Peter R. Pietzuch

An Event Type Compiler for ODL

Computer Science Tripos, Part II

Girton College

2000

Pro forma

Name	Peter R. Pietzuch
College	Girton College
Project Title	An Event Type Compiler for ODL
Examination	Part II Computer Science Tripos, 2000
Word Count	11,550
Project Originator	Dr. Chaoying Ma
Project Supervisor	Mr. Walt Yao

Original Aims of the Project

The original goal of this project was to extend the existing implementation of the Corba-Based Event Architecture (COBEA) with support for object-oriented event type classes and static type-checking of events. Application programmers should be able to use the Object Definition Language (ODL) to describe event type hierarchies, which are then transformed into C++ stub code by the means of an event type compiler. Moreover, it should be possible to transform these ODL event class declarations into a programming language independent representation by using the eXtensible Markup Language (XML).

Work Completed

The design, implementation and evaluation of an event type compiler was successfully carried out. The compiler transforms ODL event type descriptions into C++ stub code that is usable with the COBEA system. Complex event type hierarchies can be easily expressed in ODL, giving the event application programmer an intuitive programming model to use, combined with the advantages of type-checking. The current implementation of COBEA was altered in order to support the idea of a hierarchical structure between event types. In addition to that, XML output of the event type hierarchy can be generated.

Special Difficulties

None.

Declaration of Originality

I, Peter Pietzuch of Girton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides background information about the project and the motivation behind it. The first section gives an idea about event type hierarchies in event systems. The second section provides a brief introduction to COBEA, which is the event architecture used for this project. The final section focuses on the motivation of the project and gives an overview of its aims.

1.1 Event Type Hierarchies in Distributed Event Architectures

The notion of an *event* can be used in an object-oriented distributed programming environment to build large-scale heterogenous applications [1]. An event is defined to be an *autonomous and asynchronous occurrence*. In a *typed event system*, it usually has a certain type associated with it that describes the data that it is carrying. Whenever such an event occurs at a particular place in a distributed system, it has to be send asynchronously to all interested parties. This can be done transparently to the application programmer by an event architecture that provides an event passing service. It is an important requirement of such a service to be easy and efficient to use, but also to be part of a scalable framework in order to support large distributed applications.

Application areas like multimedia systems, mobile systems and telecommunication systems especially benefit from event communication because they inherently rely on the notion of autonomous asynchronous occurrences. For instance, in a mobile system, intelligent agents can communicate by passing events between each other. Events are a good paradigm in this case because agents are required to interact with the world in real-time, and thus generate events in response.

The object-oriented approach to programming [8] has many advantages and has therefore been adopted for distributed systems wherever possible. Such an approach is also favourable for expressing the types of events. A collection of event types can often be structured into an *inheritance type hierarchy* so that base classes capture the general properties of events and derived classes allow specialisation into particular types of events. Figure 1.1 shows an example of an event type hierarchy that could be used for a surveillance system that involves different kinds of sensors. All event types are derived from a general class BaseEvent and inherit attributes from all their ancestor classes. For

instance, the class EntranceEvent is a specialisation of a RoomEvent since it is signalled when a person enters a room. Such an event type hierarchy is easy to use by the application programmer because it directly tries to model the real world system.

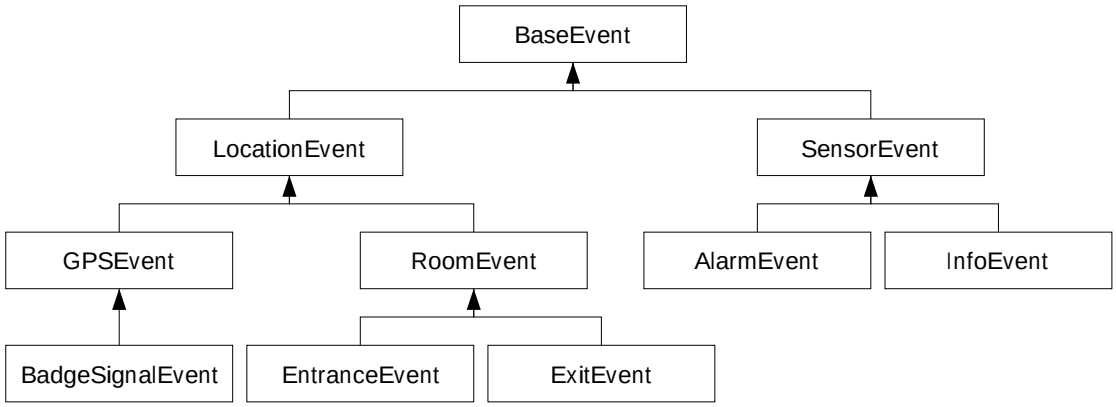

Figure 1.1: An Event Type Hierarchy for a Surveillance System

The main part of this project is about providing means of specifying and using object-oriented event type hierarchies with the event architecture COBEA[1]. This includes the modification of the existing COBEA system and the design and coding of an *event type compiler* that compiles an event hierarchy description into a representation understandable by COBEA and usable by the event application programmer.

1.2 COBEA — A Distributed Event Architecture

The Opera Systems Group in the Computer Laboratory has developed a distributed event architecture called COBEA which is based on CORBA[2] as an underlying middleware. CORBA is an industry standard for object distribution in networks. Although CORBA provides its own event service, it lacks several important features like parameterised filtering, fault tolerance, access control and composite events. COBEA has been built to address these problems.

The COBEA system is an implementation of the Cambridge *publish-register-notify* paradigm for event systems [6]. In this model, the entities in an event system are classified into *event sources* that supply events and *event sinks* that consume them. Event sources publish the event types that they are willing to signal. Then, event sinks can register their interests in particular event types with an event source. Whenever an event occurs at an event source, all registered event sinks are notified by the COBEA system (Figure 1.2). In a client/server model, event sources can be viewed as servers whereas event sinks behave like clients.

[1] COrba-Based Event Architecture
[2] Common Object Request Broker Architecture

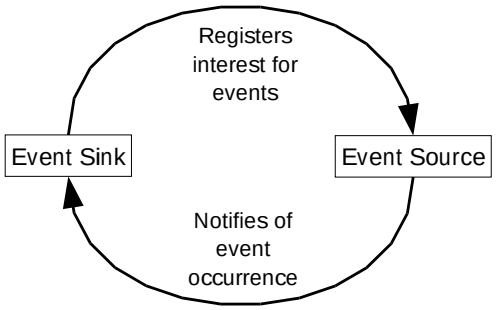

Figure 1.2: The COrba-Based Event Architecture

The registration process of COBEA supports *parameterised filtering* so that event sinks can register for events with particular data values. For example, Figure 1.3 shows the event type GPSEvent that could be generated by a GPS[3] receiver. It has three attributes which define the location of the receiver while the event was signalled. Parameterised filtering allows a client to be notified only when particular values of the event are set so that, in this example, it will only receive events for which x=100. Asterisks denote a wildcard value. Such a scheme has the advantage that it is scalable because network traffic for event notification is only generated if an event sink has registered for a particular instance of an event type that has occurred. In other words, filtering of events happens at the source-side so that superfluous notifications are avoided.

GPSEvent	
double x	100
double y	*
double z	*

Figure 1.3: Parameterised Event Registration

Event sinks and event sources are decoupled from each other and communicate via specified interfaces that are provided by an underlying *event engine*. However, in a system where events have a type and are type-checked, there needs to be common knowledge about event types between event sources and event sinks. Consequently, event sources have to provide different methods for interest registration depending on the event type.

[3]Global Positioning System

1.3 Motivation for the Project

1.3.1 Static Type-Checking for Events

In the current implementation of COBEA every event type is identified by a string carrying a unique *event type name*. The additional data associated with the event is stored in form of attributes in a data structure, as shown in Figure 1.4. Attribute values are represented by a name of type string and a value of the CORBA data type *any* that is capable of holding any possible IDL[4] data type. Several attributes are combined into an IDL *sequence* structure.

```
struct Property {
  short len;
  string name;
  any value;
};
typedef sequence<Property> PropertySeq;
```

Figure 1.4: Event Attribute Representation in the COBEA System using the Type *any*

A disadvantage of such a scheme for event type representation is that it does not allow *static type-checking* of event types. This is the case because, at compile-time, all event types appear to have the same type to the compiler, namely a sequence of type any. However, when event sources and event sinks communicate with each other at run-time, they may use different declarations of an event type but use the same event type name. Consequently, an event sink will fail when extracting the attribute values from a sequence of anys since it expects different value types. The CORBA system will raise an exception if, for example, the application attempts to extract an integer value from an any variable in which a string is actually stored.

A possible solution to this problem is to enforce the usage of certain event types for both event sinks and event sources. In most distributed applications, the events used by the entire system will be specified during the design phase and become part of the specification document. Therefore, it is desirable to take these *event type declarations* and transform them into programming code which event applications can be linked against. As a result, all event entities in the application have a common knowledge about the defined event types and static type-checking can be carried out by the compiler. This project has resulted in the development of an automated way of transforming event type declarations into C++ stub code using an *event type compiler*. It is felt that this greatly facilitates the development of distributed event applications using the COBEA system.

[4]Interface Definition Language used by CORBA

1.3.2 Support for Event-Type Hierarchies

A second criticism of the current COBEA implementation is that it does not support event type hierarchies like the one mentioned previously. All event types in COBEA represent a flat collection of data types which makes it difficult to express relations between them. As C++ is inherently an object-oriented programming language, it is desirable to retrofit the support for inheritance type hierarchies on top of COBEA.

This has been achieved by making the event type compiler generate C++ classes which represent event types. These classes provide methods for interfacing the underlying COBEA system in a way that the C++ event type declarations are translated back into the simple COBEA event type representation. In addition to that, modifications of the original COBEA system were necessary to incorporate the notion of event super-class notification.

