Interact ons between objects : an aspect of object-or ented languages

L. Berger
I3S - CNRS UPRESA 6070
Bât ESSI, 930 Rte des Colles
BP 145
06903 Sophia-Antipolis, France
04 92 96 50 66
berger@essi.fr

A.M. Dery
I3S - CNRS UPRESA 6070
Bât ESSI, 930 Rte des Colles
BP 145
06903 Sophia-Antipolis, France
04 92 96 51 62
pinna@essi.fr

M. Fornarino
I3S - CNRS UPRESA 6070
Bât ESSI, 930 Rte des Colles
BP 145
06903 Sophia-Antipolis, France
04 92 96 51 61
blay@essi.fr

ABSTRACT

This paper is based on our experience of integrating the interactions to several object-oriented languages and on our conclusion: interactions should be viewed as an "extensible" aspect of object-oriented languages.

KEYWORDS

aspects, object, meta-programming, interaction, synchronization

1 INTRODUCTION

It is now well known that it is not easy to manage the interactions between objects in conventional object-oriented languages [Rum92, Bos94]. The interactions are tangled in the code of the objects, specializing classes, sending messages to other objects in the code of methods or referencing interacting objects by specific attributes. The consequence is that the semantic of the objects participating to an interaction is modified and the application maintainability and extensibility are harder.

So different works have proposed some extensions to object-oriented languages to take into account this lack of abstraction [Neu91, Hol92, AF93, Pin93, Frø94]. We did the same [DFP95, DDF96, BDFJ97]. But, today, we argue that interactions are conceptually independent of any object-oriented language and match exactly the definition of an aspect described in [KLM+97] :

> ASPECTS tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways.

So, we propose a language, called IL (Interaction Language) to describe the interactions between objects. This language allows to specify interactions between objects referring to the objects only through their interface (list of methods). IL is an aspect language for object-oriented languages. So, whatever the object-oriented component language may be, classes and behaviors of objects are written in the component language without having to deal explicitly with interactions.

Our experience with integration of interactions to object-oriented languages has shown that the expressiveness required for the interactions can vary according to the component languages and the target applications. When the language integrates concurrent or distributed paradigms, the interaction expression implies to deal with concurrent behavior or distributed communication. When the applications need more static checking, an analysis or control of the global interaction graph is essential. In the first implementations, we defined a reflexive architecture in order to give the way to a programmer to adapt the tool to its specific applications. Today, we think that this approach lives a good way to allow the implementation and evolution of Aspect Weavers[1]. Although the aspect approach, when it is possible, gives a higher level of specification better adapted to the final developers. So, we propose to customize the interaction language both using directly aspects on the IL language and indirectly to adjust the Aspect Weavers to the programmer needs.

This paper is based on our experience of integrating interactions to different object-oriented languages and on our conclusion : interactions should be viewed as an "extensible" aspect of object-oriented languages. The paper is organized as follows. The next section provides a short description of IL and illustrates its integration in several environments. Section 3 presents our proposal solution as Interaction Aspect Weaver. This solution is essentially based upon meta programming to facilitate the future integration of new aspects. The last section presents some aspects to extend the basic IL such as a synchronization aspect to synchronize interactions in a concurrent environment. Then we conclude.

2 INTERACTIONS : THE IL LANGUAGE

This section briefly presents the IL language through a simple example : a Smalltalk version (Figure 1) and an IL version (Figure 2). A more complete description of IL can be found in [Ber97a]. We conclude this section with remarks on our different experimentations for integrating interactions to object-oriented languages.

---

[1] Aspect Weaver is a trademark of Xerox Corporation

2.1 Example: a Diary Manager in Smalltalk without IL

A diary manager handles the association between shared diaries and personal diaries. When a rendezvous is added to a shared diary the participants of this rendezvous (reunion) are notified and their personal diaries are updated.

Figure 1 shows an implementation part of this specification using Smalltalk [2].

