An Aspect–Oriented Approach based on Multiparty Interactions to Specifying the Behaviour of a System [*]

Antonio Ruiz, Rafael Corchuelo, José A. Pérez, Amador Durán and Miguel Toro
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla
Avda. Reina Mercedes s/n, Sevilla 41012, Spain
Tel/Fax: +34 95 4557139, e–mail: aruiz@lsi.us.es

Abstract

Isolating computation and coordination concerns into separate pure computation and pure coordination enhances modularity, understandability and reusability of parallel and/or distributed software. This can be achieved by moving interaction primitives, which are now commonly scattered in programs, into separate modules written in a language aimed at coordinating objects and expressing how information flows among them. The usual model for coordination is the client/server model, but it is not adequate when several objects need to collaborate simultaneously in order to solve a problem because natural multiparty interactions need to be decomposed into a set of low–level, binary interactions.

In this paper, we introduce CAL, an IP–based language for the description of the coordination aspect of a system. We show that it can be successfully described in terms of simple multiparty interactions that can be animated and are also amenable to formal reasoning.

Key words: Aspect–oriented specification, coordination, multiparty interactions, distributed systems.

1 Introduction

There are several object–oriented specification languages for specifying the behaviour of a system that incorporate multiparty interactions: TROLL [7], LCM [3], TESORO [17], and so on are good examples. Unfortunately, finding a mapping to transform such specification languages into efficient programs gets more and more difficult as their level of abstraction increases. As a result, only a few relatively low–level specification languages have bridged the gap between specification and implementation. Among them, IP [4] stands out because it is equipped with sound semantics that turn it into a language amenable to formal reasoning, but it is also a viable implementation tool one can use to express abstract solutions that can be compiled and animated in an efficient way. Nevertheless, we agree with the authors of IP in that it is not intended to replace current programming languages because they rarely fade away. On the contrary, they hope to see IP incorporated into such languages to describe cooperation among objects.

We think that the new aspect-oriented approach is the key to incorporate it [8]. Figure 1 depicts the general framework we need to specify a system following this approach. First of all, the behaviour of the system needs to be decomposed into orthogonal aspects that can be specified in an independent way by means of specific languages. Several important aspects are coordination, distribution, security and computation, which are orthogonal and representative enough. Isolating such aspects enhances modularity, reusability and understandability of software, but the key here is that each aspect should be described in terms of a specific language that should be intended to deal only with the details concerning it. This way, it can be kept small and yet amenable to formal reasoning. Another important feature is that these languages can be usually translated into programming languages such as Java or C++, so that the translations can be linked into a piece of code that can then be compiled

[*]This work is funded by the CICYT project "MENHIR". TIC 97–0593–C05–01, Ministry of Education and Science (Spain)

to produce an efficient program. This task is usually done by means of a tool called *weaver* [8], which is the backbone of the aspect–oriented approach. Many references on this topic can be found, and [10] describes the state–of–the–art weaver. Unfortunately, it deals only with client/server interaction and distribution.

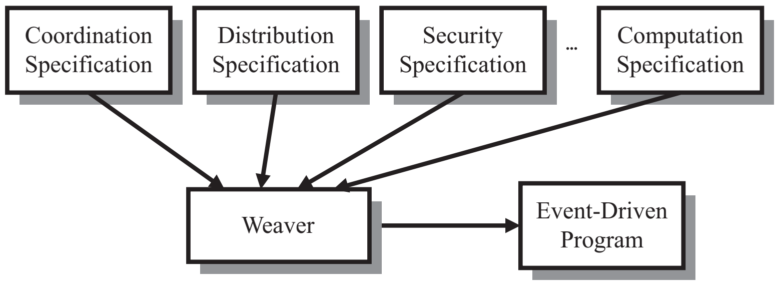
Coordination Specification
Distribution Specification
Security Specification
...
Computation Specification
Weaver
Event-Driven Program

Figure 1: A general framework for Aspect Oriented Specification

This paper aims at presenting CAL (Coordination Aspect Language), an IP–based language for specifying the coordination aspect of a distributed system. Its main feature is that it is based on multiparty interactions as the sole means of object synchronization and communication, thus providing an adequate framework for describing problems where several objects need to collaborate simultaneously to solve a problem. It is organized as follows: section 2 shows the aspect–oriented paradigm and the notion of multiparty interactions by means of the well–known debit–card system; section 3 offers an overview of our implementation; section 4 glances at other authors' work, and section 5 shows our conclusions and the work we are planning on doing.

