Modeling Interface Definition Language Extensions

H.-Arno Jacobsen*
INRIA-Rocquencourt
Projet Caravel, B.P. 105
78153 Le Chesnay-Cedex, France
Arno.Jacobsen@inria.fr

Bernd J. Krämer
FernUniversität
Lehrstuhl für Datenverarbeitungstechnik
D-58084 Hagen, Germany
bernd.kraemer@fernuni-hagen.de

Abstract

Interface definition languages serve to specify module and interface names, as well as operation signatures. However, IDLs lack means to express aspects, such as synchronization constraints, pre- and post conditions, invariants, quality of service annotations, and real-time annotations. We develop a framework to specify the interface definition language and a given IDL extension in a combined modeling language based on XML. We show how this specification can serve to obtain tools to process the extended interface definition language. We study this approach along the lines of OMG IDL and the CORBA middleware platform. The specification of semantic aspects and the specification of IDL is based on XML Document Type Definitions.

Keywords: Interface Definition Language Extensions, CORBA middleware, eXtensible Markup Language, Document Type Definitions, eXtensible Style Language.

1: Introduction

Many IDLs such as CORBA IDL, ODL, DCE IDL, MIDL, and ASN.1 typically specify object interfaces in terms of module names, interface names, structured types, and signatures of operations. A signature defines the operation name, return type, argument modes and types, and possibly an exception type. This simplicity and generality ensures that IDL is applicable to a wide range of application domains and can be mapped to a large variety of implementation languages. But the price for this generality is that:

- semantic properties of application objects such as functional and dynamic behavior, timing requirements, synchronization constraints, security aspects, and quality of service requirements cannot be formally documented in the object interfaces;
- the separation of a distributed system's requirements and constraints into a number of concerns that can be addressed, analyzed, and evolved in isolation throughout the lifetime of an application is not supported;
- the automatic synthesis of code from specifications is limited to the generation of header files, skeleton, and stub code;
- only functional aspects of the object implementation (operation signatures) are documented at the interface level.

Compared with state–of–the–art specification and design languages such as UML with its structural, dynamic, and functional views on object specifications [3], IDLs are expressively weak. The consequence is a growing number of proposals for IDL extensions, application–specific IDL conventions, and supplements including, for example: real–time [27, 31], quality of service [34], behavioral [33] and synchronization constraint [18] annotations to IDL.

The author is supported by a postdoctoral fellowship awarded through the European Research Consortium for Informatics and Mathematics (ERCIM).

0-7695-0918-5/00 $10.00 © 2000 IEEE

Abstractly speaking, these extensions break through the abstractions provided by the middleware platform and allow users to specify aspects of an application normally hidden behind transparency mechanisms. A critical issue of all these proposals is the question of how they are implemented. Certainly, the simplest way is to wait for the proper extension of your preferred middleware standard and its implementation by a vendor. But such reference implementations may take time and applications exploiting such IDL extensions are not portable as long as they depend on proprietary platform extensions. Furthermore, it is rather unlikely that language constructs for specifying real–time requirements, synchronization constraints, functional or dynamic behavior will ever be included in future versions of OMG IDL.

To escape this trap, the approach described in [18, 15] proposes to include annotations as comments in IDL interface definitions and separately compile these annotations into code implementing corresponding sanity checks. In [15, 14] this idea has first been presented and experimentally validated. Moreover, in [14] we have developed a set of design patterns to integrate code fragments that implement extensions with the skeleton code generated by standard stub–compilers.

In this paper we further explore this idea by introducing a framework to specify the proposed extension and the interface definition language in a combined modeling language based on XML. We sketch the design of a tool to process the extended interface definition language. The contribution of this paper are therefore threefold: (1) a framework to model interface definition language extensions that extend beyond a single method invocation, (2) a specification of a document type definition (DTD) for OMG IDL and an exemplification of generic IDL extensions modeled as enhancements of this DTD, and (3) a design for a tool to process the extended service interfaces based on current XML processing technology.

The rest of this paper is organized as follows. Section 2 surveys and classifies interface definition language extensions. Section 3 develops the framework for specifying extensions and the IDL in a combined modeling language based on XML. Section 4 provides examples that demonstrate the modeling procedure.

