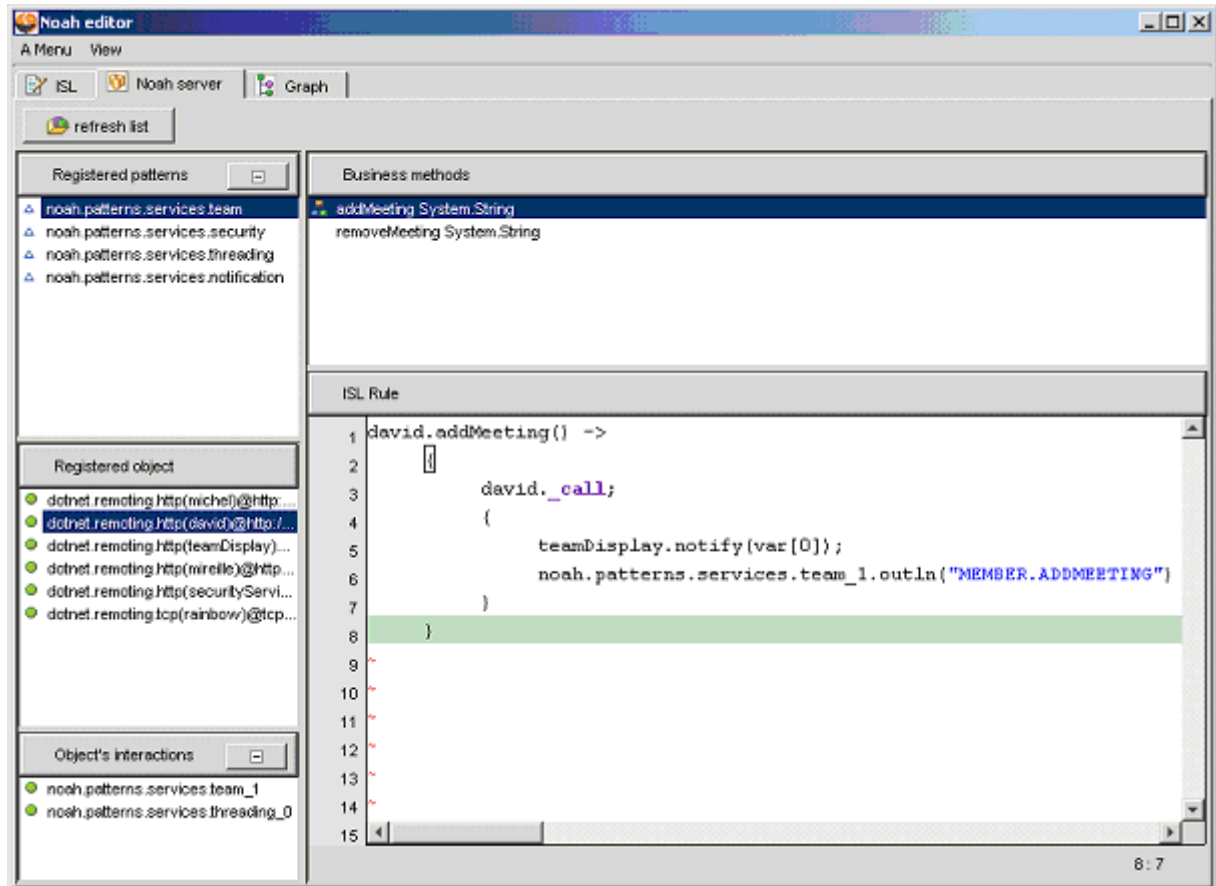


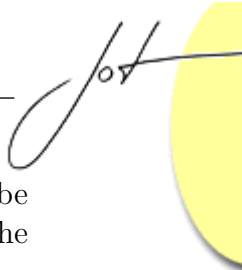
Figure 2: Screenshot of Noah Editor

Interacting components

Components that appear on the left side (notifying message) of an interaction rule must necessarily have been made interacting components. However, on the right side (action) of an interaction rule, every component (even non-interacting) may appear. Interacting components fulfil these requirements:

- they can switch the main execution thread to a local control after the reception of a notifying message.
- they are able to dynamically merge and unmerge interaction rules.
- they can send messages directly to the other connected interacting components without going through the interaction server.

The execution of messages (method calls) within interacting components is quite different from the ordinary method execution. When an interacting component receives a message that turns out to be a notifying message, the interaction rule



that is associated with that message is evaluated locally. This evaluation can be thought of as interpreting the rule's action. During this evaluation, calls among the involved components are direct and do not pass through the interaction server.

Several tools are shipped with the interaction server. Code Instrumentation and Code Engineering techniques are the base upon which these tools perform class modification. The classes are modified, so that they can manage interaction rules (adding, removing and merging). The tool JavaGenInt handles this task for local and RMI Java components. It uses the Byte Code Engineering Library (BCEL)[5].