We have colored the background of the code in grey when the code applies to interactions between objects.

[2] The `SharedDiary` object has also methods to manage the list of persons who share the diary

```
Object subclass: #Diary
     instanceVariableNames: 'aRdVList '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Diaries'!

!Diary methodsFor: 'adding'!

addRdV: aRdV
     "Add a Rendezvous to a diary"
     aRdVList add: aRdV!

modifyRdV: aRdV new: aNewRdV
     "Modify a Rendezvous to a diary"
     aRdVList del: aRdV
     aRdVList add: aNewRdV!

removeRdV: aRdV
     "Remove a Rendezvous to a diary"
     aRdVList del: aRdV! !

Diary subclass: #SharedDiary
     instanceVariableNames: 'anDiaryList'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Diaries'!

!SharedDiary methodsFor: 'adding'!

addRdV: aRdV
     "Add a Rendezvous to a shared diary"
     (anDiaryList collectDiaryOf: (aRdV affectedTo))

        do: [:each | each addRdV: aRdV]

     (aRdV affectedTo)

        do: [:each | each notify: "Rendezvous added"]

     ^super addRdV: aRdV!

removeRdV: aRdV
     "Remove a Rendezvous to a shared diary"
     (anDiaryList collectDiaryOf: (aRdV affectedTo))

        do: [:each | each removeRdV: aRdV]

     (aRdV affectedTo)

        do: [:each | each notify:
               "Rendezvous removed"]

     ^super removeRdV: aRdV!

... ! !

RainbowDiary := SharedDiary new.
SchoolDiary := SharedDiary new.
LaurentDiary := Diary new ownedBy: Laurent.
SchoolDiary addDiary: LaurentDiary.
...
```

Figure 1: An implementation of shared diaries in Smalltalk

2.2 Short description of IL

With IL, the interactions are specified outside of the objects, using only their interface. The methods in the component language describe the intrinsic behavior of objects without taking care of future participations in interactions.

The description of interactions in IL is based on the expression of *interaction rules*. An interaction rule describes, for a given message, the set of partial ordered messages corresponding to its effective execution. This description uses the sending message operator (noted `.`), two reactive operators (`;` and `//` respectively for synchronous and asynchronous sending), and a conditional operator.

The interaction rules are described in *interaction managers*. An interaction manager allows to specify the state and behaviors (for example the cardinality [OMG95, Mul97, Rat97] and the delayed execution) fitted to the designed interaction. An interaction manager can be specialized adding or completing interaction rules and participating objects.

In the IL language, an interaction manager can be statically declared between given objects. But, most of the time, the interactions between objects have to be dynamically established or destroyed. A given object (instance) interacts with different objects during the execution. So dynamically adding or removing interaction managers is essential and the resulting language must offer such a functionality.

2.3 A shared diary manager in IL

We show in Figure 2 the code of the diary manager presented in section 2.1 using IL. This manager describes the interactions between a shared diary SD, a personal diary PD and its owner P. Three rules specify the consequences of a message sending on a shared diary. For example, when a message addRdV is received by a shared diary, the RdV is concurrently added to the shared and personal diaries and then the owner is notified, if needed.

The use of IL separates the basic functionalities of objects and interactions and then greatly decreases the level of code tangling. Subclassing the Diary class is not needed to obtain shared diaries. A shared diary is just a diary interacting with other diaries. So the code is easier to extend and to reuse.