1.3.3 XML Output of Event Type Declarations

Finally, it is desirable to have a representation of event type declarations that is not bound to a single platform or programming language like C++. For example, this would allow the storage of event types in an *event type repository* at a central location. Clients and servers in a distributed heterogenous environment can then query the repository dynamically at run-time, in order to find out about event types and use these consistent type declarations for event communication. This is particularly important in inter-domain applications so that clients and servers can cross domain-boundaries. For this purpose, the XML[5] Schema Language, being a new emerging industry standard, was chosen for event type representation.

[5] eXtensible Markup Language

Chapter 2

Preparation

The preparation chapter concentrates on the work done before the actual implementation phase. Good software engineering practice was one of the overall aims so that a large amount of preparatory work was carried out. The first section describes all the standards, software systems and languages that were learned and used for this project. Care has been taken to explain the design decisions behind the choice of particular languages or systems and to show the amount of knowledge acquired for this project. After giving this background knowledge, the next section establishes the requirements of the project by describing its subcomponents. Following that, a list of acceptance criteria is established for the evaluation process. The final section details the project milestones used to assess the progress of the project. All this ensured that the implementation stage would go smoothly and in a coherent, well-organised manner.

2.1 Implementation Prerequisites

2.1.1 C++ and the Development Environment

C++ was chosen as the main implementation language for the entire project. This was motivated by the fact that the present implementation of the COBEA system is written in C++ and having to deal with a single language for the whole body of code was felt to make development easier. Another reason was that C++ allowed a professional object-oriented approach to the whole project. Prior to this project, I had no practical knowledge of C++ so that a certain amount of time during the preparation phase was spent familiarising myself with the language and writing small sample problems to understand the usage of template classes, template friend functions and pointers to member functions. The C++ book [9] by B. Stroustrup was a helpful guide, especially when dealing with more advanced features of the C++ language.

The C++ compiler used was GCC 2.95.2, running under the operating system Linux 2.2.12 on my private machine. The source tree and all other associated project files were managed by CVS[1]. Daily

[1] Concurrent Versions System

backups of the code and documentation repository were automatically made to the Pelican Archive Service by a Unix shell script. Further backups were made with a tape streamer.

2.1.2 CORBA — Common Object Request Broker Architecture

The CORBA specification [7] was produced by the OMG[2] in order to standardise object-oriented distributed computing in heterogeneous networks. It allows processes to use remote objects over a network in a transparent way. A variety of programming languages are supported. The implementation of CORBA 2.0 used by COBEA is omniORB2 [5] developed at the AT&T Laboratories Cambridge.

CORBA relies on IDL[3] which is a way of defining the interfaces between objects that are called remotely. IDL provides a certain set of data types and a syntax for defining interfaces consisting of method signatures. The CORBA system knows how to marshal and unmarshal the IDL data types and therefore can provide a platform-independent transport service for these. Since CORBA is language-independent as well, mappings from IDL to a variety of programming languages are defined in the CORBA specification. By means of an IDL compiler, an IDL interface specification can be transformed, for example, into C++ header classes for clients and servers that use this interface.

It was an important task during the preparation phase to familiarise myself with the basic mechanisms of CORBA [10] and the particular implementation omniORB2 [5]. In order to efficiently do this, a couple of example client/server applications were designed and tested. I had to learn the syntax and semantics of IDL and the notion of an ORB[4]. In particular, I would like to mention the effort to understand the subtleties involved in dealing with the general *any* data type when storing arrays and sequences. The omniORB2 mailing list was a valuable help for this.

2.1.3 COBEA — CORBA-Based Event Architecture

A deep and thorough understanding of the current version of the COBEA implementation was essential for the success of the project. Unfortunately, there does not exist a detailed documentation of the source code so that it was necessary to infer most of its functionality by direct analysis[5] and from past student projects [3, 4]. However, the code has changed considerably over the time so that past projects were only of limited help. Parts dealing with parameterised event filtering especially required a detailed investigation. My analysis revealed a number of minor bugs and forced me to re-implement a part of COBEA concerned with the de-registration of event types with event sources. In the following, I would like to give a short overview of the main features of the COBEA source code which are necessary for the understanding of this project.

The COBEA system can be divided into classes concerned with event sinks and with event sources (Table 2.1). The two classes FSrc_i and Snk_I implement the IDL interface definitions FSrc and

[2] Object Management Group
[3] Interface Definition Language
[4] Object Request Broker
[5] Thanks goes to W. Yao and Dr. Ma for helping me every time that I seemed to be completely lost in the code.

Snk. The source interface, FSrc, has the operations reg_event and dereg_event for event registration and de-registration. The sink interface, Snk, supports the operations notify, which is called for notification of an event occurrence it has previously registered for, and disconnect_snk for disconnecting a sink from a source when the event source stops generating events. The actual implementation of the event source, FSrc_i supports two further methods, namely signal for signalling an event and new_event which registers a new event type with an event source, subsequently allowing event sinks to declare their interest in this event. A more complete listing of the COBEA methods can be found in Appendix A.

Class Name	Description
IDL Definitions	
FSrc	Interface for an event source
Snk	Interface for an event sink
Event Source	
FSrc_i	Implementation of event source interface
Template_Table	Stores registration information of event sinks
Priority_q	Priority queue used for event notification delivery
Event Sink	
Snk_i	Implementation of event sink interface
Call_backs	Stores call back functions for event notification

Table 2.1: Main Source and Sink Classes of COBEA

A problem that has to be addressed by a distributed event architecture like COBEA is network unreliability. An event sink might not receive event notifications because of a network communication failure without noticing it. For this reason, event sources in COBEA can periodically signal special *heartbeat* events which event sinks can register for. In case of a network problem, an event sink will stop receiving these heartbeat events and consequently knows that the connection to the event source has been lost.

2.1.4 ODL — Object Definition Language

ODL is a standardised way of describing objects in an object-oriented database environment which was defined by the ODMG[6] in [2]. The specification includes the definition of an object model that is adopted and a binding from ODL to C++ that allows the translation of ODL object declarations into C++ classes. It has its own set of ODL data types for type declarations that encompasses both simple and complex data types.

In this project, events type declarations as processed by the event type compiler are expressed in ODL. This has several advantages: Firstly, ODL can be considered as being a superset of IDL because, in addition to facilities for specifying method signatures of IDL (which are not needed in

[6] Object Database Management Group

event type declarations), it allows the declarations of data members in classes that can be ODL data types. Secondly, it is possible that the event type declarations in ODL can be directly stored in an ODMG-compliant database without further changes or additions. Finally, ODL proves to be a relatively mature standard with a flexible collection of data types. During the preparation phase, the idea behind ODL and the syntax was learned from the specification documentation.

2.1.5 XML Schema — eXtensible Markup Language

As previously mentioned, the XML Schema language is used in this project to represent event type declarations in a different way from ODL. XML Schemas are an extension to the original XML Specification [11] by the W3C[7] and allow the expression of arbitrarily structured data under the constraint of data types. They are defined in two specifications, one focusing on the structure of schemas [12] and another on the available simple and user-defined data types [13].

The main reason for adopting XML is that it is becoming an industry standard for expressing structured data. However, it is important to stress that by the time of this project, the XML Schema Specification is still a working draft which means that certain aspects of it are either contradictory or entirely unspecified. Nevertheless, it was successfully used to express at least a subset of all possible event types in XML. The XML output is created by the event type compiler in addition to the C++ stub code.

Similarly to ODL, XML Schemas were completely unfamiliar to me before this project. I had to study the two XML Schema specifications understanding of which was dependent on the general XML specification in [11]. In order to practice the creation of XML files, I used the Apache Xerces-J Parser[8] under Java to check my sample XML files for well-formedness and validity, since this parser seemed to be quite mature in its development stage.

2.1.6 libxml — An XML library

The C library libxml[9] provides functions for parsing and generating XML data. It tries to be compliant with the DOM[10] by W3C for representing the XML parse tree. Calls to this library are made for building an XML parse tree from event type declarations in memory and writing the final XML Schema to a file. Although libxml is not XML schema-aware, it is possible to generate XML Schemas with this library because the syntax used by schemas is a subset of the general XML syntax. Being still under development, the library restricts the generation of an XML parse tree to certain XML constructs only, which is a problem if for example well-formed XML Schema output is necessary. The library is accompanied by short documentation that helped me to learn its usage by studying the example code provided.

[7] World Wide Web Consortium
[8] available from *xml.apache.org* under the Apache Software License
[9] available from *www.xmlsoft.org* under the GNU Lesser General Public License
[10] Document Object Model

2.2 Requirements Analysis

In this section, I would like to state the requirements made by the individual parts of the project. It contains a functional description of the event type compiler that was created before the implementation phase. Moreover, the requirements of the mappings defined in the implementation phase are described.

2.2.1 The Event Type Compiler

The event type compiler should take an ODL input file that contains event class declarations and output C++ stub code which can be used with the COBEA system. Additionally, it should generate XML Schema declarations of the event types. The compiler follows the standard multi-stage structure for compilers (Figure 2.1). The lexer and parser for ODL are developed with the help of the Unix utilities *Flex* and *Bison*. Because the compiler is supposed to output both C++ code and XML code, an important requirement was to make the abstract parse tree language independent and allow the addition of several code generators to the compiler. In other words, the compiler back-end should be independent from the code generation front-end in order to increase code re-usability. The output files produced for an ODL input file named Events.odl should be as follows:

Events.h contains the C++ event class declarations generated from the ODL declarations. All generated C++ classes are directly or indirectly derived from a base class called BaseEvent declared in BaseEvent.h and implemented in BaseEvent.cc. Common functionality of all event classes is provided by this base class.

Events.cc contains implementation code that deals with marshalling and unmarshalling of event data so that it can be passed on to the underlying COBEA system.

Events_Src.h contains the C++ event classes used by event sources. These classes are derived from the original classes in Events.h and are extended with methods for signalling events, etc. The naming convention is the original class name plus the extension _Src. Applications which are event sources are supposed to include this header file.

