

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

SOFTWARE INTERACTIONS INTO THE SSCLI PLATFORM

Anis Charfi, David Emsellem, Michel Riveill

Projet RAINBOW

Rapport de recherche
ISRN I3S/RR-2004-03-FR

Janvier 2004

RÉSUMÉ :

MOTS CLÉS :

ABSTRACT:

By using an Interaction Specification Language (ISL), interactions between components can be expressed in a language independent way. At class level, interaction pattern specified in ISL represent models of future interactions when applied on some component instances. The Interaction Server is in charge of managing the life cycle of interactions (interaction pattern registration and instantiation, destruction of interactions, merging). It acts as a central repository that keeps the global coherency of the adaptations realized on the component instances. The Interaction service allows creating interactions between heterogeneous components. Noah is an implementation of this Interaction Service. It can be thought as a dynamic aspect repository with a weaver that uses an aspect composition mechanism that insures commutable and associative adaptations. In this paper, we propose the implementation of the Interaction Service in the SSCLI. In contrast to other implementations such as Java where interaction management represents an additional layer, SSCLI enables us to integrate Interaction Management as in intrinsic part of the CLI runtime.

KEY WORDS :

Software interactions, .Net, rotor

Software Interactions into the SSCLI platform

Anis Charfi

David Emsellem

Michel Riveill

Université de Nice – Sophia Antipolis

Ecole Supérieure en Sciences Informatiques

930 route des Colles – BP 145 – 06903 Sophia Antipolis Cedex – France

riveill@essi.fr

Abstract

By using an Interaction Specification Language (ISL), interactions between components can be expressed in a language independent way. At class level, interaction pattern specified in ISL represent models of future interactions when applied on some component instances.

The Interaction Server is in charge of managing the life cycle of interactions (interaction pattern registration and instantiation, destruction of interactions, merging). It acts as a central repository that keeps the global coherency of the adaptations realized on the component instances. The Interaction service allows creating interactions between heterogeneous components. Noah is an implementation of this Interaction Service. It can be thought as a dynamic aspect repository with a weaver that uses an aspect composition mechanism that insures commutable and associative adaptations.

In this paper, we propose the implementation of the Interaction Service in the SSCLI. In contrast to other implementations such as Java where interaction management represents an additional layer, SSCLI enables us to integrate Interaction Management as in intrinsic part of the CLI runtime.

1 Introduction

The Interaction Service allows the dynamic adaptation of component-based applications. It is based on an interaction model. In this model interactions are described in a meta-language, which is independent of the component implementation language. Interaction patterns are registered on a specific server and then instantiated on component instances. This approach allows to dynamically link components and to adapt their behaviour to their environment by using interaction instantiation and destruction at runtime. The user defines interactions at the application level. The merging mechanism assures commutability and transitivity when several interactions are applied to a

component. It is based on the **SL** language and insures a consistent adaptation of the application by several users. We next explain what interactions are, introduce the interaction service through an example and describe the ISL language for interaction pattern definition. Then, we discuss how we used the SSCLI to build a platform where interactions are integrated as an intrinsic part of the CLI runtime.

2 Interaction model

The interaction model [01, 02, 03] allows expressing interactions between components and it is responsible for carrying out the application adaptation according to the active interactions within the system. The model should fulfil the following requirements:

- Avoid inconsistencies that may be entailed by the adaptation
- Manage the interaction composition
- Insure interaction interoperability across heterogeneous components
- Enable direct communication between the interacting components. Central interaction management should not provoke a performance bottleneck.

1.1 Interaction properties

Interactions are basic atomic elements with the following properties:

- An interaction pattern defines the behavioural dependencies between the components it connects. The interaction instances of this pattern preserve this coherency locally.
- An interaction pattern is implementation-independent and an interaction instance can combine heterogeneous components across different platforms. For instance, a Java object may interact with a .NET object.
- Only the *component interface* may be used to describe an interaction pattern.
- Interactions cannot control properties that do not belong to the component interface. Thus encapsulation is not broken. The interface of an interaction-bound object is not modified even though its behaviour (concretely behaviour is the execution of a method) is modified.
- Interactions and interaction patterns can be dynamically created and destroyed during the application execution.
- The component interactions management is based on a composition mechanism that ensures commutability and associability.