2 Aspect–Oriented specification and multiparty interactions

In this section, we introduce the main ideas of the aspect–oriented approach and CAL by means of the well–known debit–card system. We shall also illustrate how multiparty synchronization and communication work.

2.1 Aspect–Orientation in short

Object–oriented languages such as Java or C++ are not adequate enough to model distributed systems because aspects such as coordination, distribution, communication or replication do not usually fit into the scope of a class. In fact, much of the complexity and brittleness in existing systems stems from the way the implementation of these aspects ends up in *spaghetti code* that is plenty of problems such as expensive maintenance, difficult understanding, dependence on the libraries we use to implement these aspects, and so on. However, spaghetti code is not a real problem unless it needs to be written and supported manually.

In order to deal with these cross–cutting aspects, a number of researchers began working on several approaches to this problem that allow programmers to express each aspect in a separate module that can be described by means of *ad hoc* languages. These languages incorporate only those constructs needed to deal with the details concerning an orthogonal aspect, and they should be designed so that they can be translated into an efficient programming language and combined with the languages we use to describe other aspects of the same system. This approaches are usually called *Aspect–Oriented* and several interesting proposals may be found in the literature [8]. We shall glance at them in section 4.

Figure 2 sketches our aspect-oriented proposal for specifying the behaviour of a distributed system. The coordination aspect is specified in CAL, computations are described in Java, and a weaver combines both sources into a piece of code that can then be compiled and executed on top of a CORBA layer that we describe in section 3. Notice that CAL, is only intended to describe coordination, i.e., how objects interact in order to solve a problem and how data flows among them. In doing so, we need to call methods that are described in Java, our computation language, but what we must point out is that those methods are written without taking coordination details into account. Decomposing a system into these two layers that deal with computation and

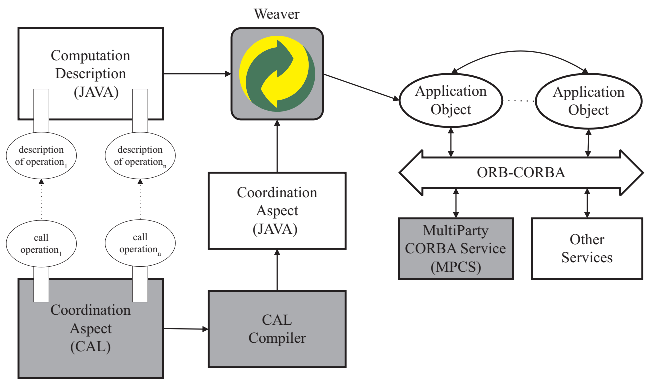
Weaver
Computation Description (JAVA)
description of operation_1
description of operation_n
call operation_1
call operation_n
Coordination Aspect (CAL)
Coordination Aspect (JAVA)
CAL Compiler
Application Object
Application Object
ORB-CORBA
MultiParty CORBA Service (MPCS)
Other Services

Figure 2: Our proposal to specifying coordination by means of aspects

coordination separately enhances modularity, reusability and understandability.

2.2 The Debit–Card System

The debit–card system can be viewed as the basic behaviour pattern in a distributed electronic commercial system. It is composed of a set of n point–of–sales terminals, and a number of computers that hold m customer accounts and p merchant accounts. This is a problem that can be clearly described by means of multiparty interactions because a three–party interaction needs to be carried out when a clerk inserts a debit card into a terminal in order to transfer funds from a customer's account to a merchant's account. We say that these objects need to reach an agreement so that funds can be correctly transferred. This problem is usually described in terms of client/server primitives, but we shall show that the multiparty interaction approach makes our solution simpler.

