Component Interoperability

Antonio Vallecillo[1], Juan Hernández[2], and José M. Troya[1]

[1] Dept. Lenguajes y Ciencias de la Computación. Universidad de Málaga. Spain
{av,troya}@lcc.uma.es
[2] Dept. Informática. Universidad de Extremadura. Spain
juanher@unex.es

Abstract. Component-based software development is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manners. In this new setting, interoperability is one of the essential issues, since it enables the composition of reusable heterogeneous components developed by different people, at different times, and possibly with different uses in mind. Currently most object and component platforms (such as CORBA, DCOM, or EJB) already provide the basic infrastructure for component interoperability at the lower levels, i.e., they sort out most of the "plumbing" issues. However, interoperability goes far beyond that; it also involves behavioral compatibility, protocol compliance, agreements on the business rules, etc. This chapter tries to go through the basic concepts related to component interoperability, with special emphasis in the syntactic, protocol and operational specifications of components. Our main goal is to point out the existing problems, survey the current solutions and how they address those problems, and to draw attention towards some of the still open issues and challenges in this interesting research area.

1 Introduction

Component-Based Software Engineering is an emergent discipline that is generating tremendous interest due to the development of plug-and-play reusable software, which has led to the concept of '*commercial off-the-shelf*' (COTS) components. Constructing an application under this new setting involves the use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind. The ultimate goal, once again, is to be able to reduce development costs and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated.

This approach moves organizations from application *development* to application *assembly*. The development effort now becomes one of gradual discovery about the components, their capabilities, their internal assumptions, and the incompatibilities that arise when they are used in concert. Therefore, one of the key issues of building applications from reusable components is *interoperability*.

Interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction [29, 47]. Traditionally, two main levels of interoperability have been distinguished: the *signature level* (names and signatures of operations), and the *semantic level* (the "meaning" of operations). The first one deals with the "*plumbing*" issues, while the second one covers the "*behavioral*" aspects of component interoperation. Konstantas also refers to them as the "*static*" and "*dynamic*" levels [29], and Bastide and Sy talk about components that "*plug*" and "*play*" for referring to those two levels [4].

Currently the situation is stable at both levels. In the first place, syntactic interoperability is now well defined and understood, and middleware architects and vendors are trying to establish different interoperational standards at this level. They have given birth to the existing commercial component models and platforms such as CORBA, EJB, or DCOM. However, all parties are starting to recognize that this sort of interoperability is not sufficient for ensuring the correct development of large component-based applications in open systems.

On the other hand, the existing proposals at the semantic level endow component interfaces with information about their behavior [31]. Although much more powerful than mere signature descriptions, dealing with the behavioral semantics of components introduce serious difficulties when applied to large applications: the complexity of proving behavioral properties of the components and applications hinders its practical utility. As a matter of fact, most of the formal notations have been around for several years already, and they recurrently appear whenever there is a new paradigm, call it ADTs, objects, or components. However, the industrial use of formal methods for proving "semantic" properties of applications is still quite reduced. There have been some partial advances in this level, but we are still far from reaching any satisfactory solution yet. Besides, semantic interoperability should go beyond operational semantics or behavioral specifications of components. Agreements on names and meaning of shared concepts (ontologies) are also needed, as well as context dependencies, quality of service information, non-functional requirements, etc. [25]

In this chapter we will also contemplate a different interoperability level —the *protocol* level—, that deals just with the partial order between the components' exchanged methods, and the blocking conditions that rule the availability of the components' services. This level, firstly identified by Yellin and Strom [49], provides more powerful interoperability checks than those offered by the basic signature level. Of course it does not cover all the semantic aspects of components, but then it is not weighed down with the heavy burden of semantic checks. At the protocol level the problems can be more easily identified and managed, and practical (tool-based) solutions can be proposed to solve them.

2 Objects, Components and Interfaces

As soon as the SE community discovered objects, object-oriented programming became of the pillars on which software engineering was based. Object systems

were built, and hence object interoperability was one of the issues involved in the development of open heterogeneous applications.

However, new entities called *components* soon sprung up, shifting the focus towards them for the construction of open and evolvable applications. In general, software components are binary units of possibly independent production, acquisition and deployment that interact to form a functioning system [44]. Objects and components are not the same, they can be even considered as orthogonal concepts (a discussion about the distinction between both is beyond the scope of this paper, but the interested reader can consult [7, 26, 31, 44, 45]). However, most of the issues related to interoperability are common to both objects and components, and therefore the two will be treated here.

Something that is common to both objects and component is encapsulation, which means that their capabilities and usages can be specified by means of *interfaces*. An interface can be defined as "a service abstraction, that defines the operations that the service supports, independently from any particular implementation" [30]. Interfaces can be described using many different notations, depending on the information that we want to include, and the level of detail of the specification. For instance, the interfaces of CORBA distributed objects are described using an Interface Description Language (IDL) which contains syntactic information about the objects' supported capabilities. Since CORBA follows an object-oriented model, CORBA interfaces consist of the object public attributes and methods. Microsoft's COM follows a similar approach, although COM components may have several interfaces, each one describing the signature of the supported operations. The new CORBA Component Model CCM [43] also contemplates that components may describe not only the services they support, but also the interfaces they require from others components during their execution. Other approaches include further syntactic information, such as the events that components may raise, or some constraints on the usage or capabilities of components.

Apart from providing a textual description of the components' functionality, there are two main benefits of using IDLs. First, IDL descriptions can be stored in repositories, where service traders and other applications can locate and retrieve components from, and use them to dynamically learn about object interfaces and build service calls at run time. And second, IDL descriptions can be "compiled" into platform-specific objects, providing a clear frontier between component specification and implementation, which facilitates the design and construction of open heterogeneous applications.

3 Signature Interoperability

At the signature level, component interoperability is based on the signature of the operations that components offer, i.e., the names and profiles (parameter types and return values) of the components' operations. Based on this information, there are two major checks that can be carried out between components: compatibility and substitutability.