2: Interface Language Extensions and Processing Schemes

In this section we give an overview of interface definition language extensions as they have been proposed in the literature, and evaluate alternative processing schemes. IDLs have been designed to provide interfaces to objects and components in an implementation language independent manner, with the objective of keeping the interface independent of the object implementation. The design of the interface definition language is also constraint by the set of possible target implementation languages, which are to be supported in the implementation environment. For OMG IDL, for instance, this set extends to implementation languages as diverse as C, C++, Java, Cobol, Eiffel, and Lisp. IDLs are therefore kept purely syntactical expressing operation signatures, defining names paces, and interfaces only. Most common IDLs lack any semantic capabilities to express behavioral annotations, quality of service attributes, interaction protocols, synchronization constraints, object grouping facilities, and miscellaneous extensions. This deficiency has lead to many IDL language extensions.

2.1: Interface Definition Language Extensions

Much work has focused on annotating IDL with behavioral extensions, such as pre- and post-conditions, invariants, abstract operation semantics, data integrity conditions, and Horn clauses [26, 25, 7]. The annotation of interfaces with real time constraints (e.g., priorities, deadlines, execution time) and quality of service attributes (e.g., required min/max bandwidth, allowed jitter, resource needs) have been widely proposed [27, 1, 31, 34]. Object interaction protocols are IDL annotations that describe sequences of legal operation invocations between interfaces or for one interface. They provide hints to clients on "how" to use an interface and give rise to static and dynamic checking of a caller-callee interaction. The annotation of IDL with path-expressions to express order of invocation requirements is presented in [29]. Alternatives are the management of server execution with Petri-nets [10], the features of TINA ODL [23, 4] to express order of execution, IPDL (Interaction Protocol Definition Language) [6], and regular types

for active objects [20]. The annotation of IDL interfaces with synchronization constraints to guarantee mutual exclusion of operations and to protect shared resources has been explored in detail [9, 28, 18, 15]. Other IDL extensions include object co-location constraints [12], coordination constraints [12], data parallelism [17], security annotations [11], and component definition language extensions [19, 21]. In [14] we provide a first taxonomy of interface definition language extensions and present a more exhaustive description.

2.2: Processing IDL

The interface specification is processed by a *stub compiler* that generates support code for client and server side to enable communication between components across address space and machine boundaries. Client–stubs manage packaging and marshaling on the client side, and server–skeletons implement these functions on the server side. This is the general approach taken by CORBA, DCE, DCOM, and many RPC systems (e.g., ONC RPC).

For the implementation of IDL extensions the following solutions have been presented in the literature. Many of the above listed language extensions directly modify the IDL syntax by including additional keywords (e.g., [26, 32]). Such an approach is tied to the particular platform used and leads to non–portable application code. Product lock–in is the inevitable result. In previous work [18, 15] we have proposed to extend IDL via comments, i.e., place IDL annotations in comments. This has the advantage that extension unaware IDL compilers are still able to process the specification, whereas extension aware compilers may fully exploit the annotations. A third alternative relies on programming conventions to implement the IDL extension. Interface attributes and operation arguments are used to encode extensions. For instance, the specification of real time constraints have been implemented in this manner (cf. [27, 31]). This approach is based on naming conventions that have to be adhered to by the programmer. If these conventions are to be enforced automatically, a language processor must be provided. Moreover, extensions implementing support code must also be integrated with the generated stub code. This solution is thus constrained by the same portability argument as the former approach.

Table 1 summarizes the IDL extensions discussed and classifies them according to the degree of support they require from the infrastructure in order to be implemented. In Jacobsen and Krämer [14] we have developed the *extension adapter design pattern* to integrate the IDL extension supporting code with the code generated from the stub-compiler (i.e., with server skeletons)[1]. This approach supports all IDL extensions that operate on a state across one or multiple operation invocations. However, some of the extensions, such as quality of service annotations, require support from lower middleware layers in order to be implemented. As this support is not available in standard platforms, a portable approach for these extensions is not possible.

A recent paper by Beugnard *et al.* [2] presents an abstract view on making distributed object implementations contract aware, i.e., integrate various IDL extensions in their interfaces. Beugnard *et al.* name four levels of contracts: syntactic level (interfaces and signatures), behavioral level (pre, post conditions and invariants), synchronization level, and quality of service level. While this view is very elegant, the question of *how* the various extensions are to be modeled and implemented given standard middleware technology remains open. Furthermore, a method to generically specify the extensions is also deferred to future work. In our previous work [15, 14] we investigated techniques to address the implementation problem of IDL extensions. In this paper we address the issue of modeling IDL extensions in a uniform manner and propose a framework to process these extensions automatically.