2.4 An Integrated Environment for IL

IL allows to describe interactions independently of the component language. A programming environment which allows the description and the edition of interaction rules and managers is provided with the IL language. This environment is developed in Smalltalk and includes at present time two generators, one for Corba and C++, another one for Smalltalk. This tool is an interaction editor and also allows the application designing by means of "interaction" scenarii. Interaction managers are then consequently generated in IL. In the

```
Object subclass: #Diary
        instanceVariableNames: 'aRdVList '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Diaries'!

!Diary methodsFor: 'adding'!

addRdV: aRdV
     "Add a Rendezvous to a diary"
     aRdVList add: aRdV!

modifyRdV: aRdV new: aNewRdV
     "Modify a Rendezvous to a diary"
     aRdVList del: aRdV
     aRdVList add: aNewRdV!

removeRdV: aRdV
     "Remove a Rendezvous to a diary"
     aRdVList del: aRdV! !
```

```
manager ConsistencyDi ( Diary SD, Diary PD, Person P )
{
    SD.addRdV ( aRdV ) → SD.addRdV ( aRdV ) //
        if aRdV.affectedTo ( P ) then PD.addRdV ( aRdV ) ;
            P.notify ( "Rendezvous added" ) endif,
    SD.modifyRdV:new ( aRdV, aNewRdV ) →
        SD.modifyRdV:new ( aRdV, aNewRdV ) //
        if aRdV.affectedTo ( P ) then
            PD.modifyRdV:new ( aRdV, aNewRdV ) ;
            P.notify ( "Rendezvous modified" ) endif,
    SD.removeRdV ( aRdV ) → SD.removeRdV ( aRdV ) //
        if aRdV.affectedTo ( P ) then
            PD.removeRdV ( aRdV ) ;
            P.notify ( "Rendezvous removed" ) endif
}
ConsistencyDi ( SchoolDiary : Diary,
                LaurentDiary : Diary, Laurent : Person);
```

Figure 2: A shared diary implementation in Smalltalk and IL

same time, a set of class skeletons is deduced. These skeletons help the design of an application and are used during the code generation to check the existence of classes involved in the interactions.

2.5 Experimentation with Component languages

Some experimentations have been done to integrate the interaction language to several object-oriented languages : Smalltalk [Bec97, Ber97b], Clos [ABB+89, DFP95], Open C++ [Chi95, CH97], Corba-C++ [ORB92, Nac97], Corba-Open C++. At the present time we are interested in component languages that integrate network management such as Java RMI [SUN96], Corba-Smalltalk and Corba-Smalltalk-C++. Our aim is to see the impact of a distributed environment to the architecture and the possible consequences on the aspect language (see sections 3 and 4).

According to the component language, the resulting language deals with simple interactions between objects, concurrency or communication between remote objects. For example, when the resulting language, such as our first implementation in Clos, does not manage concurrency the operator // is equivalent to ;, otherwise the example using the interactions is, without change, concurrent.

We can also note that no assumption is done concerning the physical location of objects in IL. So, when a component language is coupled with a distributed language, the interaction aspect weaver has to take into account the object location. In Corba, for example, it must generate the IDL (Interface Description Language) of the remote objects from the interaction managers described with IL. In order to hide that some objects in the interaction managers are local or remote, the weaver has to handle the physical locality of the objects at the message execution time.

So, according to the kind of component language, the interaction aspect weaver has to deal with different paradigms. In answer to this extensibility problem, we will detail in the next section the proposed architecture of an interaction aspect weaver. The following section will deal with this problem in term of aspects when the considered paradigms are weaver implementation dependent.

3 INTERACTION ASPECT WEAVER

For each experimentation, we have developed an aspect weaver. Those weavers have common features. They generate in the component language first class objects for the interaction rules and interaction managers described in IL. They add code to glue the component objects and the generated objects and they install a set of units which manage the interactions. We can consider this set as a runtime library. The principal common characteristic of all the weavers is the redefinition of the execution of a message to manage interactions. So, in the resulting language, it is necessary to catch the messages and to determine if the messages are concerned by interactions. In this case, the interaction rules concerning this message at this time are selected, and the new behavior of the message is executed. A specific unit is responsible for each step of this message execution. Each of these units has a specific behavior according to the component language and the target resulting language features. Each unit has to be easily extensible and substituted without altering the global behavior.

The chosen solution is to associate a meta-object to each interacting object and to introduce these units in each meta-object. So, we developed all our weavers using an open object-oriented language.