In the first place, *compatibility* can be described as the ability of two objects to work properly together if connected, i.e. that all exchanged messages and data between them are understood by each other. At the communication level current object and component platforms (such as DCOM or CORBA) already provide mechanisms for component interoperability. These platforms allow heterogeneous components to interoperate based just on "syntactic" agreements. They use the IDLs of components to disclose syntactic information about their services, independently from their implementation language or execution model, and provide communication and data exchange mechanisms for letting them interact.

In addition, most of these platforms are now providing bridges among them, allowing their components to seamlessly interoperate with the rest of the platforms' components. These facilities fix a known limitation that object platforms had: being *closed to themselves*.

Although all those component models provide developers with the infrastructure for supporting component compatibility at the signature level, this is not the only issue at this level: Zaremski and Wing also raised a very important problem, which they called *specification matching*. This is a process that tries to determine if two software components can be related, and how [50]. Here, based on the description of the interfaces of two components, the question is whether one can be replaced by the other. This notion, also known as *substitutability*, is crucial in CBSE since it not only has to do with the evolution and maintenance of software systems but, more importantly, it constitutes the basis for the search and retrieval processes of software components: traders look for COTS components living in a repository that can substitute the specification of *abstract* required components. Substitutability and compatibility are the two flip sides of the *component interoperability* coin.

At the signature level, substitutability of component A by component B is roughly a matter of checking that all services offered by A are also offered by B, adorned at most with contravariance of input parameter types and covariance of output parameter and return types (i.e., more specific inputs, and more general outputs). In object systems, this is known as subtyping.

Unfortunately, the information provided at the signature level has proved to be insufficient for achieving effective software reuse and resource discovery in large real-world applications. In general, we cannot suppose that software services developed by independent parties will be assigned the same names and profiles. And if we relax the matching condition to just parameters and return types (without explicit names, i.e., modulo variable renaming), we may get into troubles: for instance, integer addition and substraction both have the same signature, but completely opposite behavior; the same happens to the C library routines `strcpy` and `strcat`, and users would be very unhappy if one were substituted for the other. Furthermore, in large software modules many methods have the same signature but very different behaviors. For instance, in the C math library nearly two thirds of the functions are `float`→`float` [51].

Another limitation that most current approaches present at this level is that interfaces only describe the services that components offer (i.e., their *supported* operations), but not those services that they require from other components during their execution (i.e., their *required* operations). This issue, firstly raised in [41], is crucial in component-based development environments, and the new component models (e.g. CCM) are already taking it into account. Furthermore, nothing is said either about the order in which the objects expects their methods to be called, their blocking conditions, or their functionality.

4 Semantic Interoperability

The term *semantic* interoperability was first coined by Sandra Heiler [25], and tries to ensure that the exchange of services and data among components in large-scale distributed systems makes sense, i.e. that requesters and providers have a common understanding of the meaning of the requested services and data.

In general, semantic interoperability of components is a difficult issue, mainly because embracing all semantic aspects of components interoperation is almost impossible. This is one of the reasons why traditional studies on semantic interoperability concentrate just on the behavioral specifications of components.

4.1 Behavioral Semantics

There are many proposals that try to endow component interfaces with semantic (behavioral) information (for a comprehensive survey on current approaches we refer the reader to [31]). Each proposal uses a different notation depending on the problems being addressed, and the properties that need to be proved: temporal logic, pre/post conditions, process algebras, petri nets, refinement calculus, etc. However, component compatibility and substitutability are common checks to all proposals.

Here, *compatibility* is again the ability of two objects to work properly together if connected. However, at this level it means that the behavior provided by a component should be accordant to the behavior expected from its client component, as has been the basis for the "design by contract" development discipline [35]. Compatibility can be checked by proving, for instance, that a component's pre-conditions on its methods are met by the calling components when invoking them, and that methods' post-conditions satisfy the caller's expectations.

On the other hand, substitutability is known in this context as "*behavioral subtyping*". Pierre America [3] was the first author to talk about this concept, which in object systems is related to the polimorphic substitutability of objects in clients, and it means that the behavior of subclass instances is consistent with that of superclass instances. Here the behavior refers to the specification on how the object methods manipulate the object attributes, and behavioral types are defined as an extension of object signature types that try to associate behavior to signatures and to identify subtypes that conform to their supertypes not only syntactically, but also *semantically*.

The work by America has been followed by many others, which use different notations for specifying the objects' behavior and deepen into the intrinsic problems of behavioral subtyping. Along the lines of America, the works by Liskov and Wing [32], Zaremski and Wing [51], and Dhara and Leavens [16] use behavioral specifications based on pre/post conditions and invariants for proving partial correctness of object substitutability. Nierstrasz uses his own notation in [40], and Mikhajlova uses a refinement calculus-based extension of an object oriented language with non-deterministic constructs and abstract data types for their treatment of class refinement [36]. Algebraic specifications have been also used (e.g. by Goguen et al. [22]) for investigating software component search and replacement in global environments.

Of course behavioral subtyping is more expressive than signature subtyping, but the problem is that syntactic subtyping is decidable and can be checked in a reasonable time by a computer, while behavior-preserving subtyping is in general undecidable. In addition, *all* previous approaches either only partially capture the semantics of the objects they describe, or they make some assumptions which restrict the general applicability of their solutions. For instance, pre/post conditions only cover partial correctness, assuming that the methods will ever finish, but do not cover total correctness. On the other hand, the algebraic approach by Goguen in [22] supposes Church-Rosser and terminating specifications, which do not cover the case of reactive, non-terminating servers —the most common objects in open systems—. Furthermore, algebraic approaches also present some limitations when the components they specify have state, as Goguen has pointed out. This issue has given birth to the so called *Hidden Algebra* [21].

An additional drawback of all those approaches is again that none of them takes into account the external calls that objects make to other objects when implementing their methods, the same that happened at the signature level. In this sense, the approach by Büchi and Weck [8] proposes grey-box component specifications based on simple extensions of programming languages, which provide a richer solution for specifying components based on pre/post specifications, assertions and invariants, which also accounts for external and self-referential calls. Once again, not the whole semantics of the components can be expressed with this approach, but enough to deal with the behavior of the described objects.

