Towards a Synthetical Approach for the Construction of Distributed Applications

Martin Zimmermann, Magdalena Feldhoffer, Oswald Drobnik
University of Frankfurt, Institute for Telematics

Abstract

The paper introduces a new approach which supports the designer of a distributed application in the specification and implementation of different aspects namely application-, management- and communication-oriented aspects. Since these aspects are interrelated in many ways, we will also address different kinds of relationships. In particular, we will sketch out how the management and communication requirements of a distributed application can be derived from the application objectives. Moreover, we will illustrate an object-oriented realization of these different aspects and discuss several tools supporting the specification and implementation steps.

1. Introduction and model

Current approaches in the developement of distributed applications tend to regard either the communication-oriented aspects or the application-oriented issues. However, our experiences in the development of distributed applications have shown that this view is too restrictive. A future support environment should allow the integration of different aspects to enable a flexible and synthetical configuration of distributed applications.

New models such as Open Distributed Processing (ODP) [6] intend to address distributed applications from the application point of view by masking distribution aspects. Meanwhile, there are numerous activities in standardization bodies, in research projects and in the commercial sector in order to define a general framework for distributed application development, such as DAF [2], SE-ODP [4], ANSA [1], REX [9] and OSF-DCE [11].

The paper introduces a new approach which supports the designer of a distributed application in the specification and implementation of different aspects, namely *application-, management-* and *communication-oriented* aspects. From the application-oriented perspective a distributed application is composed of a set of interacting *application components* providing the required application functionality. Interfaces are the points at which an application component interacts with other components. The application behaviour of an application component is defined by the interactions at its *application interfaces*. In order to enable management activities, each application component must provide *management interfaces* which define operations for the control, maintenance and monitoring of application components. In particular, management-oriented aspects involve establishing and termination of a distributed application.

The communication-oriented aspects are integrated by the concept of *communication contexts*, which describe the communication requirements of an application component.

In addition to application components, our structural model distinguishes *management components* providing only management functions. In a configuration step the distributed application is built by binding the interfaces of the application and management components according to the given runtime environment (see Figure 1).

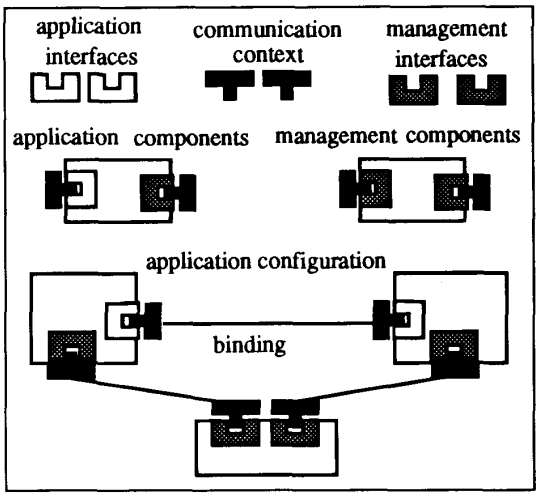

Figure 1: Construction steps for distributed applications

The flexibility of this structural model forms a solid platform for the development of a methodology to construct and engineer distributed applications. Basic parts of such a methodology are consistent *specification* and *implementation* techniques that take into account the different kinds of relationships between the application-, management- and communication-oriented aspects. Im comparison to the ODP model our approach covers the computational and engineering viewpoints.

0-8186-2755-7/92 $03.00 © 1992 IEEE

2. Specification of distributed applications

2.1 Application behaviour

Interface Specification Language (ISL)

Our interface concept is based on a bidirectional interaction specification and describes what operation each of a pair of components could request the other to perform. For a *symmetric* interface all operations can be invoked as well as performed whereas for an *asymmetric* interface we have to specify for each operation whether it will be invoked by the supplier or by the consumer side. The specific behaviour at an interface, i.e. the determination whether an application component is acting as a consumer and / or a supplier, is defined when we describe how a component is constructed from a set of interfaces. Moreover, each operation specification could be decorated with attributes, such as an attribute declaring an operation to be performed atomically.