The key concept is *multiparty interaction point* (MIP). An MIP is a shared event that several objects must engage simultaneously so that it can occur. In addition, each MIP can be equipped with a number of slots that the objects that synchronize on it can use to exchange information. Figure 3 depicts several MIPs called $Transfer_{i,j,k}$ that $Terminal_i$, $Customer_j$ and $Merchant_k$ use to transfer funds. When a clerk insert a debit card into terminal i to transfer funds from the account of $Customer_j$ to the account of $Merchant_k$, it engages $Transfer_{i,j,k}$, which is a three–party MIP and cannot be fired until $Customer_j$ and $Merchant_k$ also engage it. When this happens, we say that this MIP is enabled and thus can be fired. Therefore, MIPs can be viewed as shared events that can only occur as long as all of their participants have engaged them.

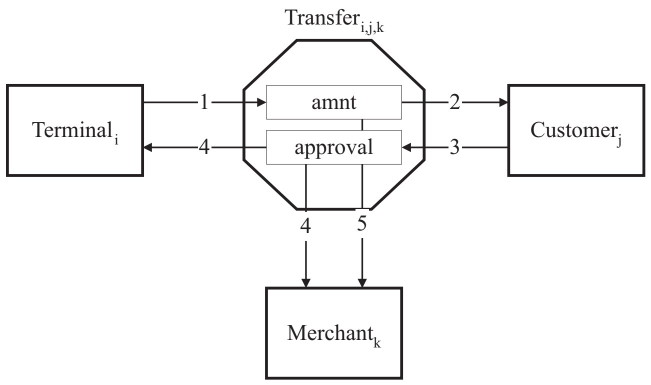

Figure 3: Information exchange through multiparty interaction point $Transfer_{i,j,k}$

Nonetheless, multiparty synchronization is not enough to describe this problem. We also need multiparty communication so that the objects that have synchronized on $Transfer_{i,j,k}$ can exchange information and really transfer funds. This can be done by using the slots $Transfer_{i,j,k}$ has. Each slot is a piece of data an object can read and/or write. For example, when $Transfer_{i,j,k}$ is fired, $Terminal_i$ stores the amount of money

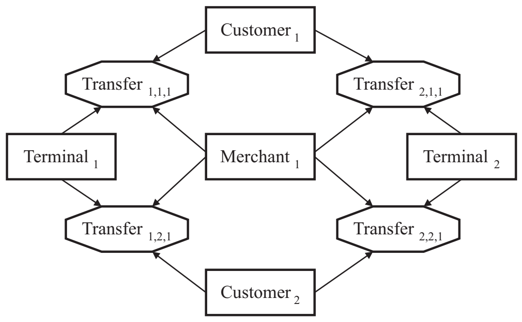

Figure 4: Multiparty interaction points needed in a particular DCS.

to be transferred in slot $amnt$, so that $Customer_j$ can read it and decide whether it can afford that purchase or not. It stores the result in slot $approval$ so that $Terminal_i$ and $Merchant_k$ can then decide what to do then. Obviously, a MIP delays an object's trying to read a slot that has not been initialized yet. Thus, they can also be viewed as critical regions where no race conditions or deadlocks can occur.

We think that this approach is suitable to describe problems where several objects need to agree and collaborate simultaneously in order to solve a problem because we do not need to design a specific protocol to transfer information so that no deadlock or race condition can arise. On the contrary, it is implicitly included in our multiparty interaction mechanism.

2.3 A glance at CAL

CAL has evolved from IP, being the main difference that a MIP can have a number of slots objects synchronizing on it can use to exchange information. In CAL, systems are understood as collections of co–operating objects whose relationships are based on multiparty interactions. Nonetheless, CAL does only provide a basic notation for describing cooperation among objects, thus their computational aspects need to be described in another language such as Java or C++.