The description of the meta-object given in the following section is a guideline for interaction aspect weaver implementation.

3.1 Meta Architecture

Figure 3 shows the architecture of a meta-object with its units and their collaboration to manage the new message execution.

Catching Message Unit
The sending message semantic is changed because when a message involves at least an interaction rule, this or these rules are executed instead of the message itself. So we need a catching message mechanism to modify the sending message
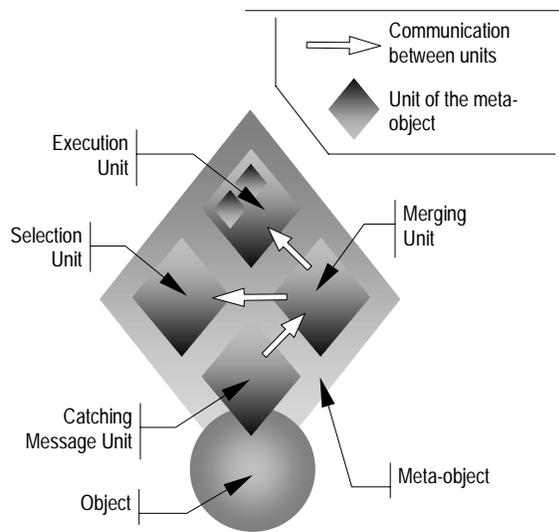
Figure 3: An overview of the meta architecture

behavior semantic. Several mechanisms for catching messages in open object-oriented languages exist. For the catching message unit, in Open C++, we have used the method wrapper ([Bra96] gives an example of method wrapper in Smalltalk). In Corba, we gain, with the Orbix version, the filters mechanism [Orb96, Nac97] that is suitable to catch the sending message between hosts. We had the choice between the techniques presented by Ducasse in [Duc97] in the Smalltalk environment and we take advantages of dynamic subclassing and compilation.

Selection Unit
This unit manages the effective interaction rules. It contains the list of all effective interaction rules in concern with the object managed by the meta-object. This list is evolutionary because the interaction rules can be included or removed dynamically, at runtime. This unit determines, when a message is received and caught, which interaction rules among the effective rules must be executed.

Merging Unit
When the set of interaction rules to execute for one sending message is fixed by the Selection Unit, the Merging Unit must merge these rules before their execution. Indeed each rule can contain the ask of the execution of the interacting message but the interacting message must be executed only one time. So we need to merge these rules to avoid extra execution of the interacting message. The merging algorithm is based on the operator semantics.

Execution Unit
The role of the Execution Unit is to manage the execution of the interaction rule resulting from the merging realized by the Merging Unit. Interaction rules and operators are first class objects which know how they execute themselves. An execution message to an interaction rule is recursively sent to its operators. So a new operator can easily be added.

We can see in the Figure 3 that the execution unit can contain other units, for example, to manage remote calls in a language that does not provide them at base level.

The implementation of these four units is complicated when the weaver has to mask the network management or to handle the concurrency. In such a case the evolution and maintainability of the meta-object is complex. The next section presents the advantage of the introduction of an aspect language to describe concurrency interactions and the future works under consideration in term of aspects.

4 ASPECT ON THE INTERACTION ASPECT LANGUAGE

The current behavior of resulting languages are not completely satisfying in some points : the network management, the synchronization of the interactions and the checking of the global interaction graph are not well integrated yet. The AOP [KLM$^+$97] can solve in part these problems. We have completed the interaction language features to the synchronized interactions due to an aspect language. We are now studying the advantages to use aspects to simplify the weaver implementation in case of distributed implementation and to introduce explicitly properties on interactions for checking the global consistency at runtime.

4.1 Interactions Synchronization : an aspect of IL

The problem of synchronization occurs when the interactions are an aspect of a concurrent or a distributed object-oriented language. We need then to express self exclusive or mutual exclusive interaction rules in an interaction manager to avoid cycle or reentrance. To extend IL with exclusion instructions is not a good solution because such an information is not used in a sequential language and transforms the code of interactions in a tangled code with synchronization. So interaction synchronization is typically an aspect for IL. After a first study of this solution, a language such as the coordinators of Cool [VLL94] seems to be a good applicant.