Finally we will mention rewriting logic [34], an approach whose interest is rapidly growing. Although starting from the same equational approach as Goguen, Maude's latest extensions to deal with object oriented specifications have shown its great possibilities for specifying the behavior of objects and components [14, 15]. Maude [12] is a multi-paradigm executable specification language based on rewriting logic. Maude integrates an equational style of functional specification in its equational logic sublanguage, with an object-oriented specification style for object systems that can be highly concurrent and nondeterministic. Maude specifications can be efficiently executed using the Maude rewrite engine, thus allowing their use for system prototyping and debugging of specifications.

4.2 Full Semantic Interoperability

However, semantic interoperability goes beyond behavioral subtyping. Apart from the "functional", "operational", or "computational" specifications we also need to express other issues. For instance, we need to be able to express the fact that a given object can be used in a certain context, independently from its signature, access protocols or operational semantics. Furthermore, describing the behavior of objects with pre/post conditions and invariants does not guarantee that the semantics obtained is useful to describe the interoperability of objects. For instance, two objects may have the same invariants concerning their internal state, but they can have different access protocols. So they cannot replace each other.

Actually, if we go back to the original sources, the term *semantic* interoperability was first coined by Sandra Heiler [25], and tries to ensure that the exchange of services and data among components in large-scale distributed systems makes sense, i.e. that requesters and providers have a common understanding of the meaning of the requested services and data. Semantic interoperability is based on agreements on, for example, the algorithms for computing the requested values, the side effects of procedures, or the source or accuracy of requested data elements. In her paper, Heiler points out some of the problems of making semantic information explicit. Even more difficulties appear when systems and components are used (or have to be used) in new environments not anticipated by their original developers. Software engineering is full of examples that show the disasters that semantic mismatching can cause in real applications: the Ariane crash and the loss of the Mars Polar Lander are two of the latest examples (for very complete listings, see [1] or [27]).

As we see it, we are currently in a position to address interoperability of objects at the "behavioral" sublevel, but still far for tackling the whole semantic level. In particular, we count now with notations that can capture the functional semantics of objects (such as those based in pre/post conditions, rewriting logic, or description logics [6]), even when they are computationally untractable due to their intrinsic complexity. However, we do not have so many options (if any) for coping with the rest of the object semantics so far. Probably this is one of the major challenges of the software engineering community in we want to effectively talk about a real (automated) software marketplace.

5 Protocol Interoperability

A new interoperability level was firstly identified by Yellin and Strom [49] on top of the signature level, inspired by the works by Nierstraz [40], and Allen and Garlan [2]. Named after their paper as the "*protocol*" level, it deals with the relative order in which an object expects its methods to be called, the order in which it invokes other objects' methods, and the blocking conditions and rules that govern the object interactions.

At this level, two components A and B are said to be *compatible* if the restrictions imposed on the interaction of each component when they call each

other are preserved, i.e. their protocols match each role they share, and their communication is dead-lock free. *Substitutability* checks are also possible at this level. Here, in order to test if component A can be replaced by component B we need to check two main issues. First, that all messages accepted by A (i.e. supported operations) are also accepted by B, and that B's outgoing messages when implementing A's services (i.e. both the replies and the required operations) are a subset of the outgoing messages of A. And second, we need to test that the relative order among the incoming and outgoing messages of both components are consistent [9, 49].

There are several proposals that address object interoperability at the protocol level. For instance, Doug Lea [30] proposes in 1995 an extension of the CORBA IDL called PSL to describe the protocols associated to an object's methods. This approach is based on logical and temporal rules relating situations, each of which describes potential states with respect to the roles of components, attributes, and events. Although it is a very good and expressive approach, it does not account for the services an object may need from other objects, neither it is supported by proving tools.

The previously mentioned work by Yellin and Strom [49] is a more general approach for describing object service protocols using finite state machines, that describe both the services offered and required by objects. It is based on a very simple notation and semantics that allow components to be easily checked for protocol compatibility. However, this approach does not support multi-party interactions (only contemplates two-party communications), and the simplicity that allows the easy checking also makes it too rigid and unexpressive for general usage in open and distributed environments.

The approach by Cho, McGregor and Krause [11] uses UML's OCL to specify pre and post conditions on the objects' methods, together with a simple finite state machine to describe message protocols. Similarly, Jun Han [24] proposes an extension to IDLs that include semantic information in terms of constrains and roles of interaction between components (but using no standard notation), that aims at improving the selection and usage processes in component-based software development. The behavior of components is described in terms of constraints, that are expressed in a subset of temporal logic. They are somehow similar approaches, although none of them is associated to any commercial component platform like CORBA or EJB, nor supported by standard tools. In addition, Jun Han's approach deals with component interfaces in terms of attributes, methods and events, although there is no explicit treatment of "required" services.

Bastide et al. [5] use Petri nets to describe the behavior of CORBA objects. Although this approach deals with both supported and required services and it is supported by tools, it describes operation semantics and interaction protocols altogether, without a clear separation of both semantic dimensions. Besides, it inherits some of the limitations imposed by the Petri nets notation: the lack of modularity and scalability of the specifications.

An approach that describes object service protocols using process algebras is due to Canal, Fuentes, Troya, and Vallecillo [9]. They propose an extension of

the CORBA IDL that uses a textual sugared subset of the polyadic π-calculus to describe CORBA object service protocols. The π-calculus is very well suited for describing component interactions, and it also allows powerful (tool-based) interoperability tests. The limitations of this approach are probably the low-level descriptions (inherited from the π-calculus, although leveraged by some syntactic sugar) and its so minimalistic semantics, which does not provide very rich feedback to system designers when errors are detected during protocol checks.

On the other hand, current model checkers (e.g. those based on Promela or SDL) also allow the specification of the observable behavior of objects. They are very good for cleaning up system designs quickly because of their efficiency (they work in polynomial time) and the quality and expressiveness of their feedback. However, a disadvantage of model checkers is their inability to deal with arbitrary number of instances, or with properties that involve universal quantifiers.

Finally, we would like to mention Message Sequence Charts (MSC, [28]), a notation that also permits the description of the interactions among objects. Now part of UML, the major problem is the lack of any formal foundation that supports this notation. They are very expressive for describing protocol interactions, but they do not allow the proof of (safety or liveness) properties about the system, or the inference of certain results.