CAL offers a small set of statements, being the most interesting the interaction statements. They are of the form $a[com]$, where a is referred to as the name of the interaction point in which this statement is interested, and com is an optional sequence of assignments or method invocations referred to as the communication part. CAL also provides guarded non–deterministic choice statements of the form $[[]_{i=1}^{n} G_i \rightarrow S_i]$, guarded non–deterministic loops $*[[]_{i=1}^{n} G_i \rightarrow S_i]$ and dummy statements denoted by the key word $skip$. Guards are of the form $B\&a[com]$, where B is a boolean expression and the rest is an usual interaction statement.

Figure 5 sketches a specification of the debit–card system in CAL. We have stated that a system is composed of n objects of class $Terminal$, m of class $Customer$ and p of class $Merchant$. They interact simultaneously by means of the $m * n * k$ MIPs that we have depicted in figure 4 when $m = 2$, $n = 2$ and $p = 1$. They have two natural slots: $amnt$, which is the amount of money to be transferred, and $approval$, which denotes if a customer has enough money in his/her account to afford a purchase. We have also stated that $Terminal_i$ is responsible for initializing slot $amnt$, $Customer_j$ is responsible for initializing $approval$, whereas $Merchant_k$ just synchronizes in this interaction point and reads the information the other objects store in its slots.

The behaviour of these objects is quite simple: $Terminal_i$ reads the amount of money a sale has cost by calling routine $next\_sale()$, which is described in the computation aspect by using Java; afterwards, it tries to engage interaction $Transfer_{i,j,k}$ together with $Customer_j$ and $Merchant_k$. When this MIP is fired, the communication part begins to execute, and it leads to the information exchange showed in figure 3: firstly, $Terminal_i$ initializes slot $amnt$ ($amnt := s$); afterwards, $Customer_j$ initializes $approval$ by calling method $get\_approval()$; next, $Customer_j$ and $Merchant_k$ update their balance and, finally, $Terminal_i$ reads slot $approval$ in order to tell the clerk whether the purchase was approved or not.

Notice that this mechanism is very flexible and allows for a number of processes to exchange information simultaneously. This is a new approach to communication, which has been traditionally broken into send/receive statements where an object sends a piece of information to another that just waits for it.

3 Implementation issues

We have implemented a CAL compiler, and the target language we selected was JAVA. The compiler uses several transformation schemes based on the multiparty CORBA service we describe in this section. This details are clearly beyond the scope of this paper, but we want to show that our approach works quite well. The reader is referred to [15] for further details about our impl ementation.

3.1 The overall picture

The need to simplify distributed application development has led to distributed object environments called middlewares. They provide the benefits of object–orientation (separation of the interface of an object from its implementation, inheritance, sub–typing, and so on) and a uniform method for calling remote procedures (RPC). They rely on an *interface description language* (IDL) for defining the interfaces of an object, and there are IDL compilers that can translate those interfaces into pieces of code that provide RPC support. Interfaces may be implemented using any of the languages for which an IDL compiler has been developed.

We have chosen CORBA [12] as the middleware we use to implement CAL because it is a *de facto* standard and there are many commercial or public domain implementations. This way, objects are translated into CORBA objects that can synchronise and exchange information by means of the *MultiParty CORBA Service* (MPCS) we have implemented. In general, we should have a service for implementing each aspects of a system. In order to simplify the weaver, CAL specifications are translated into JAVA so that the it needs to deal only with one language.

Our implementation is based on an multiparty distributed synchronization algorithm that we have designed for this purpose. This algorithm is an evolution of the one presented in [2] and is fully described in [15]. Roughly speaking, it is based on having one interaction manager per interaction point and a scheduler that decides which interaction should be fired when multiple interactions are enabled at the same time. Every process which needs to participate in one or more interactions, uses a proxy that manages its coordination aspects.