1.2 The Java implementation of the interaction service

In order to support all these properties we implemented an interaction service called “Noah” (available on the website <http://rainbow.essi.fr/noah>). It consists of the following parts:

- The interaction server: it enables dynamic interaction pattern definition, makes it possible to bind and unbind objects by instantiating and removing *interaction patterns* and provides methods to traverse the interaction graph. By *Noah Server* or simply *Noah* we refer to the Java Interaction server. The *Noah server* is implemented as a Java RMI Server.
- Interacting components: these are components that have been prepared (through classfile instrumentation) to interact with other components. Their behaviour can be dynamically modified. So, an interacting component is a dynamically adaptable component. We use both terms *interacting object* and *interacting component* to denote a component that can deal with interactions.

1.3 A 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 components: A *Display* component which displays messages, a *Security* component which authorizes or not method calls on a given component and an *Agenda* component which stores meetings.

At runtime the component instances will be bound and unbound by interactions. We define a *team* interaction between two Agenda instances and a Display, a *notification* interaction between a Display and an Agenda and a *security* interaction that associates an Agenda with an authentication component. The *figure 1* shows a possible Graph of components and interactions during the execution of the application.

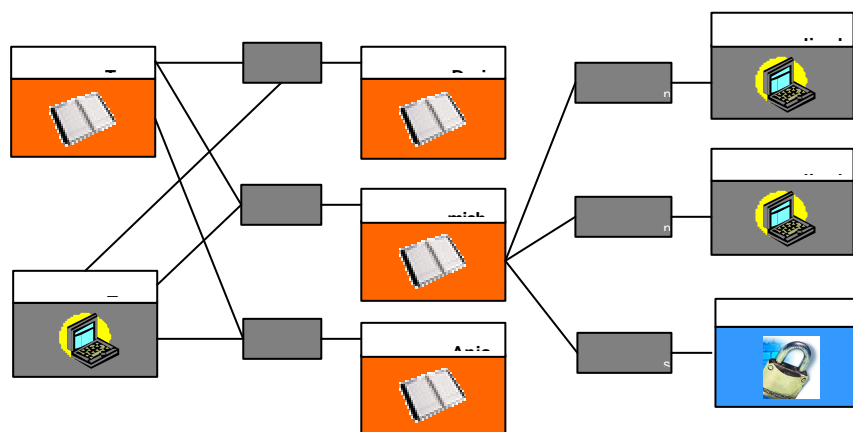


Figure 1: The interaction graph of an agenda application

Creating instances of the interactions mentioned above leads to new component behaviour. Thus, the security component checks each method call on the *Agenda* instance *micHELAgenda* before this latter responds to that method call. Only *authorized* method call are processed by the *Agenda* component instance.

1.4 Interaction pattern definition

In the agenda example, the application developer might enable the users to associate an agenda with a display at runtime, so that the users are notified whenever a new meeting is added to the agenda. Concretely, he defines interaction patterns and ships them together with the application. The interaction patterns are interdependence models that the final user can apply to the application components at runtime. The final user can in his turn create additional interaction patterns e.g. he can define a persistence pattern binding an Agenda to a database component, which results in persisting the agenda meetings to the database.

Interaction patterns are specified in the *Interaction Specification Language* (ISL). This language provides a small set of operators (sequential, concurrency, conditional, assignment, exception, invocation, etc). 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 *reaction*. The semantic of an interaction rule is to rewrite method code. That is, instead of executing the default method (what we call the *default behaviour*), the interaction runtime should execute the actions specified in the rule *reaction*. This applies to all component methods can be unified with the rule *notifying message*. The following listing shows the interaction patterns mentioned above.

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

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(_reifiedCall)
}
```

```

interaction security(Object O, SecurityService security)
{
    O.* ->    if (security.check(_reifiedCall)){
                then obj._call
            } else {
                throw "unauthorized user";
            }
}

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 object to a *Display* instance. It contains only one interaction rule expressing the fact that every message received by the object *obj* should be executed by *obj* and concurrently sent to the *Display* component.

The ISL keyword *_call* represents 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 as well as 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. Moreover, the meeting is added to the *Agenda* instance *member*. This interaction pattern defines a *collaboration*¹ relationship among the *Agenda* instances.

Once defined, the interaction pattern has to be registered on the *Interaction Server*. This latter provides a *registerPattern(String islPattern)* method that takes an interaction pattern as parameter. The *Interaction Server* acts as the central pattern repository.

1.5 Creating and removing interactions

At runtime the programmer can bind or unbind component instances together using one of the interaction patterns registered on the interaction server. The *Server* provides the

¹ Collaboration occurs every time two or more objects interact. It can be as simple as one object sending a message to another object. Or it can be as complex as dozens of objects exchanging messages.

method *instantiatePattern(String patternName, NoahProxy targets[])* throws *Exception* to create new interactions.