For Enterprise Java Beans the proxies take charge of interactions in a similar way to standard services like synchronisation, consistency and persistence. In JOnAs [4] the proxies are generated by the GenIC code generation tool. We modified the tool so that the generated proxies can manage interactions. To make an EJB interacting, the developer sets the attribute "value" of the element "jonas-interaction" to "true". This setting is specified in the deployment descriptor of the EJB application. The XML DTD of the EJB deployment descriptor has been extended appropriately.

The interaction service also supports .NET components. The MSILGenInt tool takes a .NET assembly (.dll or .exe file) as input and makes the classes of the assembly 'interacting'. MSIL GenInt performs code engineering at the intermediate language level. MSIL stands for Microsoft Intermediate Language, which is similar to Java bytecode.

For all interacting components, the overhead of the interaction mechanism is the cost of a testing instruction when no interaction rule is applied to the notifying message. In the other case, the message is executed using the dynamic invocation technique provided by the Reflection API of Java or .NET.

Interoperability and Interactions

One of the encountered problems relates to the communication between components from different platforms. The interacting components may be local objects, Enterprise Java Beans, or remote objects in java or .NET. We need to handle these components in a uniform way. This is required by the interaction server when it passes interaction objects to the interacting components the components. Note that these objects are serialized into an XML representation of XML. Furthermore, direct inter-component communication also rises the same requirement. For this reason, we should find a way to handle the components uniformly and minimize differences between the various implementations. In fact, ISL parsing, ISL tree management and rule merging are common elements among the different implementations.

Our approach is to encapsulate a reference to the interacting component in a NoahProxy object that manages message sending and provides an identical interface to all interacting components. This is fully transparent to the user.

4 THE INTERACTION SPECIFICATION LANGUAGE (ISL)

The ISL language is used to specify interaction patterns independently of the application programming language. It includes several operators such as the conditional operator (if ... then ... else ... endif), the sequential operator (;), the concurrency operator (//), the waiting operator, and the exception handling operator. The keyword 'this' within an interaction pattern refers to the interaction object itself. ISL recursively defines the component behaviour with the operators described below (each behaviour describes a behavioural class). The term behaviour denotes the interaction rule's action (right side). A detailed description of the semantics of ISL operators and constructs can be found in [2]. We shortly discuss some of them:

- The method invocation operator "." denotes a method call on the receiving component. Using the keyword `_call` in place of a method call refers to the notifying method call.
- The assignment operator " := " assigns the return value of a message sending behaviour (method call) or of an assignment to a variable.
- The sequence operator ";" states that two behaviours should be executed one after the other.
- The concurrency operator "//" states that two behaviours should be executed in parallel.
- The waiting operator (" _X" where X is a label) states that the execution of a message, variable assignment or another waiting behaviour is blocked, until the end of the execution of a behaviour labelled by "[X]".
- The conditional operator (if then else endif) states a conditional execution of a behaviour depending on the boolean result of the execution of another behaviour.
- The exception handling operator (try ... catch) permits throwing exceptions. It can only be used to express that the execution of a notifying message has been rejected. The exception thrown is then returned to the user (if it is not caught in the interaction rule).
- The delegation operator states that an action that does not contain the triggering message of the current rule should be considered as the triggering message. There is no keyword to denote this operator. It is implicitly added during the phase of semantic analysis.
- The wild card operator * matches all messages on the receiving component. It can be only used after the method invocation operator. This operator is used in the security pattern, so that all method calls on the Agenda are notifying.



- The ISL keyword `_call` represents the notifying message call. It also represents the reified notifying message when it is used alone as a method parameter.

5 RULE MERGING

The definition of interaction pattern follows the principle of Separation of Concerns. Creating interactions occurs dynamically, on behalf of several users having different point of view of the application (local view). This leads to a separation of interactions as every user expresses controls and behaviour modifications regardless of the others. Consequently, the interaction model should enforce the overall coherence of the component adaptations (global view).

Thus, when more than one interaction is simultaneously applied to the same notifying message of a component, merging interaction rules becomes necessary. Rule merging is dynamically managed by the interacting component itself each time a rule is added to or removed. The merging mechanism is specified through a finite set of merging rules and equivalence axioms based on the ISL operators. These rules and axioms are described in more detail in [2]. The rule merging fulfills the following properties:

- The coherence of interactions rules: in particular rule merging is rejected if it entails a non deterministic behaviour. Moreover, rule merging should not provoke any non-explicit waiting especially when concurrency and sequence operators are involved.
- Rule merging is commutative: the order in which the rules have been added to the interacting components does not affect the behaviour of the system. The resulting behaviour is equivalent for all sequences of interactions. In fact, if we first instantiate the interaction pattern notification and then the interaction pattern security, we would intuitively expect the same behaviour as if we instantiated both patterns the other way around.
- Rule merging is also associative. If we merge the interaction pattern team with the interaction pattern notification, then take the resulting rule and merge it with the interaction pattern persistence, we will get the same result as if we firstly merge the interactions persistence and notification together and then add the interaction team.
- Two rules that throw exceptions can not be merged because if an exception is thrown at runtime we do not know which rule has raised it. We say that raising exceptions is absorbing for the merging mechanism.
- Merging the sequential operator with the concurrency operators should not induce any chronological order that does not emanate from the input rules.