3.2 Experimental results

We have carried out a performance analysis on our algorithm in two different ways: (a) carrying out an analysis of performance versus the number of interaction points in a system, (b) carrying out an analysis of performance versus the number of participants per interaction (i.e. interaction cardinality). The results of the first analysis are very complex, since performance depends on many variables, such as the number of participants versus the number of interactions, average cardinality of interactions, average number of shared participants among interactions (this may be the most influential factor on performance). A discussion on these results is beyond the scope of this paper, but the reader interested is referred to [15]. The results of the second analysis are quite more simple, since our algorithm is designed to behave linearly (in the worst case) with respect to the number of participants per interaction.

In order to prove our theoretical results, we have carried out an empirical test. We have built a system consisting on one interaction point, and two processes which are always ready to participate in it. After each synchronization, one process (writer) stores a value in a slot, an the other process (reader) reads it. The total time elapsed to complete 1.000 interactions was measured, and the experiment was then repeated adding more readers, increasing the interaction cardinality to three, four, an so on.

```
/* Coordination Aspect */

DCS :: $\|_{i=1}^{n}$ Terminal$_i$ $\|_{j=1}^{m}$ Customer$_j$ $\|_{k=1}^{p}$ Merchant$_k$
through
  Transfer$_{i,j,k(i=1,2,\ldots,n)(j=1,2,\ldots,m)(k=1,2,\ldots,p)}$ [amnt: float,approval: boolean] among
    Terminal$_i$ writes amnt; Customer$_j$ writes approval; Merchant
where
  Terminal$_{i=1,2\ldots n}$ ::
    { s: float; ok: boolean }
    s := next_sale();
    *[ $[]_{j=1,k=1}^{m,p}$Transfer$_{i,j,k}$[amnt := s, ok:= approval] $\rightarrow$ show_result(ok); s := next_sale()].

  Customer$_{j=1,2\ldots m}$ ::
    *[ $[]_{i=1,k=1}^{n,p}$Transfer$_{i,j,k}$ [approval := get_approval(amnt); update_balance(amnt, approval)] $\rightarrow$ skip; ].

  Merchant$_{k=1,2\ldots p}$ ::
    *[ $[]_{i=1,j=1}^{n,m}$Transfer$_{i,j,k}$[update_balance(amnt, approval)] $\rightarrow$ skip; ]..

/* Computation Aspect */

public class Terminal {
  private static DataInputStream stdin = new DataInputStream(System.in);

  public float next_sale() {
    System.out.println (“Price: ”); return stdin.readFloat();
  }

  public void show_result(boolean status) {
    if (status) System.out.println(“Funds transferred”);
    else System.out.println(“Customer’s account can’t afford this purchase!”);
  }
}

public class Customer {
  public float balance;

  public boolean get_approval(float amount) { return amount <= balance; }

  public void update_balance(float amount, boolean approved) {
    if (approved) balance -= amount;
  }
}

public class Merchant {
  public float balance;

  public void update_balance(float amount, boolean approved) {
    if (approved) balance += amount;
  }
}
```

Figure 5: A specification the debit–card system in CAL

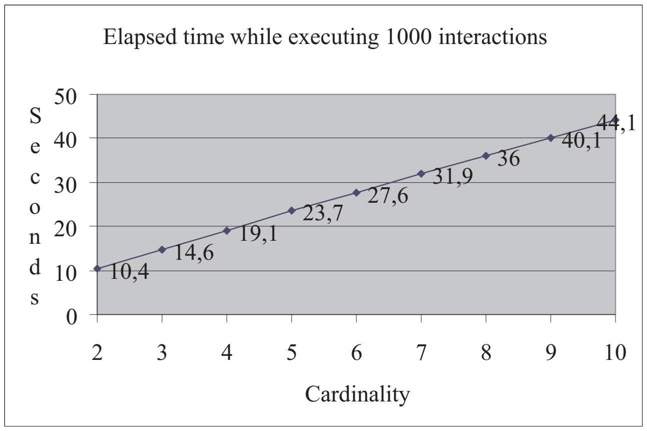

Figure 6: Performance test with one only interaction

The results of our test are shown in figure 6, where we can see that the time our implementation takes increases linearly as the number of participants increased. In fact, the resulting points can be approximated with the linear function $y = 4,2217x + 6,3917$ with $R^2 = 0,9996$, which denotes a very good linear approximation.