Events_Src.cc contains the implementation of the methods used by event sources.

Events_Snk.h contains the C++ event classes used by event sinks. These classes (original class name plus _Snk) are derived from the original classes in Events.h and are extended with methods for parameterised registration etc. Applications which are event sinks are supposed to include this header file.

Events_Snk.cc contains the implementation of the methods used by event sinks.

Events.xsd contains the XML Schema declarations of the ODL event types.

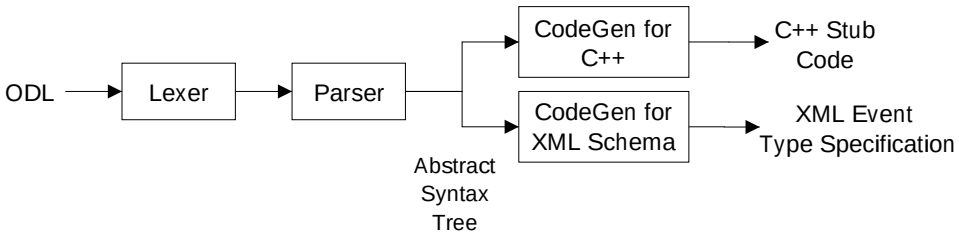

Figure 2.1: The Event Type Compiler

Mappings for the C++ Stub Code

In order to achieve the main objective of building the event type compiler, the work was subdivided into the design of several *mappings*. A mapping was defined to be a translation from one form of event description into a different, equivalent one. Figure 2.2 shows the three mappings that are required for the generation of C++ stub code for COBEA applications by the event type compiler and the following list states their functional requirements. An iterative design paradigm was intended for the mappings which meant that hand-coded versions of all the files to be automatically generated by the event compiler were created, evaluated against these requirements and the application scenarios described in Section 4.2 and refined if necessary. The description of the mappings was kept in documents in the repository in order to assist the development task.

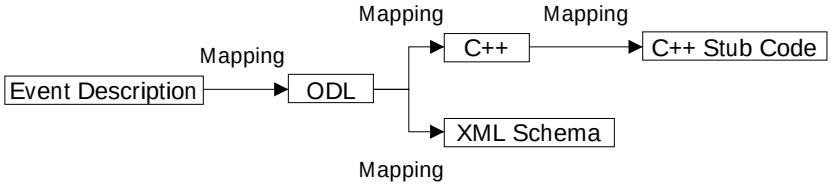

Figure 2.2: Mappings from Event Descriptions to final C++ Stub Code for COBEA and XML Output

Event Description $\rightarrow$ ODL An event description is an informal description of the event type in natural language. It contains the main features of the event type and is translated into ODL syntax that precisely defines the event type including all the data types. This mapping is usually performed by the event application programmer.

It is the goal of a good object-oriented design to structure the event types into an inheritance type hierarchy. Base types should capture the general behaviour of events and should be specialised into particular event types. All event types should share a common ancestor named BaseEvent that carries properties needed by all events.

ODL $\rightarrow$ C++ After that, the ODL specification of the event types classes requires translation into C++ classes since this is the language used for programming the application. The mapping ap-

plied here should be ODMG-compliant defined by the C++ Binding Chapter in [2]. Therefore, an implementation of all the ODL data types in C++ preserving the semantics is needed. This has to be verified by dummy applications that use those C++ ODL data types. The translation described by this mapping is performed automatically by the event type compiler.

C++ $\rightarrow$ C++ Stub Code for COBEA Finally, the C++ class definitions obtained from the ODL event types must be extended with methods for event registration, signalling, marshaling of event data, etc. These are the actual interface methods to COBEA that will be used by the event application programmer. A suitable set of interface methods was designed and implemented as part of this project and the event type compiler automatically includes them when outputting the final C++ stub code. Moreover, it is desirable that the compiler outputs separate stub code files for event sources and event sinks because of the differences in interface and implementation methods. The main consideration when defining the interfaces is that they should easily support the development of standard event applications.

Mappings for the XML Schema Output

The design of a further mapping is necessary for the XML Schema output of the event compiler as denoted in Figure 2.2. By applying this mapping, the event compiler translates the ODL event class declarations into XML Schemas. Similarly to the ODL to C++ mapping, this requires the translation of all ODL data types into XML Schema data types. However, with the most recent version of the XML Schema Specification [12, 13], it is not possible to entirely preserve the meaning of all ODL event class declarations. The main problem is that XML Schemas do not encompass a method for expressing collection data types like for example sets and lists. Therefore, XML Schema declarations currently allow only a subset of all possible ODL type declarations.

2.2.2 Wrapper Classes for COBEA

The COBEA library leaves the application programmer with the task of properly initialising the CORBA system and parts of COBEA before using events. Since the C++ stub code methods realise a certain kind of abstraction which allows faster application development as described in Section 4.3, it would be desirable to provide the same kind of abstraction when dealing with event sources and event sinks during the set-up phase. In order to solve this problem, two C++ wrapper classes for COBEA shall be provided called EventSource and EventSink. Instances of these two classes represent event sources and event sinks, respectively, and shall be compatible with the C++ event classes and event type hierarchies produced by the event type compiler.

2.3 Acceptance Criteria

During the progress of the implementation stage, it is important to have a method of ensuring that the final system is in accordance with the initial design ideas to guarantee a professional approach.

Although the requirements given in the previous sections are informal, it was believed that they would prove adequate for evaluating single system components. In addition to that, the overall behaviour of the event type compiler was measured against two *Application Scenarios* (Section 4.2) in the form of descriptions and ODL event type declarations that should be correctly transformed by the compiler. An application programmer should be able to create an event application for these two scenarios.

2.4 Project Milestones

Since a substantial amount of work was necessary to carry out this project according to the requirements, the project was split up into milestones. After each milestone, the result of the work was evaluated. The following list gives an idea of the major milestones for the project:

- Literature study and initial preparation of work plan.
- Design of a suitable mapping from event descriptions to ODL.
- Implementation of the ODL to C++ Mapping.
- Development of the C++ stub code including interface methods and their implementations.
- Creation of a Lexer and Parser for the generation of an abstract syntax tree for an ODL input file.
- Development of the code generator for the C++ stub code.
- Design of a suitable mapping from ODL to XML Schemas.
- Development of the code generator for XML Schemas using the library libxml.

Comparing this to the original project proposal, only minor changes had to be included. First, the implementation of the ODL data types in C++ required more time than predicted and therefore became a milestone of its own. Second, the Java stub code generation was excluded from the project because it was realised that it would be analogue to the C++ code generation, but nevertheless require a significant amount of repetitive work. Third, the possible project extension concerning a parser for event filtering expression was not attempted due to time constraints. This was mainly the case because the work with XML Schemas turned out to be more challenging than expected.

Chapter 3

Implementation

This chapter tries to give a detailed description of the implementation work of the project. It first describes the actual representation of event data types in ODL. After that, it shows the implementation of the C++ binding of ODL, and gives a brief overview of the event source and sink wrapper classes for COBEA. The next two sections focus on the features of the generated C++ stub code and the XML Schemas, particularly discussing any design decisions made during implementation. The chapter concludes with a presentation of the event type compiler that implements all the described mappings.

3.1 Design of the Event Type→ODL Mapping

3.1.1 BaseEvent Class

All event types that are used by a distributed event application must be derived from the ODL ancestor class BaseEvent. This has two advantages, namely, that all event types can be treated as being of the general type BaseEvent using polymorphism and that the common functionality of all event types can be passed on to user-defined event types by inheritance resulting in a cleaner design. The class BaseEvent is defined as shown in Figure 3.1.

```
class BaseEvent {
  attribute long id;
  attribute short priority;
  attribute string source;
  attribute timestamp signal_time;
};
```

Figure 3.1: The definition of BaseEvent in ODL

The attribute id is a unique identification number that is assigned to every event instance by

COBEA. The field priority gives an indication of the priority of the event instance. This allows COBEA to prefer the handling of events with higher importance. The string source contains the name and network location of the event source that has signalled this event instance. Finally, the attribute signal_time holds a time stamp of the signalling time.

3.1.2 Heartbeat Class

In order to recognise network failures between event sources and sinks, Heartbeat events are exchanged periodically. This event type is the only type not derived from BaseEvent because its interface does not require the overhead of a general event type due to its restricted functionality[1]. Figure 3.2 shows the definition of the Heartbeat event in ODL.

```
class Heartbeat {
  attribute long id;
  attribute short priority;
  attribute string source;
  attribute timestamp signal_time;
  attribute long interval;
};
```

Figure 3.2: The Definition of Heartbeat in ODL

The attributes id, priority, source and signal_time have the same meaning as for BaseEvent. The attribute interval contains the interval in milliseconds between two heartbeat events. If an event sink does not receive the next heartbeat event after this amount of time[2], it is likely that a network problem has occurred or the event source has died.

3.1.3 User-defined Classes

The main part of the mapping is concerned with a subset of ODL that can be used by the application programmer to declare event type classes in ODL. The Table 3.1 shows all the non-datatype keywords from the ODL grammar that are supported by the event type compiler. All keywords that are database-related or concerned with the declaration of interfaces in the form of method signatures are not supported because they are not directly relevant to the task of event type declaration. A list of all unsupported ODL constructs can be found in Appendix B.

All the ODL data types the are supported by the event type compiler are shown in Table 3.2 with their corresponding keywords. The actual implementation of the d_-types in C++ is described in the next section.

[1] For instance, it is not possible to explicitly signal this event.
[2] assuming the network latency is negligible

ODL Keyword	Description
class	Class definition
extends	Inheritance specification
attribute	Attribute definition
const	Constant attribute definition
typedef	Type definition

Table 3.1: ODL Keywords supported by the Event Type Compiler

The class MyVeryImportantEvent in Figure 3.3 is an example of a user-defined ODL event type class. It is directly derived from BaseEvent and combines most of the features of ODL event classes as supported by the event type compiler: The data fields no and id are integer constants that contain expressions to be evaluated by the C++ compiler. LongSet is a type definition for a set of long integers. The structure of type EventDescr, which is instantiated as eventDescr, contains the enumeration eventKind and another inner structure named details.