When the merging may introduce an order between the actions, the waiting operator is used. The rationale for this is the following equivalence:
 $m; p[X] \equiv m//p_{-}\{X\}$

When rule merging is possible, it generates one rule that will be executed instead of the merged rules. The resulting rule has the same semantics as the merged rules. Figure 3 explains how an interacting component manages interaction rules and how it performs rule merging.

An interacting component has an array of rules for each business method. At the first position (zero) of each array, the result of merging all rules that affect that business method is placed. If an interaction rule is added to a component, it will be stored in the next free slot in the respective array. Then, it will be merged with the rule in the first slot of that array. The result of rule merging is again placed in the first position. In Figure 3 we see the rule array of the Agenda instance Anis, which pertains to the method `addMeeting`. The merging of the two interactions notification and security (both have the same notifying message obj.*) generates one rule which is stored in the first slot of the array.

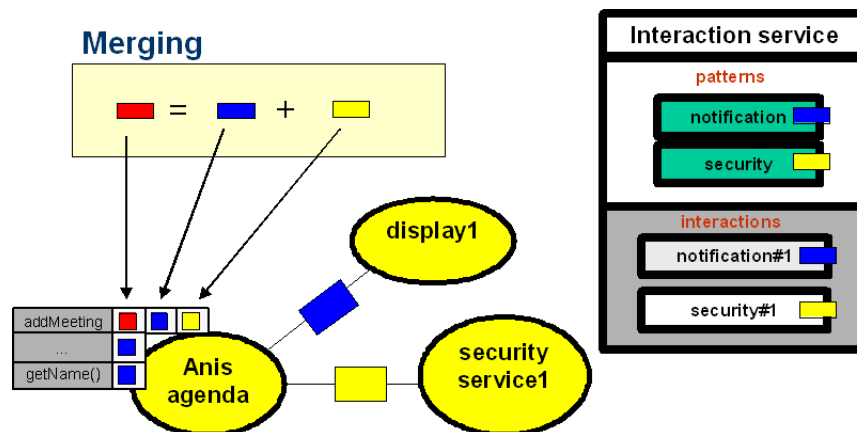
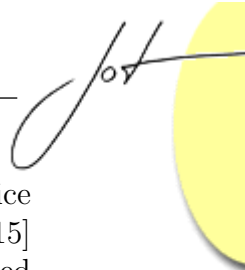


Figure 3: Merging rule

6 CONCLUSION

The interaction model allows the dynamic adaptation of components with help of interaction patterns and interaction instances. The user expresses interaction patterns at the application level using the ISL language. The merging mechanism is essential to keep the global coherence and consistency of all interactions. It ensures commutativity and associativity.

Within the current implementation, it is possible to define interactions on Java-based components (local and RMI or EJB) or .NET-based components. The Noah



interaction server can be downloaded at <http://noah.essi.fr>. The interaction service has been used to dynamically manage data bases [7], to manipulate frameworks [15] and to integrate technical services [13]. Furthermore, a study is being conducted about managing the adaptation of nomadic applications with help of interactions.

The advantages of the interaction model over AOP approaches include the support of heterogeneous components, the powerful interaction language ISL with its composition mechanism, and the support for expressing behavioural dependencies between components at runtime. In fact, the primary focus of AOP languages such as AspectJ [11] is the modularization of crosscutting concerns. Like aspects, interactions are also crosscutting in their nature because they connect independent component classes. However in AOP, it is difficult to express multi-party interactions like of the interaction team. In Meta Programming, the programmer could express interactions in terms of meta-behaviours, which is complex for the normal programmer and takes him away from the application to higher meta levels. In both AOP and Meta Programming the composition of interactions is based on the explicit order of interactions, which is difficult to manage by the final users. The advantage of our approach is the automatic composition mechanism with clear and formal semantics.

With regard to services, the CORBA notification service [8] is somewhat close to the interaction service. Nevertheless, within the notification service the programmer has to implement the interactions by means of notifications and event management. In addition, the absence of interactions, as well-structured entities, makes the composition of interactions quite difficult to manage by the system.

Several works around the interaction model are currently in progress. We consider extending the syntax of the ISL language and defining additional operators. For example the operator return will allow giving up the control before an interaction rule is fully executed. We also consider placing interactions at other joint points besides method calls e.g., field access/write and constructor calls. In this way interactions could be as expressive as current AOP languages. The interaction service has also been used in some specific domains like Expert Systems: We provided a library of interaction patterns, which can be easily used by the final users of the expert system.