4 Related work

Our work tries to narrow the problem of system specification by stating one aspect at a time. This is essentially the approach "divide–and–conquer" that Edsger Dijkstra said years ago: "Treat hard problems by dividing then into several smaller problems that you can solve." However, this approach has two important problems: finding a suitable decomposition it is not easy at all, as well as composing the parts. There is a lot of work about this topic in others phases of the software development: requirements engineering, design and implementation. Next, we summarize different approaches to similar problems in these phases.

The ViewPoints project [13] is an approach to requirements engineering which has some common similarities with our proposal. ViewPoints facilitates the partition of a problem domain into loosely coupled, distributable objects that encapsulate partial specifications described in different notations, and locally developed and managed according to different work plans. Although representation, development and specification knowledge are all bundled into the same object to facilitate local management and distribution, they are separated within a single ViewPoint into slots to facilitate their individual manipulation and enhance their tailorability and reusability. Tolerating the coexistence of multiple heterogeneous ViewPoint to specify system requirements brings to the fore the problems of integration. These include the integration of specification fragments described using different notations, and the integration of methods and tools used to develop such descriptions.

With respect to programming, there are several lines very close to the aspect–oriented programming. *Subject–oriented programming* [5, 14] arose before aspect–oriented programming. Basically, systems are built as compositions of subjects each of which is a class hierarchy modeling its domain from a particular point of view. In the model of contracts [6], systems are built by using compositions of contracts where each contract specifies a set of participant objects and their interactions, expressed as obligations. Next, these interactions and obligations are encapsulated, so that they are clearly separated from other interactions involving the same objects. A single object can participate in multiple contracts, and, in this case, it must satisfy all their obligations. The model describes several combination rules for contracts, which correspond with the aspects in our approach, cutting across classes that describe objects. Finally, *adaptive programming* [9, 11] is another approach to providing modules other than classes within object–oriented systems. A class graph describes some classes and their relationships, from a particular point of view. Class graphs do not contain code; instead, code is written in separate propagation patterns. Propagation patterns can be used with any collection of concrete classes that conform to the class graph against which they were defined. Adaptive programs are transformed into standard object–oriented programs by specific tools. With respect to this generated program, each propagation pattern is an aspect, since it contains method code that cuts across classes. The composition is performed by a specific tool in which matching is based on specifications of class graph conformance.

Regarding architectural design, current Architectural Description Languages (ADLs) such as Wright and Rapide emphasize modular specification of components and their interaction. While this emphasis has demonstrated advantages for system development, in general, distributed systems entail more complicated behavior. In particular, heterogeneity, failure, and the potential for unpredictable interactions yield evolving systems which require complex management policies. Note that such policies are not isolated into interactions between components (in [1], some examples of this situation are shown). While it is possible to embed such policies within components and connectors, doing so sacrifices modularity in the same way that embedding interaction mechanisms within objects sacrifices object modularity. In [1], it is described a new model of components and connectors which exposes architectural features such as resource usage and locality that provides new abstractions for asserting policies over these features.