Limitations

As mentioned previously, the event type compiler only supports a subset of the entire Object Definition Language. The design decision to restrict the ODL subset has been made because ODL contains means to interface with an underlying object-oriented database system which was not necessary for this project.

A second design constraint is that interface and module specifications are not allowed for event types. The idea behind ODL is to describe, in addition to the attributes that store the data, the methods for manipulating the attributes. However, events are by definition entities that carry only data and therefore do not require their own methods for manipulating this data.

3.2 Implementation of the ODL→C++ Mapping

The ODL specification in [2] defines a binding from ODL to C++. The general paradigm behind this mapping is that C++ classes stand for ODL classes and ODL attributes are represented by C++ data members. In particular, the mapping requires that most of the ODL data types as mentioned in Table 3.2 are represented by corresponding d_-types in C++. These d_-types are clearly specified, and it was part of the project to produce an ODMG-compliant implementation.

Since the majority of the required ODL types like, for example, date, time, and string are frequently used general types, it has been decided not to re-implement them in C++ from scratch but to use the correct and efficient implementations in the C++ STL[3]. The fact that the STL interfaces to the data types are different from the ODL specification was compensated by the provision of *wrapper*

[3] Standard Template Library

ODL Type	C++ Binding Type	Description
	d_Object	Ancestor class for all ODMG-compliant objects
long	d_Long	32 bit signed integer
short	d_Short	16 bit signed integer
unsigned long	d_ULong	32 bit unsigned integer
unsigned short	d_UShort	16 bit unsigned integer
float	d_Float	32 bit IEEE 754 floating-point number
double	d_Double	64 bit IEEE 754 floating-point number
boolean	d_Boolean	Boolean value
octet	d_Octet	8 bit value
char	d_Char	8 bit ASCII character
string	d_String	String
enum<>	enum	Enumeration
any	d_any	Any value
	d_Collection	Abstract base class for collection types
	d_Iterator	Iterator class for collection types
set<>	d_Set	Set collection
list<>	d_List	List collection
array<>	d_Varray	Variable size array
date	d_Date	Date
time	d_Time	Time
timestamp	d_Timestamp	Time stamp
interval	d_Interval	Time interval
struct<>	struct	Structure

Table 3.2: ODL Data Types supported by the Event Type Compiler

classes that have ODMG-compliant methods to the outside world and, internally, represent the data as STL data types. For example, the ODL data type d_Set is implemented by having a private data field which is an STL set. All access to this internal set is done by ODL-compliant methods provided by the wrapper class. ODL data types like d_Date and d_Interval that do not have analogue STL types were implemented directly using basic types in a straight-forward manner.

Although the base class d_Object has been supplied, its implementation is mainly empty because this class is supposed to provide persistency support for all objects that need to be stored in a database. The ODL-enhanced COBEA system does not provide such a facility as it is not associated with a database.

A problem that has occurred during the implementation phase of the ODL data types was an intrinsic incompatibility of the d_Iterator class with the STL iterator class: The STL iterator is meant to be associated with a particular type, so that every collection type provides its own implementation of an iterator. In contrast to that, the ODL specification sees the d_Iterator class as a general purpose object that is compatible with all possible collection types. This problem was solved by providing an inheritance hierarchy of d_Iterator classes with derived classes for each particular collection type.

```
class MyVeryImportantEvent extends BaseEvent {
  const short no = 1
  const short id = (no * 2) + 1;

  attribute string descr;
  attribute any additonalData;
  typedef set<Long> LongSet;

  attribute struct EventDescr {
    enum Kind {
      alarm,
      sensor,
      data
    } eventKind;

    struct Details {
      attribute LongSet dataValues;
      attribute unsigned long count;
    } details;

  } eventDescr;
};
```

Figure 3.3: Example of a user-defined ODL event type class

These specialised d_Iterator classes use different STL iterators as their internal representation. For instance, if a d_Iterator for the type d_Set was required, the derived class d_Iterator_Set is used which contains an STL set iterator as a private data field.

3.3 C++ Wrapper Classes for COBEA

The current implementation of COBEA exposes a significant amount of implementation detail of the event system to the application programmer. For instance, it is the responsibility of the programmer to properly initialise the CORBA system before using COBEA. Moreover, event sources have to instantiate the COBEA event source implementation in the class FSrc_i and event sinks have to instantiate the classes Snk_i and Call_backs passing a number of correct initialisation parameters. This is not very transparent to the programmer so that it was decided to provide an abstraction to event sources and sinks in the form of two C++ wrapper classes called EventSource and EventSink. The first instance of these classes performs the required CORBA set-up and after that, its methods can be used to interface the underlying COBEA system and to access the functionality of event entities. Figure 3.4 shows the architecture obtained by using these two classes. A further comparison of these

classes with old COBEA mechanisms can be found in Section 4.3.

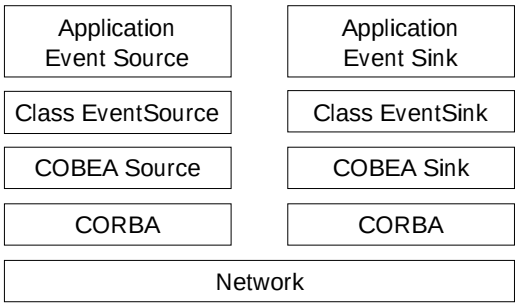

Figure 3.4: Architecture with Event Wrapper Classes

3.3.1 The Class EventSource

EventSource is a C++ class that represents a COBEA event source that generates events. It only contains a single public method called `register_new_event(...)` that is used to inform the event source that it is capable of signalling a particular new event type so that clients are allowed to register their interest in this type. The constructor is passed the string name of the new event source to be created and a host name for identification in a network. It performs all the necessary COBEA initialisation and creates a heartbeat event that is used for the heartbeat protocol described in Section 3.1.2.

3.3.2 The Class EventSink

The class EventSink is the wrapper class for a COBEA event sink that does any initialisation tasks. The class does not contain any public methods to be used by the application programmer. Every event sink must be directly connected to a single event source from which it will receive all its events. This event source is specified in the constructor of EventSink. It is important to note that the class EventSink does not have methods for event interest registration because this is done by calling methods from the stub code interface described in Section 3.4.3.

3.4 Design of the generated C++ Stub Code

After the event type compiler has transformed the ODL event type definitions into C++ classes by applying the ODL→C++ mapping described in Section 3.2, it adds certain methods to the event classes whose purpose is to act as an interface to the functionality of COBEA. This adds another level of abstraction to the event system because it allows the application programmer to treat events as objects with methods for manipulating them. There exists common functionality that is required by both event sources and event sinks. However, certain methods should only be available to event sources or event sinks. Therefore, it was decided to derive two different classes from the general event

type class, namely one for the event source and another for the event sink. In order to illustrate this division, Figure 3.5 depicts the C++ class names and inheritance relations that are generated for an example ODL input file named Events.odl that contains only a single event class, MyEvent, that is directly derived from BaseEvent.

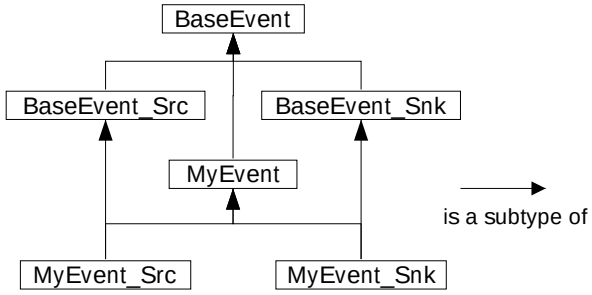

Figure 3.5: Class Names and Inheritance Relations of the C++ Stub Code

This hierarchy largely relies on the feature of multiple inheritance of C++. The header file BaseEvent.h contains the C++ class BaseEvent that is the transformation of the ODL event type BaseEvent, as described in Section 3.1.1. The classes BaseEvent_Src and BaseEvent_Snk, which are derived from the general BaseEvent class contain the specialisations required for event sources and sinks. The header file Events.h describes the class MyEvent that states the event type as defined in ODL and generated by the event type compiler. It is derived from the general BaseEvent class and further specialised into two classes, MyEvent_Src and MyEvent_Snk, which are the actual classes used by event sources and sinks for accessing the event type MyEvent.

3.4.1 Common Event Class Declaration

In the above example, the methods common for event sources and sinks are stored in a header file named Events.h. This header file mainly augments the event class declarations by methods for *marshalling* and *unmarshalling* attributes. These methods are not called directly by the application programmer and are shown in Table 3.3.

Method Name	Description
public:	
`getPropertyCount()`	Returns the number of attributes
protected:	
`getProperties(...)`	Marshals attributes
`setProperties(...)`	Unmarshals attributes

Table 3.3: Methods for Marshalling and Unmarshalling Event Class Attributes

The method `getPropertyCount()` returns the number of attributes that the event class possesses

so that the correct amount of memory can be allocated for the marshaled data. The protected method `getProperties(...)` reads the values from all the attributes of the event instance and inserts them into a sequence of type any. This sequence data structure is then passed on to COBEA. The CORBA system is capable of transporting data of type any over the network in platform independent way. The method `setProperties(...)` performs the reverse process by setting the data members of the event class from a sequence of type any. The source code for inserting and extracting the attributes from the sequence is statically created by the event type compiler and can be found in the C++ file Events.cc.

3.4.2 Event Source

The C++ header file Events_Src.h contains the declaration of the class MyEvent_Src that is the actual event class used by an event source for signalling the event type MyEvent. The interface methods available to the application programmer are listed in Table 3.4.