We also found out that the expression of generic interaction patterns is very useful. In fact, the same type of interactions may apply to different types of interacting components. The investigation of generic interaction pattern is in course and we are evaluating the various alternatives to integrate generic interactions to the interaction server Noah.

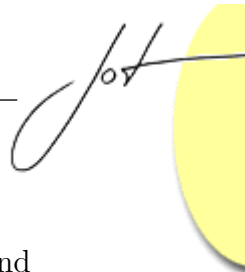
At present, the interaction model does not precise any user rights because it was originally conceived to allow collaborative programming. We are now working on a security model, which specifies the different programmer roles and the respective user rights such as create, destroy, put and remove interactions. A further thrust of research within the RAINBOW team focuses on Human Computer Interaction

(HCI). In this field, we examine how the interaction model can be used for HCI composition. With the interaction model, we can consider HCI like technical services of a business component (which contains only the application logical part). So, the HCI interface is just like the services security or persistence. In this vein, we can manage the dialog between UI and business components by means of interaction rules.

Interactions bring an additional abstraction layer and provide tools which permit controlling this layer. We also aim at offering more flexibility and more control over the application's adaptation by analyzing the interaction graph by developing administrative tools such as the interaction network viewer.

REFERENCES

- [1] S. S. Alhir. *UML in a Nutshell*. O'Reilly, 1998.
- [2] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO*. PhD thesis, Université de Nice-Sophia Antipolis, octobre 2001.
- [3] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification x3j13. In *SIGPLAN Notices (Special Issue)*, 23, 1988.
- [4] E. Cecchet and J. Marguerite. Jonas v2.4 tutorial. Technical report, Nice University and INRIA, 2002.
- [5] M. Dahm. Byte code engineering with the bcel api, 2001.
- [6] R. M. DeLine, D. Klein, T. Ross, D. Toung, and G. Zelesnik. Abstraction for software architecture and tools to support them. *IEEE Trans. Software Engineering*, 21(4):314–335, April 1995.
- [7] Moisan S. Dery A.M., Blay-Fornarino. Distributed access knowledge-based system: Reified interaction service for trace and control. *3rd International Symposium on Distributed Object Applications (DOA 2001)*, September 2001.
- [8] Object Management Group. Notification service, omg document formal/00-06-20. Technical report, June 2000.
- [9] Sun Microsystem Inc. Enterprise javabeans specification. version 1.1, January 2000.
- [10] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.



- [11] Lamping J. Kiczales G. Aspectj homepage. Technical report, 2001.
- [12] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.
- [13] Anne-Marie Dery Michel Riveill Olivier Nano, Mireille Blay-Fornarino. An abstract model for integrating and composing services in component platforms. *Seventh International Workshop on Component-Oriented Programming (in conjunction with ECOOP'2002), Malaga, Spain, June 2002*.
- [14] R. Marvie R. and M-C. Pellignini. Modèles de composants, un état de l'art. *Numéro spécial de L'Objet*, 8(3), 2002.
- [15] P. Rapicault. *Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation*. PhD thesis, Université de Nice-Sophia Antipolis, May 2002.
- [16] Jeffrey Richter. *Applied Microsoft .Net Framework Programming*. Microsoft Press, 2002.
- [17] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the corba component model. Technical report, Whashington University in St Louis, 2000.

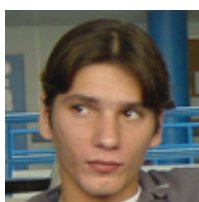
ABOUT THE AUTHORS



Mireille Blay-Fornarino is Assistant professor in CNRS/I3S laboratory, University of Nice. She can be reached at blay@essi.fr. See also <http://rainbow.essi.fr/blay>.



Anis Charfi Anis Charfi is a PhD student at the Darmstadt University of Technology. During his master thesis within the Rainbow team he implemented the interaction model in .NET. He can be reached at charfi@informatik.tu-darmstadt.de.



David Emsellem is research ingeneer in CNRS/I3S Laboratory, University of Nice. He can be reached at emsellem@essi.fr.



Anne-Marie Pinna-Dery is Assistant professor in CNRS/I3S laboratory, University of Nice. She can be reached at pinna@essi.fr. See also <http://rainbow.essi.fr/pinna>.



Michel Riveill is professor of computer science at the Université de Nice - Sophia Antipolis. He heads the Rainbow project at the Laboratoire I3S (<http://www.i3s.unice.fr>). Previously, he was successively Professor of Computer Science at Université de Savoie, Institut National Polytechnique de Grenoble since 1993. He can be reached at riveill@essi.fr. See also <http://rainbow.essi.fr/riveill>.