3: A Framework to Process Extended IDL

In this section we develop a framework to specify an interface definition language and its extensions in a unified methodology. We show how to combine these specifications and motivate how to synthesize

[1] So far, we have focused on extensions that are fully realizable by adding support code on the server-side. However, a similar pattern may be applied to integrate support code for client-side processing.

Aspect	Example	Implementation
behavior & semantic	pre, post condition, invariant, integrity constraint, service offer (Horn clauses)	state over method invocation
quality of service	real–time: priority, deadline, execution time; network QoS: min / max bandwidth & delay, jitter	state over method invocation and network layer access (QoS, RT required)
object interaction	path expressions, sequence of admissible operation, call protocol	state over method invocation
synchronization	mutual exclusion, capacity constraint, fairness condition	state over method invocation
security	access control, authentication information, required interaction protocol	state over method invocation
others	co–location, coordination, data parallelism, component definition, application domain extensions	—

Table 1. IDL extension overview. (RT - Real Time, QoS - Quality of Service.)

processing tools for the extended language. This decoupling of base language and extension allows us to add "new" language features a posteriori. We use XML to model the interface definition language and its extensions, other modeling methodology would work equally well. However, the use of XML has the following additional advantages:

1. XML has become a popular language to describe structured documents and many tools have become available to process XML specifications (e.g., the XML4J API from IBM, the LotusXSL translator, XT, lark, and JUMBO etc.).
2. The integration of XML technology with middleware platforms is an emerging paradigm with tool support becoming rapidly available [30] (e.g., IDL2XML [24]).
3. Most major database products support XML. It is therefore straightforward to create a repository for the extended interface definition language. Most middleware products implement their own repositories, which cannot be extended to also manage the proposed IDL extensions.

XML captures document structure, it is therefore well suited to define the syntactic nature of an interface definition language. XML does not convey semantic information about the content of the defined tags. The current XML model does not allow its users to associate a type with the content of tags. Any type checking has to be done by the XML processing application. This drawback is addressed by the emerging XML schema standard [8].

We use OMG IDL and the features of the CORBA middleware platform to illustrate our ideas. However, our approach extends to other middleware platforms, such as DCOM (Distributed Component Model) with the MIDL (Microsoft Interface Definition Language). In the following we develop a DTD for OMG IDL, show how to model IDL extensions with DTDs, and demonstrate how to jointly process them to obtain XIDL (eXtended IDL) syntax processing tools.

3.1: A DTD for the Interface Definition Language

The *Extensible Markup Language* (XML) defined by the WWW Consortium (W3C) [5] is a data format for structured document interchange on the Web. In contrast to HTML it allows the user to *extend* the language features to process many different classes of documents. XML may thus be used to define *customized markup languages* for document processing on the Web.

In the following we develop a data type definition (DTD) for the OMG IDL compliant to the OMG

```
<! ELEMENT SIGNATURE  ( ARGUMENT* ( RAISES? ) ( CONTEXT? ) ) >
<! ATTLIST SIGNATURE
          mode    ( oneway | twoway )  "twoway"
          name    CDATA   #REQUIRED
          rtype   CDATA   #REQUIRED  >
<! ELEMENT ARGUMENT   EMPTY >
<! ATTLIST ARGUMENT
          mode     ( in | out | inout ) #REQUIRED
          name     CDATA                #REQUIRED
          type     CDATA                #REQUIRED >

<! ELEMENT RAISES ( EXCEPTION_TYPE+ )>
<! ELEMENT EXCEPTION_TYPE  EMPTY >
<! ATTLIST EXCEPTION_TYPE
           exception     CDATA          #REQUIRED >
```

Figure 1. DTD elements defining an IDL operation signature.

CORBA standard [22]. We motivate key parts of the DTD, a complete specification can be found in [16].

Figure 1 shows DTD elements defining an IDL operation signature. An IDL operation consists of a return type (`rtype`), an operation name (`name`), a qualifier (`( oneway | twoway )`), a parameter list (`ARGUMENT*`), an optional raises clause (`RAISES?`), and an optional context clause (`CONTEXT?`). Note, that the identifier "`twoway`" is not part of standard OMG IDL, but has to be provided in the XML specification of OMG IDL to distinguish it from oneway operations. In standard OMG IDL only oneway operations are qualified with the identifier "`oneway`", with twoway being the default[2].