Method Name	Description
public:	
`signal_event(...)`	Signals event occurrence
`getInhertianceSpec()`	Returns inheritance hierarchy

Table 3.4: Public Methods supported by an Event Class Declaration for Event Sources

To be more precise, these methods are defined in the class BaseEvent_Src and inherited by MyEvent_Src because they are the same for all possible ODL event classes. The method `signal_event(...)` is called in order to signal the occurrence of the event MyEvent. A method parameter holds a reference to the event source that is to be used as an origin for this event signal. For this reason, it is possible to register the same event instance with several event sources and select the desired event source by its name at signalling time.

The second method, `getInheritanceSpec()` is not used directly by the application programmer, but instead it allows COBEA to find out about the structure of the event class inheritance hierarchy. Section 3.4.4 focuses on this mechanism used for superclass registration.

3.4.3 Event Sink

The class MyEvent_Snk in the header file MyEvent_Snk.h carries the event type and method declarations used by event sinks. Table 3.5 gives an overview about the methods of its interface.

The method `register_event_interest(...)` is called to register the interest in this event type with a particular event sink specified as a reference parameter. A second parameter holds a pointer to the call-back function that is executed whenever this event occurs. When this function is called, it is passed a pointer to an instance of the occurred event so that the event sink has access to all the attributes associated with this event. The method `deregister_event_interest(...)` undoes a previous registration.

Method Name	Description
public:	
`register_event_interest(...)`	Registers event interest
`deregister_event_interest(...)`	Deregisters event interest
`get_`*{attr_name}*`_wildcard(...)`	Gets wildcard state for *{attr_name}*
`set_`*{attr_name}*`_wildcard(...)`	Sets wildcard state for *{attr_name}*
protected:	
`getFilterExpression()`	Gets COBEA filter expression

Table 3.5: Public Methods supported by an Event Class Declaration for Event Sinks

Another important feature of the event sink interface is the support for parameterised event registration: Prior to the call to `register_event_interest(...)`, the attributes of the event instance can be set to particular values so that only instances of the event with those values set will be notified. The standard setting for all attributes is a *wildcard value* so that the current value of an attribute is ignored and no parametrised filtering is used. By making the call `set_`*{attr_name}*`_wildcard(false)` the wildcard for the attribute called *{attr_name}* is deactivated, and the current attribute value is passed on to COBEA for parameterised filtering. The method `get_`*{attr_name}*`_wildcard()` is provided for determining the current wildcard state for an attribute.

This scheme is more flexible and intuitive to use than the original COBEA syntax for parameterised event filtering. In the old implementation, a string had to be passed together with the registration call that defines for which attributes parameterised filtering is to be used depending on the *order* of the attributes. For instance, a string like "␣*␣**" defines that no wildcards shall be used for the first and third attribute of the particular event type because of the position of spaces in the string. The new mechanism provided by this project is easier to use. The translation into the underlying string representation is performed automatically by the generated stub code.

3.4.4 Support for Superclass Registration

Superclass registration is a natural advantage of an object-oriented event type hierarchy. It allows the capture of the general behaviour of a number of derived event types by a single ancestor class. For example, Figure 3.6 shows a simple event type hierarchy. All the specialised sensor events are derived from a general event class, called SensorEvent, that stands for an abstract sensor. It is then possible to allow an event sink to register for the general SensorEvent and, as a consequence, be notified of the occurrence of all events which are *derived* from it, namely MagneticSensor, ElectronicSensor, OpticalSensor and GPSSensor.

In order to support such a scheme, changes to the original COBEA source code were necessary because in its original state, the COBEA system does not have the notion of hierarchical event types. Since the COBEA system consists of a large body of code, it was decided not to change the overall design of the event architecture to be based on object-orientation but to rewrite the internal signalling

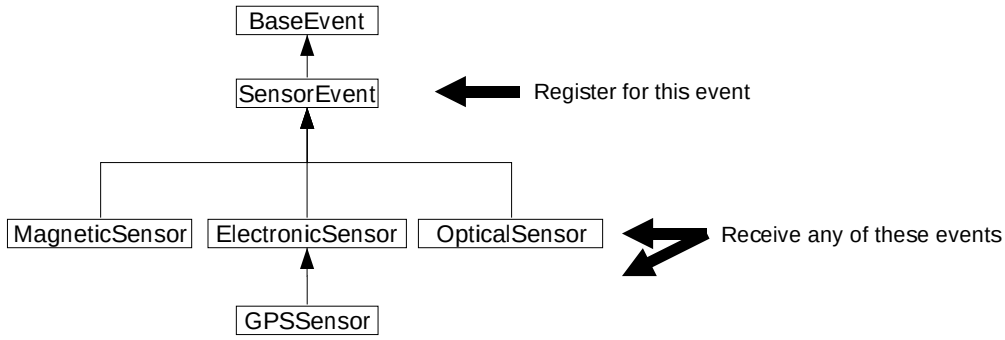

Figure 3.6: Example Event Type Hierarchy with Superclass Registration

mechanisms instead. In other words, the part of COBEA that performs the signalling call to an event sink was extended, so that it checks for any derived classes and performs notifications for these as well. The following paragraphs give a more detailed explanation of this mechanism, describing the code changes made and the data structures used to represent an event type hierarchy.

After the instance of the class EventSource has been created, all the event types that can be potentially generated by this event source need to be registered with COBEA by calls to the method

```
void EventSource::register_new_event(BaseEvent_Src& event)
```

beforehand. Internally, this method makes a call to

```
InheritanceSpec* BaseEvent_Src::getInheritanceSpec()
```

which returns a pointer to an InheritanceSpec data structure. This data structure contains information about the inheritance hierarchy of all the event type classes that have been generated by the event type compiler from the ODL input file. Its type is `list<string>` which is a list that contains the ancestor types of an event. As shown in Figure 3.7, for the event type `GPSSensor` from the example above, this list contains four entries. For every event class, this list is initialised in the constructors by statements automatically inserted by the event type compiler.

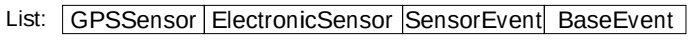

Figure 3.7: An Example InhertianceSpec for the Event GPSSensor

A new version of the method for registering event types with COBEA in the class FSrc_i has been provided that accepts an InheritanceSpec pointer as a parameter.

In addition to the update of the set of types allowed for event sink registration, this method updates the global hierarchy table for this event source. The type of this global event hierarchy table is

```
typedef map<string, set<string>* > TypeHierarchy;
```

which is a table that maps event type name strings to sets of strings which contain all the ancestors of an event.

Finally, when COBEA has to notify all registered event sinks of the occurrence of a particular event, it looks up the event type name in the global type hierarchy table and sends notifications for all descendent event types stored in the associated set as well. Although the association of all the descendents of an event type with every event type leads to a certain storage overhead, it has been adopted because of the efficiency obtained during signalling, which can be done in constant time by performing a single table look-up. A fast signalling mechanism is essential for any scalable event architecture. A further advantage of the chosen type hierarchy data structure is that hierarchy information is only added to the table but never updated so that the consistency of the table is always ensured.

Table 3.6 shows the state of the global event hierarchy table after all the event classes in Figure 3.6 have been registered with the event source.

Name of Event Class	Set of all Ancestors
BaseEvent	NIL
SensorEvent	BaseEvent
MagneticSensor	SensorEvent, BaseEvent
ElectronicSensor	SensorEvent, BaseEvent
GPSEvent	ElectronicSensor, SensorEvent, BaseEvent
OpticalSensor	SensorEvent, BaseEvent

Table 3.6: An Example TypeHierarchy Table as managed by COBEA

3.4.5 Marshalling of data types

The process of marshalling is essential for a distributed, object-oriented system. It describes the serialisation of complex data types into a form that can be transmitted over a network in a remote method call. The destination host reverses this process and hence obtains the original structured data type. The CORBA system that is responsible for the remote object method calls is capable of marshalling the data types provided by the Interface Definition Language (IDL). However, it is necessary to supply custom marshalling methods for all user-defined complex data types that can be part of an ODL event type declaration. This task is automatically done by the event type compiler that adds *insertion* and *extraction operators* to the generated C++ stub code for every complex ODL type declaration found in the file. These operators allow the transformation of the complex data type into the type any, which can then be passed on to CORBA and transmitted over the network. User-defined collections or hierarchical data types can be represented as a sequence of any. Such a scheme allows the serialisation of a large variety of types like, for example, arrays, nested structures, etc.

To illustrate the marshalling code, Figure 3.8 shows a complex ODL data type OuterS which is a nested structure and therefore requires custom marshalling support. The event type compiler

automatically generates any insertion and extraction operators that transform the data structure into a sequence of anys, as shown in Figure 3.9. In this example, the sequence of anys is nested in order to express inner structures. An example implementation of these operators can be found in the Application Scenario code in Appendix D.

```
attribute struct OuterS {
  string str1;
  struct InnerS {
    long a;
    long b;
  } inner;
} outer;
```

Figure 3.8: A Complex Attribute outer for which Custom Marshalling is necessary

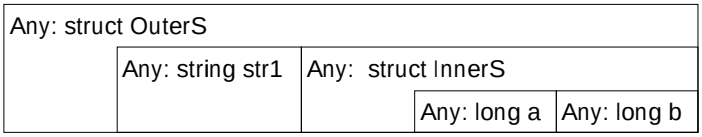

Figure 3.9: An Example of a Marshalled Data Type as a sequence of anys

3.5 Design of an ODL to XML Schema Mapping

The event type compiler is able to output XML declarations for the event classes in the ODL input file. In order to achieve this, the XML Schema Language is used which allows the declaration of structured data types. The XML Schema Specification is provided in two separate documents: [12] describes the syntax and structure of XML Schema definitions and its main features, whereas [13] concentrates on the data types and the ways of defining them by restriction.

Most of the syntactic ODL categories supported for event class definitions can be represented straight-forwardly with the XML Schema language. Classes can be seen as <element>s having an inner <type> structure that specifies the attributes and the parent <element>. For example, Figure 3.10 compares the ODL declaration of the class AlarmEvent with its XML Schema representation. For this translation to work, an exact mapping from the ODL data types to XML Schema data types is necessary, so that the semantics of event class declarations is preserved.