The behaviour at an interface, i.e. the allowable operation invocations, can be restricted by the definition of an *access protocol* in terms of *sequencing rules* and *synchronisation properties*. Sequencing rules are useful for the specification of ordering constraints for operation invocations and can be expressed by extented path expressions (e.g. op1 < op2 for sequential invocation, op1 || op2 for parallel invocation). Synchronisation properties are necessary if the interactions of two or more components have to be coordinated. E.g. in a client-server cooperation, the clients may need exclusive access to a server for specific operations or in a group cooperation only one component may be allowed to initiate interactions at a time.

```
file_access INTERFACE
  CONSUMER INVOKES {
     open,
     create, ..... }
  SUPPLIER INVOKES    {
     file_server_error }
  ACCESS PROTOCOL     {
     ROLES { Reader, Master }
     open("w", <file>) REQUIRES ROLE Master
                ..... }
```

Figure 2: Interface specification example

In our approach, roles are used to express synchronisation properties. It can be specified, that the initiation of an operation is only allowed, if the invoking component has a specific role (Figure 2). This way, roles can be regarded as attributes of a component, which can be assigned statically or requested dynamically. In order to support the specification of access protocols there are some predefined roles. For example, a predefined master role guarantees the exclusive access to an interface for a component. Additionally, userdefined roles can be introduced to express the required synchronisation properties (e.g. reader role for a client accessing a file server). Moreover, we can define relationships between roles and assign priorities to roles.

The separation of operations and access protocol within an interface specification supports reusability (i.e. the access protocol and the operations could be extended or changed separately). This modularity is very important for the design of an interface ("What is to be done?" is not mixed with "When does it have to be done?"), as well as for readability.

Interfaces can be further classified according to their creation and life time. Normally, each application component provides a set of interfaces available after creation of an application component (static interfaces). Dynamic interfaces are provided during lifetime of an application component. For example, a file server component may provide an initial interface offering an operation for opening files. After performing an open operation the opened file is visible via a dynamically created file interface enabling read and write operations.

Component Specification Language (CSL)

From the application viewpoint each component is specified by the interfaces which are used and provided by it. At component level the behaviour at each interface can be specified in terms of:

- a set of *potential roles:* For a component acting as a consumer we can define some kind of restrictions in respect to its behaviour. For instance, a client component may only be allowed to act as a reader on a file. Moreover, for each role, a policy for requesting it can be defined. We distinguish between implicit, i.e. a role is requested automatically when an operation is invoked, and explicit role requests.
- *binding properties*: They enable the definition of the allowed binding topology and the component responsible for establishing the binding. The binding topology determines whether a single or multiple binding is allowed. Regarding the responsibility, there are several alternatives. Binding can be initiated by the component itself or by another component. In the second case the application component responsible for binding has to be declared explicitly. This type of binding is normally used if we could describe an initial configuration of a distributed application separately by a specification of its application components and their bindings.
- *scheduling policy:* A policy could be declared if a component provides an interface which is decorated with the attribute multiple binding. There are some predefined scheduling policies, such as FCFS. However, a user is able to integrate its own userdefined

policy by providing an appropriate scheduling object (see 3.1).

Our CSL allows the definition of quite different component types. Moreover, *component templates* can be defined to support generic components for the construction of distributed applications. For example, a server template is characterized by a supplier behaviour at a set of application interfaces, multiple binding and a specific scheduling policy (Figure 3).

A component template supporting group work is defined by a symmetric behaviour at each of its interfaces and multiple binding. Additionally, in this case, the coordination between the group members has to be specified in terms of roles.

```
<server> APPLICATION COMPONENT
  APPLICATION PROPERTIES
    SUPPLIER AT <interface>
      MULTIPLE BINDING [RESTRICTED TO <n>]
      SCHEDULING POLICY <policy>
```

Figure 3: Template for a server component

Application Configuration Specification Language (ACSL)

The configuration language for the description of an application configuration follows the concepts introduced in [8]. A complete configuration of a distributed application describes the types of application components from which the distributed application is to be constructed, the instances, how these instances are interconnected and optionally where they are located. A binding between two interfaces of different components is only possible if the interfaces and the related communication contexts can be matched. This means that either two related interfaces must both be symmetric or one of them must act in a supplier role and the other one must act in a consumer role. In addition to specifying initial configurations, the configuration language permits changes to running distributed applications.