Figure 2 shows DTD elements that specify OMG IDL modules consisting of constructed types, constant declarations, exception declarations, forward references, and interface definitions. The DTD elements show interface definitions only (see [16] for the complete specification of other elements). An OMG IDL server interface consists primarily of attributes and operation signatures, as well as zero or more inheritance clauses.

Figure 3 shows an account interface specified in the OMG interface definition language. Figure 3 presents a specification of this interface according to the XIDL DTD elements defined here. The specification of on object interface in XML is not different from the specification of the interface in IDL. The XML specification can be easily processed to yield the tag-free interface specification in IDL. We model IDL extensions also with DTDs by extending the DTD of bare IDL with elements modeling the extension. Several examples are provided in Section 4. We now motivate how to process the IDL to obtain an XIDL (eXtended IDL) processor.

3.2: The eXtended IDL Processor

In this section we sketch the design of an XIDL processor. This processor is based on tools for processing XSL (eXtensible Style Language)[3] and XML documents. We denote the extended interface definition language by XIDL and standard IDL simply by IDL. We model the interface definition language with a document type definition, `IDL.dtd` (cf. Section 3.1). The extension of the interface definition language is modeled with a separate document type definition `EXtension.dtd` that substitutes and extends the elements in `IDL.dtd` that are enhanced (e.g., operation signatures and interfaces, cf. Section 4). Merging the document type definition of the interface definition language and the extension (i.e., `IDL.dtd` and

[2]Oneway invocations denote non–blocking operations without return value, upon execution a client resumes immediately.

[3]See: http://www.w3.org/Style/XSL/

```
<! ELEMENT MODULE  ( DECLARATION | INTERFACE )* >
<! ATTLIST MODULE
          name    CDATA   #REQUIRED >

<! ELEMENT INTERFACE  ( INHERITANCE*, ( DECLARATION | ATTRIBUTE | SIGNATURE )* ) >
<! ATTLIST INTERFACE
          name    CDATA   #REQUIRED >
<! ELEMENT INHERITANCE EMPTY >
<! ATTLIST INHERITANCE EMPTY
          ancestor   CDATA   #REQUIRED >

<! ELEMENT ATTRIBUTE
<! ATTLIST ATTRIBUTE
         mode    ( readonly  |  readwrite )  "readwrite"
         name    CDATA   #REQUIRED
         type    CDATA   #REQUIRED  >
```

Figure 2. DTD elements specifying OMG IDL modules et al.

```
<IDL>
  <comment> Account Example </comment>
  <module name="Online_Account">

    <exception name="OverDraft">
      <member name="message" type="STRING"/>
    </exception>

    <interface name="SavingsAccount">
    <inheritance ancestor="Account"/>

      <attribute mode="readonly" name="balance" type"Money"/>

      <signature mode="twoway" rtype="void" name="deposit" >
      <argument mode="in" name="amount" type="Money"/>
      </signature>

      <signature mode="twoway" rtype="boolean" name="withdraw" >
      <argument mode="in" name="amount" type="Money"/>
      <raises>
        <exception name="OverDraft" />
      </raises>
      </signature>

    </interface>
  </module>
</idl>
```

Figure 3. XML specification of an account interface according to the DTDs defined.

File	Description
IDL.dtd	DTD for bare interface definition language
<interface>.xml	defines interfaces in bare IDL conformant to the IDL DTD (i.e., IDL.dtd)
EXtension.dtd	DTD for the extension of IDL (e.g., with synchronization constraints)
XIDL.dtd	DTD resulting from merging IDL.dtd and EXtension.dtd , i.e., entire DTD of the extended interface definition language
<Xinterface>.xml	defines interfaces in XIDL according to the XIDL DTD (i.e., XIDL.dtd)

Table 2. Different specification files to model IDL and extensions.

EXtension.dtd, respectively) constitutes the DTD of the extended interface definition language, here denoted by XIDL.dtd.

The XIDL processing tool is based on expressing the transformation of the input language (i.e., XIDL) to an output format (e.g., Java stubs and skeletons) via XSL transformation rules. A tool supporting XSL is, for instance, XSLT (Lotus) from IBM. Table 2 summarizes the individual specifications and Figure 4 depicts the processing steps schematically.