ODL:

```
class AlarmEvent extends SensorEvent {
  attribute string descr;
  attribute short priority;
};
```

XML Schema:

```
<element name = "AlarmEvent">
  <type source = "SensorEvent" derivedBy = "extension">
    <element name = "descr" type = "d_String"/>
    <element name = "priority" type = "d_Short"/>
  </type>
</element>
```

Figure 3.10: An XML Schema Definition of an ODL event class

3.5.1 Data Types

The Table 3.7 shows the data type mapping from ODL to XML Schema for all the currently supported types. Since the XML Schema standard is being developed with the goal of compatibility to existing standards in mind, most of the ODL types can be mapped to similar XML Schema types. The complex ODL types enum and struct which are enumerations and structures, respectively, have to be treated as special cases because they require more complex XML language constructs, as explained below.

A simple ODL type like d_Long can be declared in XML Schema using the feature of *constraints*. Constraints are special restrictions imposed on XML Schema data types by which new types can be defined. In Figure 3.11, the ODL-compliant d_Long type is declared by constraining the value range of the general XML Schema integer type to 32 bits. Another example is the ODL type d_Char for storing a single character which is represented in XML Schema by constraining the string type to hold a string of length one.

As part of this project, definitions for most ODL data types and the BaseEvent and Heartbeat event types have been designed and are listed in Appendix C. These definitions must be available in order to properly parse and validate the data types used by the event type compiler to create the XML Schema output file with the event class declarations.

3.5.2 Structures and Enumerations

An ODL struct construction can be expressed using a syntax that is similar to the declaration of classes. This is naturally the case because classes are an extended version of structures. The Figure 3.12 tries to show the general form of a structure declaration in XML Schema. These transformations are performed by the event type compiler when outputting XML Schema Definitions.

ODL Data Type	XML Schema Data Type	Description
d_Long	integer	32 bit signed integer
d_Short	integer	16 bit signed integer
d_ULong	non-negative-integer	32 bit unsigned integer
d_UShort	non-negative-integer	16 bit unsigned integer
d_Float	float	32 bit IEEE 754 floating-point number
d_Double	double	64 bit IEEE 754 floating-point number
d_Boolean	boolean	Boolean value
d_Octet	binary	8 bit value
d_Char	string	8 bit ASCII character
d_String	string	String
d_Date	date	Date
d_Time	time	Time
d_Timestamp	timeInstant	Time stamp
d_Interval	timeDuration	Time interval
enum	enumeration	Enumeration
struct	element	Structure

Table 3.7: ODL Types supported by the XML Schema Mapping

```
<datatype name = "d_Long" source = "integer">
  <minInclusive value = '-2147483648'/>
  <maxInclusive value = '2417483648'/>
</datatype>
```

Figure 3.11: XML Schema Definition for the ODL Type d_Long

Enumerations are formed by applying the <enumeration value = . . . /> constraint to a <datatype> declaration. This constraint only allows the assignment of a predefined set of values to the data type. Therefore, it is possible to build an enumeration of strings for any possible ODL enumeration, as shown in Figure 3.13.

3.5.3 Limitations of the XML Mapping

As mentioned previously, the XML Schema Specification is a very new standard that is still a "working draft". This means that not all of its aspects were completely fixed while implementing this project, and certain necessary facilities were not provided, yet. It was very challenging to work with a specification that contained so many "white gaps". The Table 3.8 gives an overview of the ODL data types that could not be expressed as XML Schema types due to these constraints.

The first three mentioned types that could not be expressed in XML are collection types. Although

```
<element name = "attribute name">
  <type name = "structure  name">
    <element name = "attribute name 1" type = "type 1"/>
    <element name = "attribute name 2" type = "type 2"/>
    ...
    <element name = "attribute name n" type = "type n"/>
  </type>
</element>
```

Figure 3.12: XML Schema Definition of an ODL structure

```
<datatype name = "enumeration name" source = "string">
  <enumeration value = "enumerator 1"/>
  <enumeration value = "enumerator 2"/>
  ...
  <enumeration value = "enumerator n"/>
</datatype>
```

Figure 3.13: XML Schema Definition of an ODL enum

it is planned by the W3C to include collections in the Schema specification, this has not been done by the time of implementing this project. For this reason, it has been decided that the event type compiler should ignore any collection attributes in ODL event class declarations when outputting XML Schema code. The last type in the table is the ODL type any that can hold any possible value. Such a type has been not defined by the XML Schema specification and cannot be easily provided in a platform independent way.

ODL Data Type	Description
d_Set	Set type
d_List	List type
d_Varray	Variable-size array type
d_Any	Any type

Table 3.8: ODL Types unsupported by the XML Schema Mapping

Finally, the Table 3.9 is a listing of all the ODL keywords and constructs that cannot be translated into XML Schema because of deficiencies of the language. It is not possible to define new type names for old types as it is done by the ODL typedef keyword. Moreover, constants cannot be easily included into XML Schema declarations because in XML the grammar and the data are kept separate.

For the same reason, no literal expressions can be added to an XML Schema.

ODL Construct	Description
typedef	Defines a type name alias
const	Define a constant attribute
{expressions}	Expressions used e.g. for constants
{literals}	Literals used in expressions

Table 3.9: ODL Language Constructs unsupported by the XML Schema Mapping

3.6 ODL2EventComp — The Event Type Compiler

The entire process of translating ODL event class descriptions into usable C++ source code was automated by the means of an event type compiler called ODL2EventComp. Although the compiler consists of a large body of code and formed a major part of the project, its implementation was greatly simplified by the careful design of the required mappings and transformations, as described in the previous sections. Therefore, the output files were clearly specified to the detail that hand-coded versions of the stub-code to be generated were available. During the entire implementation stage, the behaviour of the compiler could be directly evaluated against these hand-coded C++ and XML output files. As a result, less time was spend on the implementation of the compiler than on the design of the mappings.

The compiler consists of a lexer, a parser with an abstract syntax tree generator, a semantic analyser, a C++ code generator and an XML code generator (Figure 3.14). The code generator is a pluggable module, so that the XML and C++ output can be supported by a single compiler with a lot of code re-use between these two back-ends. In the following sections, I would like to describe the implementation work carried out and the design decisions made for the event type compiler.

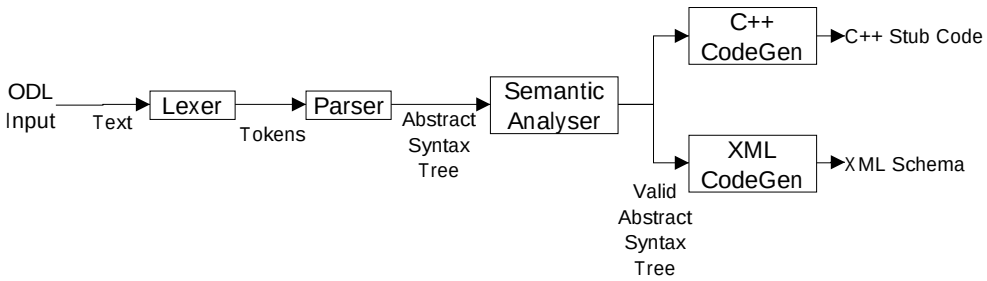

Figure 3.14: Internal Details of the Event Type Compiler ODL2EventComp

The compiler executable is called ODL2EventComp, and it is run by specifying an ODL input file as a parameter:

```
ODL2EventComp <ODL file name>.odl
```

The compiler follows the Unix tradition of terse output so that a successful compilation causes no output message. Otherwise an error message including a possible input file line number with the error condition is displayed. The compiler attempts to generate six C++ files and one XML output file with names adhering to the conventions in Section 2.2.1. Checks for the most common IO error conditions[4] are made.

3.6.1 Lexing and Parsing

The Unix utilities *flex* and *bison* were used for implementing this part of the compiler. The input files for flex and bison were written to comply with the BNF[5] grammar specified in [2]. This grammar contained several ambiguities and minor bugs that had to be spotted and corrected first. Example ODL code from [2] was used as test input to the lexer and parser in order to guarantee its correct behaviour.

The lexer is capable of validating the entire ODL grammar including keywords and constructs that are not legal in event type declarations. Although it made the design more complicated, this decision has the advantage that it is possible to notify the user about ODL syntax, that is valid in a general ODL file, but not allowed in event type declarations. Whenever the lexer encounters ODL keywords that cannot be used in event declarations like, for example, the keyword interface, the user is informed with a meaningful error message. Section 3.6.4 gives an overview of the possible error conditions raised by the event type compiler.

The lexer supports both C- and C++-style comments[6] in ODL, which is a deviation from the original ODL standard, but greatly facilitates the production of clear, human understandable ODL files.

The parser generates an AST[7] for the entire ODL input file, which is then passed on to the semantic analyser. Due to the large number of different syntactic categories in ODL, it was decided to implement the AST using a single class named AST_Node which has a NodeType field that defines the grammatical category of this node. A further list data structure contains the node's direct children, which are other AST nodes. As a result, the entire AST consists of lexical tokens that represent syntactic categories augmented with parameters. Parameters can be class names, attribute names and other identifiers encountered in the ODL source and different parameter types can be stored in an AST node by the use of C++ unions. In order to support debugging, the node class contains a method for printing the AST in human-readable form, as shown in Figure 3.15.