6 A case study

In order to illustrate the problems and how they are addressed by some of the proposals at different levels, let us use a simple example. It was first introduced by Szyperski in his book [44], and has also been extensively used by other authors (e.g. [8, 20]) for presenting their proposals. It is about the problems of properly specifying components, in particular under the presence of call-backs —as used in the observer design pattern. Call-backs are commonly used for making systems extensible, and constitute one of the key mechanisms in component frameworks [17, 18].

Let us suppose a text editor, developed following a Model-View-Controller architecture. This architecture provides a clear separation of concerns between internal representation and manipulation of data (model), data representation to the user (view), and command interpretation (controller). An instance of the model component should work with zero, one, or more simultaneous views, and must not depend on their form of presentation. Therefore, the observer pattern is used: the model is the subject, and the views are registered as observers and notified when the model changes.

6.1 Signature 'specification'

Imagine we want to design the system using CORBA objects, and then the following CORBA interfaces define the components used in the system (to keep the example simple we will just concentrate on the functions needed to delete characters).

```
interface ITextModel {
 int length();
 char charAt(in int pos);
 void deleteCharAt(in int pos);
 ...
 void register(in ITextObserver o);
 void unregister(in ITextObserver o);
};
```

```
interface ITextObserver {
  void deleteNotif(in int pos);
};
```

The first interface (`ITextModel`) defines three functions for knowing the current length of the text, providing the character at a given position, and for deleting a character. This interface also offers two operations for managing the list of registered observers that want to be notified when a character is deleted in order to modify their views. The second interface (`ITextObserver`) defines just a method, the one that will be called by the text model when a character is deleted from the text.

Providing just interface definitions and plain (English) text for specifying components is the traditional way used by most software developers and systems integrators. It is, for instance, the way CORBA services are currently specified. However, this sort of specifications have proved to be inadequate in many situations, as for instance Rémi Bastide and Ousmane Sy have shown [4]. They analyzed the CORBA Event service specification and five commercial implementations were checked against underspecifications found. The result was that the different implementers did not produce interoperable and substitutable implementations; the distinct implementations offered different behavior and exhibit behavioral incompatibilities with the rest.

Therefore, more complete specifications are needed. One way of fulfilling this need is by providing "behavioral" specifications of the components.

6.2 Behavioral specifications

There are many different notations for specifying the behavior of components, and to reason about them. Probably one of the most common approaches uses pre/post conditions. In the particular case of our example, a possible specification of the two components may be written as follows (as shown in [8]):

```
interface ITextModel {
  //@ setof <<ITextObserver>> registeredObservers = {}
  //@ seqof <<char>> text = <>
  int length();
      //@ pre:  true
      //@ post: result == len(text)
  char charAt(in int pos);
      //@ pre:  0<=pos && pos<len(text)
      //@ post: result == text[pos]
  void deleteCharAt(in int pos);
      //@ pre:  0<=pos && pos<len(text)
      //@ post: \forall i [0..pos-1]: text'[i] == text[i] &&
```

```
      //@       \forall i [pos..len(text)-1]: text'[i] == text[i+1] &&
      //@       len(text') == len(text) -1
  ...
  void register(in ITextObserver o);
      //@ pre:  true
      //@ post: registeredObservers' == registeredObservers ++ {o}
  void unregister(in ITextObserver o);
      //@ pre:  true
      //@ post: registeredObservers' == registeredObservers -- {o}
};

interface ITextObserver {
  void deleteNotif(in int pos);
  //@  pre: "character at position 'pos' has just been deleted"
  //@  (informal comment, not tool checkable)
};
```

The first specification shows how the operations of component `ITextModel` modify its state, defined here by a sequence of characters with the text being edited, and a set with the registered observers.

However, this specification suffers from several drawbacks. The first one is that specification of `deleteCharAt()` does not mention anything about calling the registered observers, and therefore we cannot know when or how they are notified. The main reason is that external calls cannot be expressed with pre/post conditions because they do not modify the state of the text model! We could include there some informal comments, as we have done for the `deleteNotif()` method (some languages from the Larch family [23] even provide the clause "`informally`" for including this sort of informal specs). However, none of these comments in natural language are machine checkable and enforceable [8].

A second question not answered by this specification is about reentrance: what happens if an observer registers another one during the execution of its `deleteNotif()` method? And if an observer deletes a character during the execution of its `deleteNotif()` method? Usually this sort of reentrant calls to object modifier methods are banned in notification methods [13], since they may make components very difficult to understand and to reason about; they may even lead to infinite mutual recursion. However, with pre/post conditions there is no information about the external calls being made from a component; this sort of specifications only deal with how operations modify the state of the object. Here we find again another situation in which we need to have knowledge about the external "required" methods by a component, and we need some notations for capturing those object interactions.

6.3 Protocol specifications

Taking a look at the limitations that the previous specifications present, we can see that they are mainly due to their inability to describe the interactions among components, i.e., how and when components call other components' methods, and the restrictions imposed on those interactions.

As mentioned earlier, protocol specifications try to address those issues, offering a description of the components' interactions. Although they do not account for the whole semantics of the components, they may provide complementary information about their behavior, and at a lower cost. In general, protocol checks do not need such a heavy machinery as theorem provers do, but still they may be (theoretically) untractable. However, most of the component protocol descriptions are usually very simple in most real applications, and therefore even the protocol tests with an a priori exponential complexity can be practically used in these situations.

In this section we will present one of the approaches that deals with component interoperability at the protocol level. It is due to Canal, Fuentes, Troya and Vallecillo [9], and for the description of protocols it uses a sugared subset of the polyadic π-calculus, a process algebra specially well suited for the specification of dynamic and evolving systems. In addition, the tests that are possible with that proposal are also described.

6.3.1 *Extending CORBA interfaces with π-calculus*

The π-calculus was originally proposed by Milner, Parrow and Walker in 1992 [38]. Although also called 'a calculus of mobile processes', no processes are actually moved around, just the identities (*names*) of the channels that processes use to communicate among themselves. It can be considered as an extension to CCS, where not only values can be passed around, but also channel names.