5 Conclusions and future work

A number of important problems in software engineering have resisted a general solution, including problems related to software understanding, maintenance, evolution, and reuse. We think that these problems share a common cause: failure of modern formalisms to satisfy the separation of concerns adequately. Numerous reasons exist to separate and integrate software, and these reasons may result in different structures [16]. In addition, many concerns may be relevant simultaneously, and the entire set of concerns may evolve over time. Despite this observation, formalisms include weak decomposition and composition mechanisms that allow only for a small, “dominant” set of concerns to be separated. This leads directly to our inability to achieve many of the goals of software engineering as a discipline. The aspect–oriented approach to specifying helps to overcome these limitations in the specification field. It allows for several kinds of separation of concerns that may not be separable in the current object–oriented model. It also promotes reuse, improves comprehension, and eases maintenance and evolution. Thus, the approach addresses some fundamental limitations in software engineering. Still, a considerable body of experience and related research now exists to support the claim that aspect–oriented specification is one of the key specification engineering issues today. The model presented is just a starting point. It must be refined, stretched and modified, and it must be instantiated for a variety of formalisms to explore issues that arise for different methodologies and at different aspects. These instantiations must be used for real development in order to evaluate them.

In this paper, we have introduced the notion of aspect–oriented specification (AOS) and a new language that allows for specifying the coordination aspect of a distributed system. This new technique is quite interesting in the field of distributed–system specification languages because it offers a new perspective to deal with distribution, coordination, fault tolerance, and so on. Basically, AOS allows the modeler to specify every aspect by means of a different formalism or language. Thus, it is possible to specify only those aspects for which we have tools able to animate them in an efficient way and describe the rest with a programming language. The main advantages of our proposal consists of a framework for specifying the behaviour of a distributed system in a flexible way, so that formal methods can be used in an industrial environment, and a better reusability of the computation specification, since this one is independent from the coordination aspects.

In the future, we are going to study how to incorporate other aspects: fault tolerance, confidentiality, and distribution, so that our work can be used in real–world problems such as e–commerce or web–based applications. In this sense, we hope to keep the independence on the platform so that our work is not bound up with a concrete language or middleware. In fact, at present, we are working on migrating our Multiparty CORBA Services towards DCOM, so that it can be used in languages such as Visual Basic or Delphi.

References

[1] M. Astley and G. Agha. Customization and composition of distributed objects: middleware abstractions for policy management. In *Proceedings of the VIth International Symposium on the Foundations of Soft-*

ware Engineering (FSE-6, SIGSOFT '98), November 1998.

[2] R. Corchuelo, D. Ruiz, M. Toro, and A. Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the XXVth Euromicro Conference (Workshop on Network Computing)*, Milan (Italy), September 1999.

[3] R. B. Feenstra and R. Wieringa. LCM 3.0: A language for describing conceptual models. Technical Report IR–344, Faculty of Matematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[4] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[5] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the VIIIst Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA' 93*, pages 411–428, Washington DC, USA, October 1993. ACM.

[6] L.M. Holland. Specifying reusable components using contracts. In *Proceedings of the European Conference on Object-Oriented Programming. ECOOP' 92*, pages 287–308, Utrecht, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science, n. 615.

[7] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL: A language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):157–211, April 1996.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 Proceedings*, pages 220–242. Lecture Notes in Comnputer Science, Springer-Verlag, 1997.

[9] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[10] C.V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Xerox Palo Alto Research Center, 1998.

[11] M. Mezini and K. Lieberherr. Adaptative plug-and-play components for evolutionary software development. In *Proceedings of the XIIIst Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA' 98*, pages 97–116, Vancouver, British Columbia, Canada, October 1998. ACM.

[12] T. Mowbray and W. Ruh. *Inside CORBA: distributed object standards and applications*. The Addison-Wesley Object Technolgy Series. Addison-Wesley, 1997.

[13] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 199.

[14] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *TAPOS: Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[15] J.A Pérez, A. Ruiz, and R. Corchuelo. Implementing multiparty interactions in the context of CORBA. Technical Report CS-TR-03–1999, Dpto. Lenguajes y Sistemas Informáticos. Universidad de Sevilla, 1999.

[16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the XXIstInternational Conference on Software Engineering. ICSE' 99*, pages 107–119, Los Angeles, California, May 1999. ACM.

[17] J.A. Troyano, J. Torres, and M. Toro. TESORO: A technique for distributed systems specfication. In *Proceedings of the IIIrd Euromicro Workshop on Parallel and Distributed Processing*, pages 563–570, San Remo (Italy), January 1995.