The interaction server creates an interaction object that represents the interaction, stores it and then requests the involved objects to take into account the interaction rules expressed by the instantiated interaction. If some interaction rules with the same notifying were already applied to those objects, they have to merge the new rules with the other ones. The merging process generates one interaction rule which is semantically equivalent to the merging input rules. From the next notifying method call on, the component behaviour is adapted. The merging process is described later.

1.6 Rule merging

Interaction pattern definition involves the principle of *separation of concerns*. Creating interactions occurs dynamically, on behalf of several users having different point of view of the application “local”. 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 coherency (“global”) of the components.

Thus, when more than one interaction is simultaneously applied to the same notifying message of a component², merging interaction rules becomes necessary. In the case of Java-based *interacting components*, rule merging is dynamically managed each time a rule is added to or removed from a component instance.

The merging mechanism is specified through a finite set of merging rules based on the ISL operators and a finite set of equivalence axioms. These rules and axioms are described in more detail in [01]. The *rule merging* fulfils the following properties:

The coherency of the overall interactions rules: in particular, the fusion of rules that may entail a non-deterministic behaviour is rejected. Moreover, it does not provoke any non-explicit waiting especially when concurrency and sequence operators are merged together.

Rule merging is *commutative*: the order in which the rules have been added to the interacting object does not affect the behaviour of the system. The resulting behaviours are equivalent for all sequences. In fact, if we first instantiate the *notification* pattern and then the *security* pattern, 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 *team* interaction with the notification interaction, then take the resulting rule and merge it with the persistence interaction, we will get the same result as if we firstly merge the persistence and notification interactions together and at last add the team interaction.

² Rule merging takes place only for interaction rules whose notifying messages have the same signature (same method name, same arity, same parameter types, same return types) and concerning the same receiving object. In this case, rules are called unifiable.

3 Rotor implementation

The interaction service can be split in two major parts: the *interaction server* on the one hand and the *component management services* on the other hand. The interaction server, currently implemented in Java RMI, is used for all component architectures (J2EE, EJB and SSCLI) whereas the *component management services* are platform specific.

The *interaction server* acts as the central repository for interaction patterns. It provides methods for instantiation of interactions patterns and for interaction rule destruction. It also performs other important tasks such as rule merging. The interaction server should be accessible to the diverse component platforms. In the case of the SSCLI platform, components use a web service to access the interaction server [04].

The *component management services* encompass the execution of interaction rules (an execution engine performs this task), the management of interactions rules (adding, removing, and call redirection), and the inter-component communication (method invocation among components). The component management services are platform specific. The next part presents the implementation of the component management services for the SSCLI platform

2.1 Interaction handling as an intrinsic component property

The SSCLI platform gives an intimate access to the execution mechanism of a .NET application. It offers a means to the research community to experiment many different computer science aspects. In our research work, we use the SSCLI to build a real (fully integrated in the runtime) interaction platform. The interaction platform has the following characteristics:

- The behaviour of each component instance can potentially be modified by an interaction
- Interaction rules are runtime structures (like classes and objects) that can be executed by the CLI platform implementation
- The runtime implementation of the method invocation mechanism has knowledge of how to execute an interaction rule.
- The rotor version of the interaction service takes advantage of the deep knowledge of the runtime internals to bring some optimisations that can not be performed for a platform like Java. In Java the interaction layer can only be add on the top of the JVM at the bytecode level.

2.2 Object layout: an additional per-instance method table

Interactions allow the dynamic modification of the behaviour of a component on a per-instance basis. Each instance must know which kind of behaviour to execute in place of the default method implementation. Therefore, the component instance must somewhere store this information and retrieve it when needed. We store the behavioural modification inside the runtime object representation by means of an extra field just after the method table pointer.

When working with interactions, two instances of the same class may have different behaviours for the same method. Therefore, each instance must have its own method table in addition to the default method table shared among all class instances. In the following, we call this new method table the *Noah method table* to distinguish it from the class method table. The *Noah method table* is stored in the extra field mentioned before.