The synchronization program refers to the interaction program by its interaction managers and interacting messages to designate a rule in this manager. For example,

```
coordinator DiaryManagerCoord : ConsistencyDi
{
      mutexclusive { SD.addRdV ( aRdV ),
                     SD.removeRdV ( aRdV ) };
}
```

specifies that the two rules cannot be executed concurrently.

We also need sometimes to consider an interaction as an atomic transaction and more generally to declare a set of interacting messages in an atomic transaction. Let see the shared diary example with the following behavior : the ask for a modification or an addition of a rendezvous requires

the acceptance of all the persons involved in the rendezvous before the effective updating of the shared diary. This example can be directly implemented in an interaction manager between all the persons and the shared diary by asynchronous sending messages managing a two phase commit. In fact, this solution is not a satisfactory answer because it is too closely related to the implementation and too far from a specification language as IL. We have chosen to introduce the keyword `atomic` (associated to a rule or a set of rules) in the synchronization aspect language. It is more declarative and then it fits better the philosophy of IL.

4.2 Future Works around aspects on aspect

Interactions between remote objects
The problem of the network management is the complexity of the library associated to the weaver. The network management is tangled in the code of meta-object and the extension and maintainability is then difficult. Moreover the user has no explicit information concerning the network behavior and cannot control it nor customize it. So we think that the distribution management for the interactions must be an aspect. In such a case, our study concerns essentially to determine two points : which kind of language is required to express distribution for interactions and if this aspect must be associated with the interaction language, the component language, the resulting language or the Interaction Aspect Weaver design.

Interactions Graph
The last point concerns control on the complete interactions graph. It should be probably resolved by an aspect but the difficulties are to find the well adapted properties we want to express (no cycle or limited loops, detection of synchronization points, multiple calls to a same message in an interaction rule resulting from the merging, reducing a list of interaction calls, etc) and to check on this graph and the consequences of the kind of the component language (concurrent or distributed) on this set of properties.

The originality of our approach is that we can propose a basic interaction aspect that can be extended with other aspects, such as coordinator language, according to the nature of the component language or the target applications. The programmer can customize the interaction language according to the component language and the target features of the resulting language.

5 CONCLUSION

In our first works on the interactions, we were interested in proposing the "ideal" architecture to integrate interactions in your "preferred" languages. Doing so, we were language dependent and it was difficult to determine which features were required by the object-oriented language and which were required by the interactions themselves. The new vision of interactions between objects in term of an aspect of object-oriented languages allows a separation between interaction language and component languages. So, we are free to elabo-rate the syntax of our language and it is easier to check properties on the interactions. But, the objective of such a language is always to be integrated in others languages. So, due to our experimentation in the interaction weaver implementation, we are today able to propose an architecture based on meta programming to implement Weavers. Separating the interaction language from an object-oriented language leads us to tackle other points such as distribution or concurrency. Our current idea is that these features don't have to be integrated in IL, and are in fact aspects on the IL language or as to be used to implement the Weavers.

In conclusion, we think that the IL language and its programming environment are a good way to investigate and to help the programmers in a new type of programming : interaction-oriented programming. Moreover, it seems to us that the interaction-oriented programming allows to fear the component oriented programming in an interesting and promising manner.

REFERENCES

[ABB⁺89] G. Attardi, C. Bonini, M.R. Boscotrecase, T. Flagella, and M. Gaspari. Metalevel programming in CLOS. In S. Cook, editor, *Proceedings of ECOOP'89*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

[AF93] G. Agha and S. Frølund. A Language Framework for Multi-Object Coordination. In *Proc. Of European Conf. On Object-Oriented Programming '93*, number 707 in LNCS, pages 347–360, July 1993.