The polyadic π-calculus [37] is a generalized version of the basic π-calculus, extended to allow tuples and compound types to be sent along channels. Semantics is expressed in terms of both a reduction system and a version of labeled transitions called *commitment*.

The description of the polyadic π-calculus is beyond the scope of this paper; here we will only describe some of its features. If `ch` is a channel name, then `ch!(v).P` represents a process that sends value `v` along `ch` and then proceeds as process `P`. Conversely, `ch?(x).Q` is the process that waits for a value `v` to be received by channel `ch`, binds the variable `x` to the value received and then proceeds as `Q{v/x}`, where `{v/x}` indicates the substitution of the name `x` with `v` in the body of `Q`. Process communication is synchronous, and channel names can be sent and received as values. Special process `zero` represents inaction, internal actions (also called *silent* actions) are noted by `tau`, and the creation of a fresh name `z` in a process `R` is represented as `(^z)R`, where the scope of `z` is restricted to `R`. Processes can be composed in parallel with operator '`|`', and the summation operator '`+`' is used for specifying alternatives: `P + Q` may proceed to `P` or to `Q` (but not to both). There is also a *matching* operator, used for specifying conditional behavior. Thus, the process `[x=z]P` behaves as `P` if `x=z`, otherwise as `zero`.

We have also added some syntactic sugar to help component designers specify their behavior. The standard polyadic π-calculus does not provide for built-in data types, but they can be easily simulated; therefore we will use numbers and some basic data types, such as lists (with operators `<>`, `++` and `--` for list

creation, concatenation and difference). Additionally, we have enriched the π-calculus matching operator so that within the square brackets we can use any logical condition, that acts as a guard for the process specified after the brackets.

Modeling CORBA object interactions turned out to be easy and natural in the polyadic π-calculus, since object reference manipulation and client-server invocations have a very good semantic matching with the π-calculus. Therefore, it is easy to define a few techniques for modeling CORBA objects' interactions (for a complete list, please see [9]):

- Each object owns a channel, through which it receives method calls. This channel logically corresponds to the object reference.
- Together with every request, the calling object should include a channel name through which the called object will send the results. Although channels are bi-directional in the π-calculus, in this way request and reply channels can be kept separate to permit an object to accept several simultaneous calls, while using specific channels for replying.
- From the client's point of view, invocation of method `m` of an object whose reference is `ref` is modeled by one output action `ref!(m,(args),(r))`, where `m` is the name of the method, `args` is a tuple with its in and in-out parameters, and `r` is a tuple containing the return channel and, optionally, other reply channel names (for possible exceptions).
- Once the method has been served, the normal reply consists of a tuple sent by the called object through the return channel, containing the return value of the method, followed by the out and in-out parameters. Arguments are transmitted in the same order they were declared.
- The state-based behavior of the objects is modeled by recursive equations, where the various parts of the object state (i.e. the state variables we want to make visible to exhibit the object behavior) are parameters.

We have also added some syntactic sugar for the sake of clarity when writing the specifications of the objects' protocols:

- Method invocation `ref!(m,(args),(reply))` is abbreviated to the simpler form `ref!m(args,reply)`, or even to `ref!m(args)` if the reference and reply channels are the same.
- Similarly, on the server side we will write `ref?m(args,reply)...` instead of `ref?(meth,(args),(reply)).[meth='m']...` for accepting the invocation of method `m(args)`.

Protocols are defined using special construct '`protocol`', that consists of a name, followed by the protocol description in textual π-calculus enclosed between curly brackets. Each `protocol` description resides in a text file (with extension `.ptl`), and corresponds to one CORBA `interface` declaration, serving as the specification of its behavior —described as one or more π-calculus processes. Unless otherwise stated, the name given to the protocol description is used to identify the interface it relates to. Keeping protocol descriptions separated from object IDLs permits the addition of protocol information to CORBA object interfaces

in an incremental and independent manner. In this way, new CORBA tools, repositories and traders can be defined as extensions to the new ones, while keeping backwards compatibility with the current tools and applications that do not make use of this new protocol information.

6.3.2 *Extending the example specification*

Let us see here how to describe the behavior of the two components in our example using this approach. The first protocol description corresponds to the behavior of component `ITextModel`:

```
protocol ITextModel {
  ITextModel(ref,observers) =
    ref?length(rep) .
      (^v) rep(v) . ITextModel(ref,observers)
  + ref?charAt(p,rep) .
      (^c) rep!(c) . ITextModel(ref,observers)
  + ref?deleteCharAt(p,rep) .
    ( [observers!=<>] Notify(ref,observers,p,rep)
    + [observers=<>] rep!() . ITextModel(ref,observers)
    )
  + ref?register(b,rep) .
       rep!() . ITextModel(ref,observers++<b>)
  + ref?unregister(b,rep) .
       rep!() . ITextModel(ref,observers--<b>) ;

  Notify(ref,observers,p,rep) = // Notification process
    [observers=<b>++others]
      b!deleteNotif(p) . b?() . Notify(ref,others,p,rep)
  + [observers=<>]
      rep!() . ITextModel(ref,observers)
};
```

This description consists of a main π-calculus process with two arguments: the name of the channel that the object will use (i.e. the object reference), and a list with the registered observers. This second argument is used to store the internal state of the object, and is updated by operations `register()` and `unregister()`.

The process starts by reading from the reference channel, and once a tuple has been read, it is checked against the implemented servies. We can see how this specification does not care about the concrete values returned by the methods, just concentrates on their number and types, and the order in which messages are exchanged. In case a character is deleted and the list of observers is not empty, a `Notify` process is started, which iterates over that list notifying the observers. Once the list is over, the process replies to the client and behaves again as an `ITextModel`.

It is important to note that this is a *possible* behavior. Here we have used a list and notified the observers according to the order in which they appear in it

(the order in which they registered). Of course, other behaviors could have been defined using this notation. For instance, the order could have been unspecified (and hence left open to the implementor), or the notification process could have been done in parallel with the main process.