However, existing configuration specification languages do not satisfy the requirements of all application domains. Especially, distributed applications in the cooperative work area are configured from interactive components. In this case, a specification technique is needed which allows the description of groups, critical member sets and time restrictions for establishing of distributed applications (if not all the members from the critical member set are available).

2.2 Communication context

A communication context describes the communication requirements in terms of communication properties for the cooperation structure (peer-to-peer or group cooperation), the kind of communication relation (connection-oriented or connectionless), the kind of interaction (message-oriented or operation-oriented), the refinements of cooperation (different role behaviour, need for transaction support etc.) and the properties of a transport service such as time contraints and the desired throughput. A defined communication context can be assigned to each interface of an application component to deal with their varying communication needs. Alternatively, a default communication context can be defined, which enables the specification of a single communication context for an application component as a whole. Furthermore, we are able to assign more than one communication context to an interface to reflect that this interface is accessible via different communication services.

For the formal specification of the communication context [5], we use the macro notation of ASN.1. Based on this macro, an example of a communication context is defined in Figure 4. The example describes a connection-oriented peer-to-peer communication with the invocation of asynchronous remote operations with replies (specified by "OPERATIONS CLASS-2"). The initiation of operations are allowed for both cooperation partners ("SYMMETRIC").

```
comm_context-example1
   COMMUNICATION-CONTEXT
            SINGLE-PARTY
            CONNECTIONORIENTED
            SYMMETRIC
            OPERATIONS CLASS-2
   ::= context01
```

Figure 4: Example for a communication context

A communication context is part of a component specification and can be referenced by its unique name (Figure 5).

```
<server> APPLICATION COMPONENT
  APPLICATION PROPERTIES
            .....
  COMMUNICATION PROPERTIES
            { context01, context02,...}
```

Figure 5: Specification of communication properties

The communication requirements can be partly obtained from the specification of the application behaviour of a component. For example, we can get the information about the cooperation structure from the binding information, the interaction type (message- or operation-oriented) and cooperation behaviour (symmetric or asymmetric distinguishing between client and server behaviour) from the interface specification.

The concept of communication contexts enables the specification of quite different communication requirements. In order to support a flexible communication service, a set of elemental communication building blocks are defined out of which a broad spectrum of specific communication services may be configured.

A communication building block (CBB) comprises a set of related communication functions that share a common purpose and is specified by a service and a protocol definition. We have developed basic and refined CBBs according to several communication properties. The following types of CBBs are defined.

- CBBs for association management which provide services for the establishment, normal and abnormal release of an association.
- CBBs for interaction support distinguishing message-oriented interaction (CBBs providing services for the transfer of messages) and operation-oriented interaction (e.g. CBBs supporting the synchronous and asynchronous invocation of remote operations).
- CBBs supporting transaction-oriented relations for handling distributed transactions.

2.3 Management facilities

Traditionally, application management is performed manually or hardwired into the application code. We seek to avoid the disadvantages of this ad hoc approach by allowing the designer of a distributed application to describe explicitly all the management facilities which have to be provided by a distributed application. We have developed a classification scheme which allows to gain systematically the management facilities for a distributed application. The particular management facilities that are needed depend on the application components to be controlled. Management facilities can be categorized as follows:

Built in management facilities are provided for each application component independent of its functionality. They include monitoring properties like test of liveness of a component, memory and processor usage of a component and its communication behaviour. Additionally, management information of the communication service (e.g. statistics about the amount of invalid PDUs) can be obtained.

Derived management facilities can be directly obtained by analyzing the specification of the application behaviour and communication context of a component. For example, if the application properties of a component are defined by a supplier behaviour at an interface with multiple bindings (i.e. a server behaviour), then monitoring facilities, such as work load monitoring, response time monitoring, throughput monitoring and queue monitoring are supported. Another example is the specification of an application component providing an indefinite amount of interface instances of a specific type. In this case, we need a management facility to create dynamically a new instance of such an interface during runtime. Dependent on the properties of the communication context we could also derive some management facilities. For example, having defined the attribute connection-oriented, management facilities for manipulating the connections (normal and abnormal release) and monitoring the connection characteristics are provided.