[BDFJ97] L. Berger, A-M. Dery, M. Fornarino, and O. Jautzy. Contribution : Interaction and Communication Models. In *ECOOP'97 Workshop Reader*. Springer Verlag, 1997.

[Bec97] K. Beck. *Smalltalk, Best Practice Patterns*. Prentice Hall, 1997. ISBN- 0-13-476904-x.

[Ber97a] L. Berger. *Définition et Implémentation d'un Modèle de Coordination entre Objets Distants*. Rapport de recherche I3S, RR-97-24, 1997.

[Ber97b] J-M. Bernelas. Rapport de stage ESSI seconde année, October 1997.

[Bos94] J. Bosch. Relations as first-class entities in layom. Not Yet Published. Available on `http://www.pt.hk-r.se/~bosch`, 1994.

[Bra96] J. Brant. Method Wrappers. Smalltalk goody, 1996. `http://st-www.cs.uiuc.edu/users/brant/Applications/MethodWrappers.html`.

[CH97] B .Cazaux and T. Haquet. FLO++ dépendances entre objets distants, April 1997. Rapport de projet de fin d'études.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.

[DDF96] A-M. Dery, S. Ducasse, and M. Fornarino. Object and Dependency Oriented Programming in FLO. In *ISMIS'96 : 9ème International Symposium on Methodologies for Intelligent Systems. LNAI*, June 1996.

[DFP95] S. Ducasse, M. Fornarino, and A-M. Pinna. A Reflective Model for First Class Relationships. In *Proceedings of OOPSLA'95*, pages 265–280, 1995.

[Duc97] S. Ducasse. Des Techniques de Contrôle de l'Envoi de Messages en Smalltalk. In *L'objet, numéro spécial Smalltalk en France : état de l'art et de la pratique*. Hermes édition, 1997.

[Frø94] S. Frølund. *Constraint-Based Synchronization of Distributed Activities*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[Hol92] I. M. Holland. Specifying reusable components using Contracts. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308, Utrecht, June 1992. Springer-Verlag.

[KLM+97] G. Kiczales, J. Lamping, A Mendhekar, C. Maeda, C lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceeding of ECOOP'97*, Xerox PARC, Palo Alto, CA, June 1997. Springer-Verlag.

[Mul97] P.A. Muller. *Modélisation objet avec UML*. Eyrolles, 1997.

[Nac97] A. Nacciu. Dependencies within a distributed object system - a CORBA approach, July 1997.

[Neu91] C. Neusius. Synchronizing Actions. In *Proc. Of European Conf. On Object-Oriented Programming '91*, number 512 in LNCS, pages 118–132, July 1991.

[OMG95] "Relationship Service Specification", Part 9 of CorbaServices Doc. Number 95.3.31, 1995. http://www.omg.org/library/corbserv.htm.

[ORB92] "The common Object Request Broker : Architecture and Specifications", Object Management Group and X/Open, 1992.

[Orb96] Orbix 2 Programming and Reference Guide, Iona Technologies Ltd, 1996. http://www.iona.com.

[Pin93] X. Pintado. Gluons: a support for software component cooperation. In Shojiro Nishio and Akinori Yonezawa, editors, *First International Symposium on Object Technologies*, volume 742 of *Lecture Notes in Computer Science*, pages 43–60, 1993.

[Rat97] Objectory Process 4.1 : Your UML Process, Rational Corp., 1997. http://www.rational.com.

[Rum92] J. Rumbaugh. Horsing around with associations. *Journal of Object Oriented Programming*, 4(6):20–29, February 1992.

[SUN96] SUN. http://www.javasoft.com:80/products/jdk/1.1/docs/guide/rmi/index.html, "Remote Method Invocation Tutorial", 1996.

[VLL94] C. Videira-Lopes and K. Lieberherr. Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications. In *European Conference on Object-Oriented Programming*, pages 81–89, Bologna, Italy, 1994.