On the other hand, the behavior of `ITextObserver` could be described as follows:

```
protocol ITextObserver {
   ITextObserver(ref,itmodel) =
        itmodel!register(ref) . itmodel?() . Observe(ref,itmodel) ;
   Observe(ref,itmodel) =
        ref?deleteNotif(p,rep) . tau . rep() .
        ( tau . Observe(ref,itmodel)
        + tau . itmodel!unregister(ref) . itmodel?() . zero )
};
```

As we can see, it first registers with an `ITextModel` component, whose reference is the one given by the second argument of the main process. Once the observer gets the reply it behaves as process `Observe`, that waits for character delete notifications until it (somehow) decides to unregister from the `ITextModel` component and then quit. This internal decision is modeled by a combination of the summation operator and `tau`'s actions.

One of the benefits of having this sort of specifications is that they are able to precisely describe the interactions among components. Thus, we can prove in this case that the order in which observers receive notifications is the same in which the characters are deleted (the behavior of `ITextModel` is sequential: until it does not notify all observers it does not process any other method). And also that no reentrance or mutual recursion occurs: an observer obeying protocol `ITextObserver` will not send any request to any other object during the processing of method `deleteNotif()`. In this way, we know that an observer will not register any other observer, or that it will not delete any character. Those were two of the issues that potentially could cause problems in the system.

6.3.3 *Interoperability checks with protocol information*

Once we have enriched IDLs with protocol information, the next question is about the sort of tests that can be carried out with it, the moment in which these tests can be done, and the mechanisms required for those purposes.

We will distinguish between static and dynamic checks. The first ones are carried out during the design time of the applications, based just on the description of their constituent components and the binds among them (the architecture of the application). In these environments we could use the protocol descriptions of the components for carrying out very interesting checks, such as high level reasoning about the components (e.g. compatibility and substitutability tests), or proving safety and liveness properties of applications (e.g. absence of deadlocks). Please note that these are the sort of checks that are commonly carried

out by software architects using their ADLs (e.g. Wright [2], ACME [19], or LEDA [10]).

On the other hand, there are many situations in which protocol compatibility has to be checked at run time. Typical cases are the applications developed in open and *independently extensible* systems [44], in which the evolution of the system and its components is unpredictable: new components may suddenly appear or disappear, while others are replaced without previous notification. The Internet is probably the most well-known example of those systems.

Component compatibility and some safety properties of applications (e.g., absence of deadlocks) are among the tests that can be carried out during run-time. One way to implement this sort of tests is by intercepting messages and verifying their correctness with regard to the services offered and the state of the destination component (using for instance the CORBA interceptors [42], an special kind of object *filters*). These tests can be very useful for system debugging, and also provide an exception handling mechanism when integrating systems from components and applications that were not designed for *open* environments, and hence do not offer any support for handling the new situations that occur in those systems.

Those tests are so important because they are the ones that service traders and brokers need for searching and matching components in software repositories, using the abstract specifications of the components. Furthermore, they can also be used for guaranteeing that the composed application will respect the specification of the target application, and to prove its correctness.

7 Open problems: the way ahead

So far we have outlined some of the current proposals for dealing with component interoperability at various levels, and some of their problems and limitations. However, apart from those issues, there are many other topics related to component interoperability that require further investigation, and that we will try to briefly summarize here. From our personal viewpoint they constitute some of the main research areas still open within the component interoperability field.

7.1 Achieving full component interoperability

As we mentioned at the beginning of this chapter, component interoperability has been traditionally studied from two main points of view only —signatures and operational semantics—, but it goes far beyond them. One of the main problems is that the "semantic" level is too broad and fuzzy. It should cover not only operational semantics and behavioral specs of components, but also agreements on names, context-sensitive information, agreements on concepts (ontologies) —which moves names agreements one level up in the meta-data chain—, non-functional requirements, and so forth [25]. Moreover, there is not even a consensus on just what that set of issues includes. Each person seems to have a clear notion of what semantics is, but it does not agree with the notion that

other people have. The sum total of this is that semantics is a broad, fuzzy concept at the present time (D. Hyberston, in [46]).

One of our main challenges is therefore to develop an understanding of each of those factors, one at a time, and start factoring them out of the big and fuzzy (but shrinking) "semantics" bin, define it precisely, and give it a name. At the present time, perhaps we could say that we understand signature well, and are gaining a pretty good understanding of protocol and operational semantics, but the rest of "semantics" is still in the fuzzy bin. At the end, when we fully understand semantics, the fuzzy bin is empty, and we will have a clearly understood set of factors or elements that constitute semantics. Tasks for each of those "factors" include defining languages for expressing them, notations for reasoning about them, "weavers" for composing them, etc.

7.2 Mixtures of notations (Aspect-Oriented Specifications?)

Usually syntactic and (operational) semantic information are all mixed within a component specification. Now we are talking of identifying other aspects of their behavior (e.g. protocols), trying to separate different concerns. Of course, mixing all aspects within one component specification will make it too large, complex, and brittle to be of practical interest.

It is important to be able to specify each aspect of the specification in a *modular* way, so different protocols can be associated to the same (functional) services, or that separate specification aspects can be composed to meet complex and changing requirements.

This could also help alleviating the specification matching checks. Each specification aspect has its own complexity, so we could start with the most efficient tests (i.e. signature matching) that will allow to easily discard many potential candidates, and leave the most complex checks (the semantic ones) just for the few remaining components that have passed all the previous filters. This process has been successfully used just for signature and algebraic specification, as reported in [22]. We propose to adopt it for all the remaining levels of a system specification (no matter how many there are), hence improving the specification matching and compatibility checks.

There are of course many difficult problems involved in the idea of separating different specifications levels. For instance, how to deal with inconsistencies, or how to marry the different underlying logics of each notation. Formal interoperability [33] is a recent approach that tries to address those issues.

7.3 Incorporating business requirements

Perhaps one of the most important issues required for the description of the "full" semantics of components is how to incorporate business requirements into system and component specifications.