The XSL / XML processor validates an input file (<Xinterface>.xml – a user defined service interface that takes advantage of the extensions, e.g., synchronization constraints) – against the DTD (XIDL.dtd) defined. It generates stubs, skeletons, and support code, implementing the extensions.

The support code implementing the IDL extensions are integrated with the stub and skeleton code according to the *extension adapter design pattern* (cf. Jacobsen and Krämer [14] for details about this design pattern.) Stubs and skeletons must either be based on the dynamic invocation and the dynamic skeleton interface or exploit the Java portability layer as the stub / skeleton ORB interfaces are not specified in the CORBA standard (cf. Jacobsen [13] for implications and limitations).

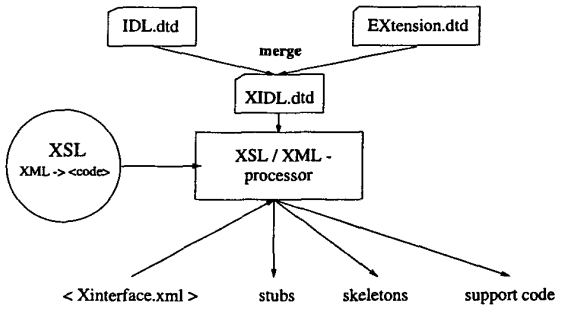

Figure 4. XML DTD processing stages to synthesize XIDL processor.

4: Examples

In this section we describe various examples to illustrate the different IDL extensions on concrete application scenarios. As base scenario we use variants of the Account interface introduced in the previous

```
interface Account {

  Money balance;
  // { invariant:  balance >= OverDraftLimit }

  // { pre: amount.value > 0 }
  void deposit( in Money amount );
  // { post: @balance.value == balance.value + amount.value}
  //   '@' refers to value of balance after method execution
  ...
}
```

Figure 5. Adding `pre` and `post` conditions to an IDL interface.

section (cf. Figure 3). We demonstrate how to model the extensions with XML DTDs. The actual IDL extension is described in an intuitive form, added as comments to standard IDL. Note, that this serves merely to describe the extension. In lieu of this intuitive description, an XML document valid according to the underlying DTD is processed. We avoid presenting the XML conforming to the DTDs, which models the actual extended interface definition language, to simplify the overall presentation. Figure 3 shows an example of a service interface defined in XML.

4.1: Behavior and Semantic Annotations

Behavior and semantic annotations comprise `pre` and `post` conditions, as well as `class invariants`. Figure 5 shows how these conditions may be applied to ensure the integrity of operation input and interface attribute values. A DTD to model these behavioral annotations extends the `SIGNATURE` and the `INTERFACE` elements of the `IDL.dtd`. The conditions are modeled as binary relations between two variables. Figure 6 shows DTD elements for the annotated IDL interface also shown in Figure 5.

4.2: Synchronization Constraints

The access to the account object must be serialized to protect the internal shared state of the object. The administrative action of calculating the interest on the account, for instance, cannot run concurrently to a deposit operation. Figure 7 shows an IDL interface annotated with synchronization constraints. This extends the `SIGNATURE` element of the `IDL.dtd` (cf. Figure 7).

4.3: Object Interaction Protocol

Object interaction protocols define admissible operation invocation chains on object interfaces. In an online banking scenario, where the online system offers an API against which a client can write her access code, such protocols can ensure correct operation execution. A client has to authenticate herself, for instance, before any other operation can be executed on the remote system. Figure 8 shows an IDL interface annotated with object interaction protocols. This extends the `INTERFACE` element of the `IDL.dtd` (cf. Figure 8).

5: Conclusion

In this paper we have introduced a framework to specify IDL and IDL extensions in a combined modeling language based on XML document type definitions. We have demonstrated how this framework

```
<! ELEMENT SIGNATURE  ( ( PRE? ) ARGUMENT* ( RAISES? ) ( CONTEXT? ) ( POST? ) ) >
<! ATTLIST SIGNATURE ...

<! ELEMENT PRE EMPTY >
<! ATTLIST PRE
          mode    ( lt | gt | le | ge | eq ) #REQUIRED
          lhand    CDATA   #REQUIRED
          rhand    CDATA   #REQUIRED  >

<! ELEMENT POST EMPTY >
<! ATTLIST POST
          mode    ( lt | gt | le | ge | eq ) #REQUIRED
          lhand    CDATA   #REQUIRED
          rhand    CDATA   #REQUIRED  >

...

<! ELEMENT INTERFACE  ( INHERITANCE*, ( DECLARATION |
                        ATTRIBUTE | SIGNATURE | INV)* ) >
<! ATTLIST INTERFACE
           name    CDATA   #REQUIRED >

<! ELEMENT INV EMPTY >
<! ATTLIST INV
          mode    ( lt | gt | le | ge | eq ) #REQUIRED
          lhand    CDATA   #REQUIRED
          rhand    CDATA   #REQUIRED  >

...
```