Finally, a set of *optional* management facilities can be distinguished. They have to be declared explicitly by the designer of an application component and particularly involve configuration management facilities for evolutionary changes during runtime at component and application level. At component level, the declaration of management facilities such as replacement, replication and migration of components should be supported by an extended CSL (Figure 6). At application level, a rule based management language is suitable allowing the specification of management activities which have to be performed when a specified condition is observed. For example, dependent on the load of a server, a rule could describe that replication has to be initiated; in a group cooperation application the activities to include a member can be specified by this notation.

```
<server> APPLICATION COMPONENT
  APPLICATION PROPERTIES
        .....
  MANAGEMENT PROPERTIES
     REPLICATION,
     MIGRATION,
     REPLACEMENT,
        .....
```

Figure 6: Specification of management facilities

The management architecture is based on a hierarchy of management operations. On the lowest level, a set of elemental management operations are provided. There are operations supporting creation and deletion of components and interfaces, binding and unbinding of interfaces as well as saving and restoring of components. Moreover, management operations for manipulating the administrative state, such as initializing, reinitializing, passivating and resuming of components and interfaces are facilitated.

On top of these management operations, a set of high level management operations are defined. For example, an operation *freeze (ComponentSet)* is defined by a sequence of elemental management operations. In this case, first the components have to be passivated. Then all the bindings have to be terminated and the components must be saved on storage. Finally, the components have to be terminated.

In terms of OSI management, resources are modelled as objects and the management view of a resource is referred to as a *managed object*. In this model, application components can be regarded as managed objects. Moreover, each interface and communication context (which are parts of an application component) can also be modelled as managed objects which are members of managed objects representing application components. From the management viewpoint, each interface is described by an administrative status and a set of bindings (multivalue attribute). The administrative status defines whether an interaction is allowed at a specific time or not.

3. An implementation environment

For the realization of our approach we have developed an object-oriented model where application components, interfaces and communication contexts are represented by a set of interacting objects. For this purpose we use the object-oriented language C++ [12]. Based on the formal specification, the implementation of a distributed application can be supported by a set of tools which facilitates automatic derivation of object-oriented implementations.

3.1 Application components

Application Objects

An interface specification can be naturally mapped onto a set of class definitions. Instances of interfaces are represented as objects. For each application interface we can define a related *application class*, which realizes the interface specification. Additionally, in order to enable a remote access to an object we need a local representative (stub) of a remote object at the consumer and supplier side. A *stub class* has the same public interface as the related application object but in contrast to the remote object a stub object is responsible for the mapping of an operation call onto a set of communication service primitives according to the provided communication service. To support the synchronous and asynchronous invocation of methods a stub object provides for each operation op_i the methods *op_i (Args)* and *op_i _result (Result)*.

Access Protocol Objects

The access protocol feature of an interface is represented by a base class realizing the userdefined specification. In order to implement a declarative access protocol specification, a translation of this specification into a computational representation is needed. In our approach, we model the sequencing rules of an access protocol using petri nets. Operations which are declared to be invoked concurrently are mapped onto threads. For this purpose we use the C++ NIH library [7] which provides threads as special classes. Roles are mapped onto related role objects which provide operations for requesting and releasing roles.

Communication Objects

In our object-oriented model the state oriented specification of the protocol of each CBB is realized by the interaction between a protocol object, a state object and a set of PDU (Protocol Data Unit) objects. The objects rsp. their classes representing the CBBs are stored in a class library. In addition to the class definitions further information is stored to support the retrieval and selection of appropriate classes based on a specific communication context. Such information is the semantical description of a class behaviour in terms of keywords and several kinds of relationships to other classes. The required communication service is built by the combination of suitable protocol objects together with their needed state and PDU objects. This combination can be facilitated by multiple inheritance from the protocol classes.

Management Objects

Management objects are responsible for providing management operations. There are a set of predefined management classes which realize the management interfaces.