The need to understand and specify business and system semantics in a precise and explicit manner, independently of any (possible) realization, has

been recognized for a while. Some progress has been made in this area, both in academia and in industry. However, in too many cases only lip service to these ideas has been provided, and as a result the systems we build or buy are all too often not what they are supposed to be. In this sense, Mehmet Aksit [46] has mentioned three key issues involved in the process of dealing with business requirements in component-based environments: (a) incorporating problem domain knowledge into the component descriptions; (b) assuring component relevancy within certain business scenarios; and (c) implementing synthesis processes that allow the search of solutions from the solution domain. With them we could develop methodologies for synthesizing formal specs from the users' general requirements, using knowledge from both the problem and the solution domains.

7.4 Enhancing current notations for capturing semantic information

Related to the previous topic, we cannot ignore the current situation, in which formal description techniques are far from being widely accepted and utilized in real industrial environments. Currently, type checking (based just on signature specifications) is the only practical, useful and accepted way of formal analysis. Furthermore, type definition is the practical limit which real users are willing to specify and have reasonable chance of getting right (J. Wileden, in [46]).

We could mention many different reasons for that. For instance, semantic specifications are very difficult to write, and the computational complexity of their tests makes them impractical in most situations. And protocol specifications are seldom really useful or sufficiently precise. Besides, formal models (even types) always miss the aspects that turn out to matter in the real world, since they do not allow to capture many of the non-functional requirements involved in any real application. Other important issue is about users, and the fact that we have to communicate with them in ways they can understand; and formal description techniques may be useless for them.

Therefore, another challenge ahead is devising languages and notations for capturing semantic information in such way that users and developers can communicate each other, and that can also be used for system designers and architects for proving properties of the composed applications. There is a compromise here about how precise and formal they should be. The more formal, the less chances we have for users to use them. The less formal, the less chances we have to avoid ambiguities and vagueness in specifications.

Two initiatives are gaining wide acceptance. On one hand, ISO and ITU-T have been working on a joint standardization effort under the heading of Open Distributed Processing (ODP). The goal is to define a reference model to integrate a wide range of future ODP standards for distributed systems and maintain consistency among them. The reference model (known as RM-ODP, Reference Model - Open Distributed Processing) provides the coordination framework for ODP standards, creating an infrastructure within which support of distribution, interworking and portability can be integrated. On the other hand, UML has become a defacto standard notation for modeling systems. However, it lacks some

formal basis for reasoning about the specifications produced, and for providing consistencies among the different notations. Precise UML (pUML) is the name of a group that is actively working on providing formal basis to the standard notation (http://www.cs.york.ac.uk/puml).

7.5 Enhancing traders and brokers

We currently live in a world in which often users rely on human intermediaries to sort things out. However, with the rapid development of E-commerce and the Internet there could be no human intermediaries any longer. Most of the interoperability tests will have to be done by computers due to the vast amount of information available, possibly leaving to the system designer just a few final decisions. Software traders and brokers will be the ones in charge of those tasks, and will have to know about component capabilities and carry out component interoperability tests.

Therefore, we not only need to explore new notations for expressing more and more properties of components and to reason about them, but also we need to decide where to store this information, and the sort of mechanisms that will allow traders to make effective use of it.

A promising paradigm for enabling all this is *reflection*, a mechanism for capturing, observing and modifying the components' meta-information. For instance, CORBA's Meta-Object facility (MOF) is the natural mechanism that CORBA offers for storing and retrieving model and semantic information about its objects. Standardizing this facility, improving its general availability, and enhancing current traders and brokers to make effective use of it seems like a very promising way to go in component-based development environments.

7.6 Mediation

The concept of mediation [48] appears when we want to get components to work together when they are functionally compatible, but not necessarily signature and/or protocol compatible. By functionally compatible we mean that, at high enough level of abstraction, the service provided by one component is equivalent to the service required by the other component.

The main question here is whether it is possible to build some extra components that *adapt* their interfaces, bridging their incompatibilities. Those extra components are usually called *adaptors* or *mediators*, and their automated construction right from the description of the interfaces of the original components is a difficult problem (cf. [49]).

Mediation is a crucial issue in CBSE, since independently developed components are seldom reusable as they are. Therefore we need to go not only for more powerful searching and matching processes, but also for automated adaptation of components.

7.7 Many-to-one substitutability

Another limitation of current approaches for dealing with component substitutability is that they are defined on a one-to-one basis, studying when a component can be replaced by *just* another component. Searching for *several* components that can replace a given one is a difficult task, since it may introduce many different problems (such as service overlaps, gaps, or the need of addtional services). A similar situation happens when components may offer (and require) multiple interfaces. Again, traditional approaches fail to provide a solution for those cases.

7.8 Conformance to specifications

The last issue that we will discuss here is about how to test that a given implementation of an object conforms to a given specification of its behavior. There is no problem at the signature level, where it is a matter of checking that all methods defined in the interface are actually implemented by the object. However, it is a completely different situation at the protocol or semantic levels. In those cases we need to check that the actual implementation conforms to the specified behavior, but this is usually impossible: we are dealing with black-box components, whose code is unaccessible. However, there is one possible way to deal with this problem at run-time using the aforementioned interceptors. They were used for checking that incoming messages to an object were valid with regard to its current state. But they can also be used for checking that the object's behavior is valid with regard to the one it is supposed to implement. In this sense, those filters can be used to *enforce* behaviors, in a similar way to Minsky's Law Governed Architectures [39].

On the other hand, checking the conformance to specifications when the actual code is available is a different issue. Here we have used the black-box approach, assuming that this is how we will find our COTS components. Studying the conformance to protocol specifications under the white-box or grey-box [8] approaches is the subject of separate research.

8 Concluding remarks

As software technology becomes a core part of business enterprises in all market sectors, customers demand more flexible enterprise systems. This demand coincides with the increasing use of personal computers and today's easy access to local and global communication networks, which together provide an excellent infrastructure for building open distributed systems. However, the specific problems of those large systems are currently challenging the Software Engineering community, whose traditional methods and tools are finding difficulties for coping with the new requirements.