2.3 Method invocation with interaction handling

In order to understand how interaction rules can dynamically modify the method invocation mechanism, let us consider the example of an *Agenda* instance with a method *addMeeting*. When none of the *Agenda* instances is redefined by interactions, the Noah method table is *null*. The invocation of the method *addMeeting* is dispatched (like in the initial version of the SSCLI) through the corresponding virtual method table slot, the JMI thunk(s) and then the execution of the JIT compiled code [05]. Suppose now the method *addMeeting* is redefined by an interacting rule. The runtime has to reflect this change on the component. To achieve this, the extra field of the instance points to the newly created *Noah method table*, which contains as many slots as in the shared class virtual method table. The corresponding slot in the Noah virtual table points to a JIT compiled version of the interaction rule. The slot of the class virtual table is then marked as being "*interacting*" (this means that at least one instance of the class has redefined this method). This slot is also updated to point to the *MethodDesc prestub* in order to force JIT recompilation. When this method is invoked again, the jitter comes into play. When compiling the method, the jitter is aware of the "*interacting*" property, and therefore it adds a piece of code between the native default prologue and the user code to handle the method redirection to execute the interaction rule. This small piece of code tests if any interaction rule exists in the Noah method table slot. If so, it jumps to the JIT-compiled interaction rule. In the other case it jumps to the user default method code (see figure 2).

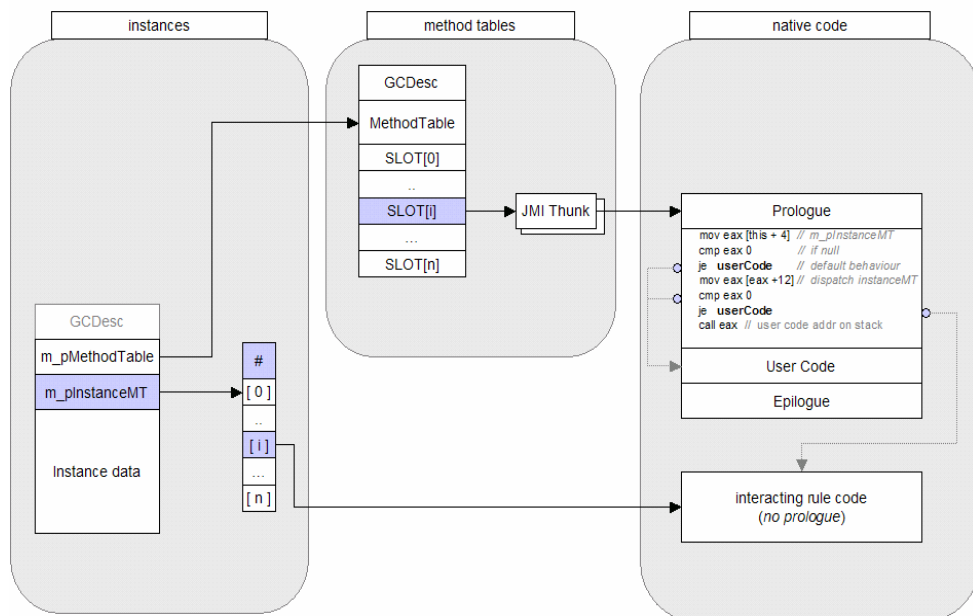


Figure 2: runtime data structure layout for interaction management

6 Concluding remarks

The interaction service enables the dynamic component adaptation. This is attained by defining interaction patterns and putting or removing interaction rules at runtime. Thus the user specifies the interactions in a natural way at the application level with ISL. The merging mechanism guarantees commutability and associability when new interaction rules with the same notifying message are added. With the initial Java implementation it was possible to have interactions on local, RMI or EJB Java objects. In these implementations of the interaction service, an execution engine interprets interaction rules.

The SSCLI offers a deep access to the CLI execution engine and the method invocation mechanism. We use this opportunity to implement a new version of the interaction service, which allows us to have some optimisation on critical points. Firstly, we reduced the memory overhead due to by rule-storage. We also decreased the time overhead caused by having a method wrapper for each method. The interpretation of interacting rules has also caused an important time overhead in the Java version. With the SSCLI version, we compiled the interaction rules into native code.

With the new SSCLI interaction platform we are able to manage interactions for Java components as well as CLI components, by using the same Java interaction server. Moreover with the Common Type System, we can handle automatically components written in any language implemented on the top of the SSCLI platform.

References

[01] *Laurent Berger*, Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO, PhD thesis, Université de Nice, 2001.

[02] *Anis Charfi*. Software interaction: Towards dynamic adaptation of .net objects. Master Science Thesis - Technical University of Munich, July 2003

[03] *David Emsellem, Anis Charfi, and Michel Riveill*. Dynamic component composition in .net. In *ECOOP'2003 Workshop on .NET: The Programmer's Perspective*, Darmstadt, Germany, 22 July 2003

[04] *Adam Freeman and Allen Jones*, Microsoft .NET XML Web Services Step by Step, Microsoft Press, October 2002.

[05] *David Stutz, Ted Neward, Geoff Shilling*, Shared Source CLI Essentials. O'Reilly, March 2003.