Figure 6. XML DTD capturing `pre-` and `post-` conditions

```
interface Admin_Account : Account {
  ...
  // { mutex(deposit, interest) }
  void interest();

}
```

```
<! ELEMENT INTERFACE
      ( INHERITANCE*, ( DECLARATION |
                        ATTRIBUTE |
                        SIGNATURE | MUTEX)* ) >
...

<! ELEMENT MUTEX      ( ( OPERATION OPERATION )
                          OPERATION* ) >
<! ELEMENT OPERATION EMPTY >
<! ATTLIST OPERATION
        name     CDATA   #REQUIRED  >
```

Figure 7. Synchronization constraints in an IDL interface and XML DTD elements to specify constraints as IDL extensions

```
interface Online_Account : Account {
  ...
  //{ authenticate;
  //  { deposit | withdraw }; signoff }
  void authenticate( in String pw )
     raises Access_Denied;
  void signoff();
}
```

```
<! ELEMENT INTERFACE ( INHERITANCE*,
                       ( DECLARATION |
                         ATTRIBUTE   |
                         SIGNATURE   |
                            PROTOCOL )* ) >
<! ELEMENT PROTOCOL EMPTY >
<! ATTLIST PROTOCOL
           name     CDATA   #REQUIRED >
```

Figure 8. Object interaction protocols in IDL interfaces and XML DTD elements to specify protocols as IDL extension.

can be used to model several different IDL extensions. Moreover, we have sketched a design of a processor for eXtensible IDL (XIDL).

We are currently working on an implementation of this framework based on standard XSL processing tools (i.e., LotusXSLT). As outlined above, our implementation will only be able to exploit the less performant dynamic interfaces of current ORBs, or the Java portability interface (only available for Java language mapping conformant ORBs). This is due to limitations in openness and extensibility of the CORBA standard (for details on this issue see Jacobsen [13].) We aim at a generic tool that supports several different ORBs, and is not specific to one particular CORBA implementation.

As the XML Schema[4] specification becomes fully available, we will extend our DTD for OMG IDL to define and include types to improve type checking of service interfaces defined with XML.

References

[1] C. Becker and K. Geihs. Generic QoS specifications for CORBA. In *Kommunikation in Verteilten Systemen (KIVS'99)*, pages 184–195. Springer-Verlag, 1999.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[3] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide.* The Addison-Wesley Object Technology Series. Addison-Wesley, 1998.

[4] M. Born, M. Hoffmann, and M. Winkler. Evaluation and improvement of mapping rules from CORBA-IDL and TINA-ODL to SDL-92. (GMD Focus Internal Report).

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (editors). Extensible markup language (XML) 1.0. Technical report, W3C, Feb. 1998.

[6] B. Bukowski. Interaction protocols: Typing of object interactions in frameworks. Technical report TR B 96–10, Freie Universität, Berlin, 1996.

[7] C. Della, T. Cicalese, and S. Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, July 1999.

[8] David C. Fallside. XML Schema. Technical report, W3C, 2000. (working draft) URL: http://www.w3.org/XML/Schema.html.

[9] S. Frølund. *Coordinating Distributed Objects.* MIT Press, 1996.

[10] H. Gruender and K. Geihs. On the object-oriented modelling of distributed workflow applications. In *3. Internationale Tagung Wirtschaftsinformatik (WI)*, Berlin, Germany, 1997.

[11] D. Hagimont, J. Mossire, X. Rousset de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 282–289, May 1996.

[12] O. Holder, L. Ben-Schaul, and H. Gazir. Dynamic layout of distributed applications in Fargo. Technical report, Technion – Israel Institute of Technology, Aug 1998.

[4] See http://www.w3.org/XML/Schema