Monitoring is realized by a set of metric management objects. Each metric object is characterized by the application or communication object which has to be monitored, the metric procedure and its attributes, e.g. the sampling frequency, notifications to be sent, and some time constrains such as the duration of activity.

Change management is supported by a set of objects realizing the hierarchical management functions mentioned in 2.3. For example, we need a management class which enables the initial configuration of a component and its reconfiguration during runtime. Reconfiguration includes the generation of a new stub object and the creation of a suitable communication service object.

However, the application and communication-oriented objects are not independent from the management properties. Dependent on the management properties of an application component respectively a communication object, each of these objects have to be enriched with some management operations to enable management activities.

In our object-oriented approach this can be achieved by using multiple inheritance. A *managed application object* is obtained through inheritance from an application class and a set of required management classes. For example, a stub object enabling administrative state transitions is obtained by multiple inheritance from an application class (providing operations like open and dir in the file server example) and a state administration class (pro-

viding operations like passivate or resume). As a consequence, different viewpoints can be achieved using polymorphism. This means, that from the application point of view only the application class is visible, whereas a manager sees only the management part.

Object Architecture

The object-oriented structure of an application component consists of a set of interacting basic objects (Figure 7): application objects, management objects, related stub objects and a set of communication service objects. An access protocol object is only configured if the related application component specification involves a specification of this property. The dispatcher object is responsible for invoking the correct operations of an application component (an operation to be performed is indicated by a communication service object). Furthermore, in case of multiple bindings the dispatcher also has to perform the defined scheduling policy. As a result, realization of a component can be regarded as a configuration activity. From a library, the needed application, management, stub, communication objects are selected and configured.

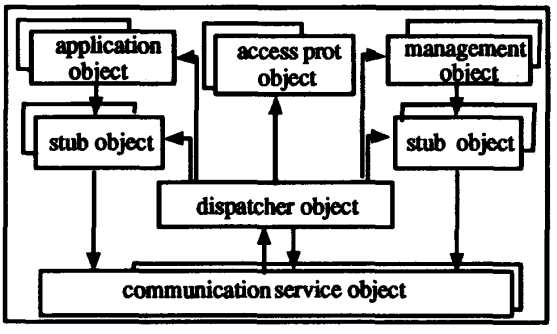

Figure 7: Object architecture of a component

3.2 Application Configuration

A management component is responsible for the *establishment* of a distributed application. Establishment includes the determination of suitable locations for each component (dependent on the component properties and the node facilities), distribution of the related source text, compilation, creation of component instances, their initialization, and setting up the defined bindings. Moreover, a management component enables monitoring and initiating dynamic changes (dependent on the management characteristics of each application component) during runtime.

Within a management component each application configuration specification is represented by a related application configuration object, which can be generated automatically from a formal specification. The public interface supports the definition of components and bindings as well as the establishment and termination of the related distributed application. Moreover, each application component specification is represented as a member object of the application configuration object. It contains all the properties of an application component. From the management point of view, this object architecture allows a fast access to an application configuration and its consistent computational representation. In order to support the creation, removal and retrieval of application configuration objects a dictionary object provides appropriate operations.

This approach has several advantages. First of all, a declarative configuration description can be mapped onto the creation of a computational application configuration object. Furthermore, a 'normal' application component can be responsible for the configuration, which means, to define and initiate a dynamic extension of a distributed application without the need of a separate configuration description. Consequently, providing the concept of configuration objects there is no need to extend a programming language in order to support a dynamic configuration modification during runtime initiated by an application component. This way interactive and programmed configuration and reconfiguration of distributed applications are possible.

3.3 Tool support

We distinguish between tools which create classes and objects, and tools which are responsible for configuration from predefined elements. For example, a *stub generator* produces stub classes from a specification of interfaces. In contrast, an *application component configuration* tool is responsible for the configuration of application components triggered by a component specification. The following tools are needed:

- interface compiler: produces for each interface specification a set of classes (stubs, access protocol, application classes)
- communication context compiler: responsible for the configuration of a communication service from a set of protocol, state and PDU objects representing communication building blocks (triggered by a communication context specification)
- component compiler: produces a dispatcher class and configures a component from a set of application, management, access protocol and communication objects
- application configuration compiler: creates for each application configuration specification a related application configuration object, which is part of a management component.