Interoperability is one of the major challenges in these new settings, particularly within component-based software development environments, an approach in which prefabricated reusable software components from independent

sources are assembled together to build applications. There are many aspects related to component interoperability, including syntactic agreements on method names, behavioral specifications of components, service access protocols, business domain knowledge and shared ontologies, negotiation of QoS and other non-functional properties, mediation, etc. So far the situation starts to be clear for some of those aspects, specially at the signature, protocol, and operational semantic levels, for which their possibilities and limitations seems to be widely known and acknowledged. However, much work remains to be done with the rest of the issues. Many problems still need to be solved before components can seamlessly interoperate and be easily composed to build applications. It is a long and winding road ahead, but we are now starting to be able to formulate the problems, which undoubtedly is the first step towards solving them.

References

1. ACM. ACM Risks Forum Digest. `http://catless.ncl.ac.uk/Risks`.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, July 1997.
3. P. America. Designing an object-oriented programming language with behavioral subtyping. In J. der Bakker, W. de roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, 1990 REX School/Workshop, Noordwijkerhout, The Netherlands*, number 489 in LNCS, pages 60–90. Springer-Verlag, 1991.
4. R. Bastide and O. Sy. Towards components that plug AND play. In A. Vallecillo, J. Hernández, and J. M. Troya, editors, *Proc. of the ECOOP'2000 Workshop on Object Interoperability (WOI'00)*, pages 3–12, June 2000.
5. R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag, 1999.
6. A. Borgida and P. Devanbu. Adding more "DL" to IDL: towards more knowledgeable component inter-operabiltity. In *Proc. of ICSE-21*, pages 378–387, Los Angeles, May 1999.
7. M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component? *Software–Concepts and Tools*, 19:49–56, 1998.
8. M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, Aug. 1999. `http://www.abo.fi/~mbuechi/publications/TR297.html`.
9. C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proc. of TOOLS Europe 2000*, pages 208–225, France, June 2000. IEEE Press.
10. C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In P. Donohoe, editor, *Software Architecture (Proc. of WICSA'99)*, pages 107–125. Kluwer Academic Publishers, Feb. 1999.
11. I. Cho, J. McGregor, and L. Krause. A protocol-based approach to specifying interoperability between objects. In *Proceedings of TOOLS'26*, pages 84–96. IEEE Press, 1998.
12. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. of the 1st International Workshop on Rewriting Logic*

and its Applications, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, California, September 1996. Elsevier. Available at `http://www.elsevier.nl/locate/entcs/volumen4.html`.
13. D. Coleman et al. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.
14. G. Denker et al. Specification and analysis of a reliable broadcasting protocol in Maude. In B. Hajek and R. S. Sreenivas, editors, *Proc. of the 37th Annual Allerton Conference on Communication, Control, and Computation*, University of Illinois, USA, September 1999.
15. G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, June 1998.
16. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 258–267, Berlin, Germany, 1996. IEEE Press.
17. M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.
18. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks.* John Wiley & Sons, 1999.
19. D. Garlan, R. T. Monroe, and D. Wile. ACME: An architectural interchange language. In *Proc. of the 19th IEEE International Conference on Software Engineering (ICSE'97)*, Boston, May 1997.
20. G. Genilloud. On the abstraction on objects, components, and interfaces. In H. Kilov, K. Baclawski, A. Thalassinidis, and K. P. Tyson, editors, *Proc. of the OOPSLA'99 Workshop on Behavioral Semantics*, pages 93–104, Nov. 1999.
21. J. Goguen. Hidden algebra for software engineering. *Australian Computer Science Communications*, 21(3):35–59, 1999.
22. J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins. Software component search. *Journal of Systems Integration*, 6:93–134, Sept. 1996.
23. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
24. J. Han. Semantic and usage packaging for software components. In A. Vallecillo, J. Hernández, and J. M. Troya, editors, *Proc. of the ECOOP'99 Workshop on Object Interoperability (WOI'99)*, pages 25–34, June 1999.
25. S. Heiler. Semantic interoperability. *ACM Comput. Surv.*, 27(2):271–273, June 1995.
26. B. Henderson-Sellers, R. Pradhan, C. Szyperski, A. Taivalsaari, and A. C. Wills. Are components objects? In *OOPSLA'99 Panel Discussions*, Nov. 1999.
27. T. Huckle. Collection of software bugs. `http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html`.
28. ITU-T. Message sequence charts. Rec. Z.120, ITU-T, 1994.
29. D. Konstantas. Interoperation of object oriented applications. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 69–95. Prentice-Hall, 1995.
30. D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.
31. G. T. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems.* Cambridge University Press, 2000.
32. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

33. J. Meseguer. Formal interoperability. In *Proc. of the 1998 Conference on Mathematics in Artificial Intelligence*, Florida, Jan. 1998. `http://rutcor.rutgers.edu/~amai/Proceedings.html`.
34. J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*, NATO Advanced Study Institute, Germany, 1998. Springer Verlag.
35. B. Meyer. *Object-Oriented Software Construction. 2nd Ed.* Series on Computer Science. Prentice Hall, 1997.
36. A. Mikhajlova. *Ensuring Correctness of Object and Component Systems.* PhD thesis, Åbo Akademi University, Oct. 1999.
37. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
38. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
39. N. Minsky and J. Leichter. Law-governed Linda as a coordination model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems*, number 924 in LNCS, pages 125–146. Springer-Verlag, 1995.
40. O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.
41. A. Ólafsson and D. Bryan. On the need for "required interfaces" of components. In *Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96*, pages 159–165. Dpunkt Verlag, 1996.
42. OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 2.3 edition, June 1999. `http://www.omg.org`.
43. OMG. *The CORBA Component Model.* Object Management Group, June 1999.
44. C. Szyperski. *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
45. C. Szyperski. Components and the way ahead. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 1–20. Cambridge University Press, 2000.
46. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP'2000 Workshop Reader*, number 1964 in LNCS, pages 256–269. Springer-Verlag, 2000.
47. P. Wegner. Interoperability. *ACM Comput. Surv.*, 28(1):285–287, Mar. 1996.
48. G. Wiederhold. Mediation in information systems. *ACM Comput. Surv.*, 27(2):265–267, June 1995.
49. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. Prog. Lang. Syst.*, 19(2):292–333, Mar. 1997.
50. A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170, Apr. 1995.
51. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.