[4] For example, "File not found" etc.
[5] Backus-Naur Form
[6] /*...*/ and //
[7] Abstract Syntax Tree

```
[specification]
 [class (Name1:LocationEvent)]
  [class_header (Name1:LocationEvent)]
   [extends]
    [scoped_name (Name1:BaseEvent)]
  [interface_body]
   [attr_dcl (Name1:areaCode)]
    [char_type]
   [attr_dcl (Name1:locDescr)]
    [string]
   ...
```

Figure 3.15: Fragment of an AST generated from an ODL file

3.6.2 Semantic Analysis

The semantic analysis stage of the compiler attempts to determine whether the AST can be correctly transformed into a valid event class declaration. This analysis ensures several aspects, namely that

1. the ODL file only consists of class declarations, and, in particular, does not contain interfaces or modules, and that

2. all declared ODL classes are directly or indirectly derived from the ancestor class BaseEvent, and that

3. the structure of class declarations is valid so that, for example, classes do not extend themselves.

The semantic analysis is done by a traversal of the AST. During this traversal, it is checked that the syntactic category of each AST node is supported for ODL event hierarchy declarations. In addition to that, whenever a class declaration is encountered, the class name together with its ancestor class is stored in a table so that it can be checked whether there exist any classes which are not derived from BaseEvent.

3.6.3 Code Generation

The final stage of the compiler is the code generator that performs the actual file output. The class hierarchy of the two classes CodeGen_Cpp for C++ and CodeGen_XML for XML code generation is given in Figure 3.16. Both classes are derived from the abstract class CodeGen that contains the general functionality of a code generator like, for example, dealing with the output files and compiler header strings.

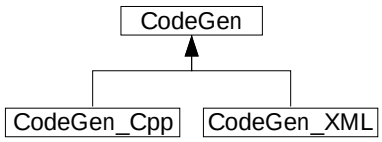

Figure 3.16: Classes for C++ and XML Code Generation

C++ Stub Code Generation

The code for the C++ stub code generation is relatively complex because, as described in Section 2.2.1, it is necessary to output six C++ header and implementation files for a single ODL input file. Preference was given to a solution where all the output files are created in parallel while doing a *single* traversal of the AST as opposed to a solution with several passes. The main reason for this decision is that serial file generation would have degraded performance and not necessarily made the compiler source code easier to develop.

Another difficulty with the C++ code generation is that it does not strictly follow the structure of the AST. For instance, marshalling and unmarshalling operators have to be generated for all user-defined ODL data types, which may require knowledge that is not locally available at the current point of the AST where the data type is first used. In such cases, the required data must be passed along in suitable method parameters while traversing the AST.

The code generation starts with the output of predefined headers for each file. After that, the AST is traversed and calls to a number of *generate methods* are made. These methods output lines of C++ stub code depending on the kind of the current AST node. For instance, the `genClass(...)` method outputs the C++ code for an event class declaration if the current AST node is an ODL class declaration. Some generate methods output code into specific stub files only, as for example the method `genStructType_main_h(...)` which produces the code for an ODL structure and writes it into the main header file. Data types are generated by `genType` methods that have switch constructs in order to deal with all ODL types.

Further generate methods are used to output code for attributes, structures, enumerations, type definitions, constant expressions, marshalling and unmarshalling operators, etc. The code generation for nested structures involves recursive calls within generate functions. Method parameters are supplied to these recursive calls that store information about the current level of nesting so that names of inner structures[8] are correctly generated. The same mechanism is used to ensure the correct indentation of nesting levels.

[8] like for example `OuterStruct::InnerStruct::InnermostStruct`

XML Schema Code Generation

The XML Code generator outputs a single file that directly mirrors the structure of the AST. Therefore, the implementation of the XML code generation was easier than of the C++ one, as an AST node often directly corresponds to an XML construct. The XML token output is done by the libxml library, that generates an XML parse tree by making calls to appropriate methods. If the XML code generator encounters an AST node that cannot be expressed in XML Schema, a warning message is displayed, and the node is simply skipped.

The main work is done in the `genNode(...)` method. This method looks at the current AST node and depending on its syntactic category, the appropriate output methods from the libxml library are called. These output methods build an XML parse tree in memory by adding XML tokens and properties into a tree structure so that the ODL→XML mapping as described in Section 3.5 is implemented. After the entire AST has been traversed, a call to the method `xmlDocDump` from the libxml library writes the XML tree into the output file.

Due to limitations of the libxml library, certain XML keywords like, for instance, DOCTYPE cannot be synthesised. In order to obtain a *valid* XML file, as opposed to only a *well-formed* one, these keywords have to be added to the final XML Schema declaration file before it can be analysed by a schema-aware XML parser.

3.6.4 Exceptions supported by the compiler

A significant number of different error conditions can arise for which the event type compiler generates exceptions. These exceptions are classified as being *fatal errors*, *errors*, and *warnings*, as shown in Figure 3.17. All exceptions are derived from the Comp_Exception base class and are part of an exception hierarchy. Three specialisations from the base class denote the three general kinds of exceptions:

Comp_Fatal_Exceptions are fatal compiler errors that should never occur and are caused by bugs in the compiler. They are normally raised by sanity checks inside the compiler.

Comp_Error_Exceptions are compiler errors that are caused by invalid ODL input and result in a failure of the current compilation.

Comp_Warning_Exceptions are warnings displayed by the compiler which may result in incorrect compilation output but are otherwise ignored.

Further sub-categories of exceptions exist for most compiler stages at which exceptions can arise, like for example the semantic analysis, XML code generation, and IO operations.

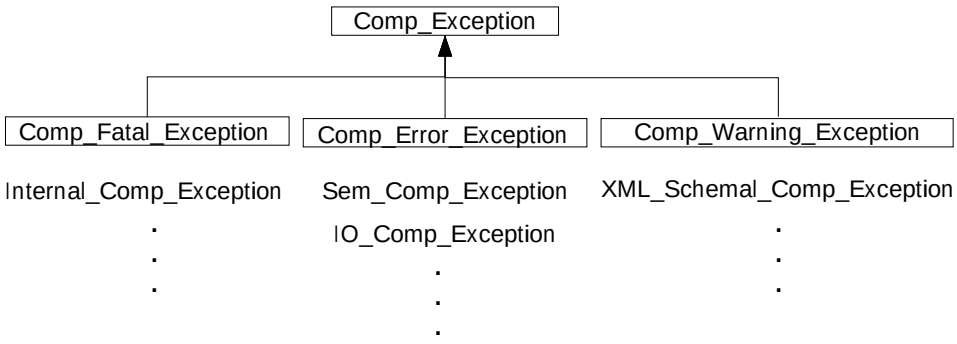

Figure 3.17: Hierarchy of Exceptions raised by ODL2EventComp

Chapter 4

Evaluation

Testing during the development phase and the evaluation afterwards were an important part of the project. In accordance with the original goals, the process of evaluation was carried out in several stages in which three different criteria of the final project result were investigated: Correctness, Usability and Maintainability. Qualitative evaluations of these criteria are described in the following sections.

4.1 Correctness

The main goal was correctness of the project which means that the generated event stub code behaves as expected from the ODL event type declaration written by the application programmer. Rigorous evaluation procedures were necessary to ensure the soundness of the transformations of the ODL input files. The complexity of the compiler body is mainly due to its size of approx. 7,800 lines of code.

The individual components were evaluated by detailed module testing. A special header file was written that allowed the quick insertion of structured debug print statements whose behaviour could be controlled by environment variables. Moreover, the GNU Debugger *GDB* was used to search for bugs in the source code and it turned out to be a very valuable tool.

In order to ensure the correctness of the entire project, semantically valid mappings from ODL to C++ stub code and a correctly working event type compiler that implements these mappings, were required. Both of these two aspects were evaluated with two *Event Application Scenarios*, which are informal descriptions of possible scenarios in which a distributed event architecture may be applicable. The two scenarios were carefully defined including all the required mappings and output files to be produced by the event type compiler. It must be stressed that this was done before the actual implementation of the compiler took place. After that, the output from the compiler could be compared with this ideal hand-coded output and modified when necessary.

4.2 Event Application Scenarios

The two Event Application Scenarios are carefully defined *model use cases* that represent possible applications of a distributed event architecture. Their definitions consist of an *informal description* of the application task, a formal *ODL declaration* of the events and the necessary *C++* and *XML stub code* usable within the programming model described in this project. The first scenario is rather simple and employs only a subset of all available ODL features for event class declarations, whereas the second scenario attempts to encompass the entire spectrum of ODL event definitions. Both scenarios make strong use of inheritance and a variety of different data types. After the compiler has generated the C++ stub code for a scenario, it should be possible to create a test application within a dummy application framework, as described in Section 4.2.2.

4.2.1 Application Scenario 1 (Simple)

Informal Description: The first scenario requires the application programmer to design and build a system to locate people within and around a building. People wear small electronic badges in their pockets that are capable of communicating with several base stations via a radio signal. These badges have sensors on them that recognise if a person enters or leaves a particular room. When a person is outside the building, a built-in GPS receiver inside the badge can find out about the exact position of the badge and transfers this information to one of the base stations.

The task is to engineer and built a distributed event system that manages an entire building with several base stations and a number of people wearing electronic badges. The system should be able to track every person and to notify interested parties about people entering or leaving certain rooms.

ODL Declaration: The Figure 4.1 shows a diagram of the classes in the ODL event class declaration file for this application. A general LocationEvent has been derived from the BaseEvent class that represents an event generated by a badge announcing its current location. Two specialised event types, namely RoomEvent and GPSEvent, denote a sensor event inside a room and a GPS location outside the building, respectively. The two events EntranceEvent and ExitEvent are generated by a person entering or leaving a room.

The entire ODL file, whose name is AppScen1.odl, can be found in Appendix D. A variety of different ODL constructs is used in this file and thus must be correctly transformed by the event type compiler.