Regarding the configuration aspect, we can distinguish different configuration steps according to the various levels of abstractions: At component level a set of application-, management-, and communication-oriented objects are configured. However, the objects building an application component themselves have to be constructed by a separate configuration step. For example, a communication object is combined from a set of protocol objects, state objects and PDU objects. At application level, a set of components has to be configured according to the application configuration specification. Finally, the resulting object structure has to be mapped onto basic elements of a concrete operating system. A fundamental structuring element provided by most operating systems is the process. For the transformation of the configured objects into an efficiently running implementation several alternatives have to be considered depending on the facilities of the underlying operating system. E.g., there are different strategies for mapping application components to processes and for handling the information flow between them.

Using the object-oriented paradigm the configuration activity is supported by the concepts of multiple inheritance and the property of building new objects by hierachical combination of basic objects.

4. Conclusion

In this paper we have developed a new methodology for a synthetical construction of distributed applications. Our specification technique supports the specification of quite different aspects of a distributed application, namely application behaviour, management facilities and communication contexts. Moreover, we have analyzed the different relationships between these aspects, especially the derivation of management properties from the application behaviour and the communication requirements.

At implementation level, a tool-based environment is provided which allows the automatic generation of C++ code based on the formal specification of interfaces, communication contexts, application components and application configurations. We have illustrated, that implementation can be regarded as a sequence of configuration steps.

The clear separation of application-, management- and communication-oriented aspects of our concept is a solid base for a synthetical construction of distributed applications. Moreover, the consecutive steps of specification and implementation allow the validation at specification level and the automatic generation of the implementation. At the implementation level, the different aspects could be integrated into a general object-oriented architecture. As a consequence, modularity and reuse of software is improved.

Some basic concepts of our approach have been validated by a prototype implemenation of the *User Agent - Message Store* protocol *(MHS P7)* as part of a joint research project with Digital Equipment Corporation [3].

Future work will focus on the examination of the applicability of our model in respect to several cooperation types. Moreover, a graphical support environment for the development and management of distributed applications will be realized. A prototype of the described tool set will serve as a basis for the development of more general configuration methods and tools.

References

[1] ANSA: ANSA Reference Manual, APM Ltd., 24 Hills Road, Cambridge CB2 IJP, UK, March 1989

[2] CCITT Study Group 7: Support Framework for Distributed Applications (DAF)

[3] Doemel, P. et al.: *Concepts for the Reuse of* Communication Software, Technical Report 5/91, University of Frankfurt/M

[4] ECMA: Support Environment for Open Distributed Processing (SE-ODP), ECMA TR/49 January 1990

[5] Feldhoffer, M.: Communication Support for Distributed Applications, Int. Workshop on ODP, Berlin, Oct. 1991

[6] Geihs, K.: The Road to Open Distributed Processing Technical Report No. 43.9002, IBM European Networking Center (ENC), Heidelberg, 1990

[7] Gorlen, K. E.: An Object-Oriented Class Library for C++ Programs, Software Practice and Experience, Vol 17(12), Dec. 1987

[8] Kramer, J.; Magee, J.; Finkelstein, A.: A Constructive Approach to the Design of Distributed Systems, 10th International Conference on Distributed Computing Systems, Paris, France, May 1990

[9] Magee, J.; Kramer, J.; Sloman, M.; Dulay, N.: An Overview of the REX Software Architecture, Second IEEE Workshop on Future Trends of Distributed Computing Systems in the 1990s, Cairo, 1990

[10] Marzullo, K.; Cooper, R.; Wood, M.; Birman, K.: Tools for Distributed Applications Management, Computer, August 1991

[11] Open Software Foundation: Distributed Computing Environment Request for Technology, DCE Framework - Preliminary Position Paper, Open Software Foundation, Cambridge/USA, January 1990

[12] Stroustrup, B.: The C++ Programming Language, Addison-Wesley 1987