C++ and XML Stub Code: The C++ and XML stub code to be generated from the ODL input file can be found in Appendix D as well. It consists of six C++ files, namely AppScen1.h, AppScen1.cc, AppScen1_Src.h, AppScen1_Src.cc, AppScen1_Snk.h and AppScen1_-Snk.cc which are the header and stub files for the event sinks and the event sources. In addition, the file AppScen1.xsd contains the XML Schema representation of the event declarations. In

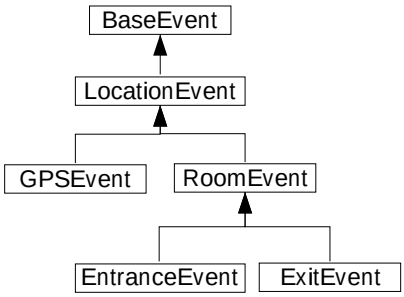

Figure 4.1: ODL Classes and Inheritance Relations for Application Scenario 1

this example, the event sources are the electronic badges that produce events, whereas sinks can register their interest in the location of certain persons.

The description of the second, more complex Application Scenario together with a diagram showing its inheritance hierarchy can be found in Appendix F.

4.2.2 Application Scenario Testing

The application scenarios were used to test and evaluate the correctness of the entire event type compiler after system integration. A large number of problems and faults were identified with the help of these two application scenarios.

The process of testing the application scenarios was automated by the means of *Makefiles* that initiated the build an example client/server application which incorporated the stub code files created by the event type compiler. Erroneous stub code output from the event type compiler could immediately make this compilation fail. The correct behaviour of the entire event application was tested by a dummy application framework consisting of an event source and an event sink. These files can be found in Appendix E.

The following Figure 4.2.2 is a screen shot of the output generated by an example event source and sink that exchange the events defined in the first application scenario. The event source output on the left-hand side is shown in parallel with the corresponding event sink output on the right-hand side so that simultaneous events share the same line in the screen shot.

The *Event Source* starts its initialisation by creating an EventSource object and five event objects of the types BaseEvent, GPSEvent, RoomEvent, EntranceEvent and ExitEvent. All these event objects are registered with the event source so that event sinks are allowed to register their interests in these events. After that, the event source signals a number of different events.

The *Event Sink* creates an EventSink object, first, and, then, registers several event objects with the event source in order to be notified of the occurrence of these events. The registration for the event class EntranceEvent includes a parameterised filter so that the attribute person of this event must equal the string “Fred”. After this registration process, the event sink waits for notifications.

The first event the event source signals is an EntranceEvent (with attribute person = 'Fred'). This causes the event sink to receive two events: An event of type EntranceEvent because the correct parameter is passed and another of type RoomEvent because of the feature of superclass notification. The next event signalled by the event source (EntranceEvent with person = 'John') is not received by the sink because of the filtering expression for this event type. The following events are dealt with in a similar manner. The column for the event sink gives a subset of all the attributes that were received with each event.

During the period of the screen shot, one Heartbeat event is received by the event sink. These events are send autonomously by the source and prove that the connection between the sink and source is still valid.

Event Source

```
Building event source 'Source1AP1'
Building BaseEvent
Registering BaseEvent with 'Source1AP1'
Building GPSEvent
Registering GPSEvent with 'Source1AP1'
Building RoomEvent
Registering RoomEvent with 'Source1AP1'
Building EntranceEvent
Registering EntranceEvent with 'Source1AP1'
Building ExitEvent
Registering ExitEvent with 'Source1AP1'

Signalling events.........

Signalling... EntranceEvent (person = 'Fred')




Signalling... EntranceEvent (person = 'John')
Signalling... ExitEvent (person = 'Alison')




Signalling... GPSEvent


Signalling... EntranceEvent (person = 'Fred')




Signalling... EntranceEvent (person = 'John')
```

Event Sink

```
Building event sink 'Sink1AP1' bound to 'Source1AP1'
Building Heartbeat
Registering Heartbeat (Sink1AP1->Source1AP1)
Building event GPSEvent
Registering GPSEvent (Sink1AP1->Source1AP1)
Building event RoomEvent
Registering RoomEvent (Sink1AP1->Source1AP1)
Building event EntranceEvent
Setting up parameterised filtering (person = 'Fred')
Registering EntranceEvent (Sink1AP1->Source1AP1)
Building event ExitEvent
Registering ExitEvent (Sink1AP1->Source1AP1)
Waiting for events............

Received EntranceEvent (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Library)(roomLoc.room = Reading Room)
                       (roomLoc.no = 1)(person = John)
Received RoomEvent     (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Library)(roomLoc.room = Reading Room)
                       (roomLoc.no = 1)

Received ExitEvent     (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Main Building)(roomLoc.room = Office)
                       (roomLoc.no = 223)(person = Alison)
Received RoomEvent     (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Main Building)(roomLoc.room = Office)
                       (roomLoc.no = 223)
Received GPSEvent      (areaCode = A)(locDescr = Car Parking)
                       (x = 123.456)(y = 42)(z = 67)
Received Heartbeat     (id = 20)(signal_time = 10/04/2000 - 19:48:00)(interval = 5)
Received EntranceEvent (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Library)(roomLoc.room = Reading Room)
                       (roomLoc.no = 1)(person = Fred)
Received RoomEvent     (areaCode = B)(locDescr = Building)
                       (roomLoc.building = Library)(roomLoc.room = Reading Room)
                       (roomLoc.no = 1)
```

Figure 4.2: Screen shot from an Event Source and Sink in the Application Scenario 1

The corresponding application framework for the second application scenario can be found in the two files TestSourceAP2.cc and TestSinkAP2.cc in Appendix G together with a similar screenshot to that given for the first scenario.

4.3 Usability

As mentioned in the original project proposal, usability of the object-oriented event class programming model was another of the major goals of this project. In this section, I would like to give a high-level comparison of the new programming model with the one that was provided by the previous COBEA implementation and point out the advantages of the new approach. The code excerpt in Figure 4.3 shows an old COBEA event source that creates an event type and signals it.

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB2"); // CORBA init
CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "omniORB2_BOA"); // CORBA init

FSrc_i * mySourceImpl = new FSrc_i(orb, "mysoure1",
                                   "localhost", FALSE, 1); // FSrc init
mySourceImpl->_obj_is_ready(boa);

BaseEventServer::FSrc_var mySourceImplRef = mySourceImpl->_this();
bindObjectToName(orb, mySourceImplRef,
                 "mysource1@localhost"); // Name service registration
boa->impl_is_ready(0, 1);
...
mySourceImpl->new_event("eventtype1"); // Registration of new event type
...
BaseEventServer::Property ps[3]; // Data structures for attributes
BaseEventServer::PropertySeq* pss;
ps[0].value <<= "Hello World"; // Set attributes
ps[0].len = 12; ps[0].name = "MyProperty1";

pss = new BaseEventServer::PropertySeq(1, 1, &ps[0],
                                       TRUE); // Packing attributes

BaseEventServer::EventHeader e; // Construct event header
e.event_type = "eventtype1";
e.source_id = "mysource1@localhost";
...
mySourceImpl->signal(e, *pss); // Signal event
```

Figure 4.3: Example Event Source as implemented using the old COBEA Programming Model

Two general features of this programming model can be noticed from this example: Firstly, in this model it is the duty of the application programmer to correctly do all the initialisation. Secondly, the event types including their attributes must be created from first principles and passed to the COBEA

signalling call. Everything has to be done manually by the programmer. It is important to note that the example code only creates a *single* event type with one attribute. Using such a scheme for a large number of hierarchical and dependent event types is very error-prone and probably not feasible.

In contrast to that, Figure 4.4 shows the code of an event source that uses the event programming paradigm that has been created by this project. It can be seen that this code is more compact and easier to understand. All initialisation tasks are done automatically by the instantiation of the EventSource class. The access of attributes is very intuitive because they are data fields of the event class EventType1. An intuitive object-oriented approach can be used. A similar comparison can be made for an event sink which reveals the same advantages of the new approach.

```
EventSource source1(argc, argv, "Source1AP1",
                    "localhost", 5); // Create EventSource object

EventType1_Src eventtype1; // Build event type object
source1.register_new_event(eventtype1); // Register with event source

eventtype1.MyProperty1 = "Hello World"; Set attribute
eventtype1.signal_event(source1); // Signal event
```

Figure 4.4: Example Event Source as implemented using the new Event Class Programming Paradigm

4.4 Maintainability

Using ODL declarations for event types improves the maintainability of an event software project. All the type declarations can be kept at a central location and updated when necessary. The event type compiler then generates the required stub code for all event sources and event sinks that are part of the distributed event applications. The changes to the event types are globally available to all programmers and a re-compilation with the new stub code leads to an up-to-date version of the event application. The addition of new event types to an ODL file requires a re-compilation but does not break the compatibility of any existing code.

Another reason why maintainability is improved is that all programmers have the same view of the event types they use. In the old programming model, event sources and sinks that wanted to exchange events were required to have the same knowledge about event types. However, this knowledge was not enforced by any mechanism so that many difficult to discover bugs could be introduced by incompatible event types being passed inside an event system.

Chapter 5

Conclusions

This chapter summaries the final project work and investigates the lessons that were learnt. The last section describes a number of sensible extensions to the project for the future.

5.1 Summary

The project has resulted in a successful implementation of an event type compiler for ODL guided by the rules of good software engineering practice. The compiler supports all the required mappings from ODL to C++ stub code and XML Schemas. The output of the compiler was evaluated and correctness was shown with two precisely defined Application Scenarios leading to a structured evaluation phase which focused on a number of acceptance criteria. The existing COBEA system was modified to introduce the notion of event type hierarchies and superclass registration. Therefore, it can be said that an easily applicable programming framework for distributed event applications has been created.

All the milestones listed in Section 2.4 were achieved, largely as expected. In comparison to the original project plan, only minor changes were necessary because of the better understanding of the project as it progressed.

5.2 Lessons Learnt

A number of important lessons were learnt from this project: Firstly, a considerable amount of experience was obtained in dealing with the COBEA system, the CORBA middleware, all the other applied systems and languages, and distributed event systems in general. The familiarisation phase was therefore an important part of the project and the initial time writing dummy applications and test code was well spent and allowed smooth implementation and evaluation phases.

Secondly, the structured software engineering approach for this project proved to be highly successful and made me realise the importance of good project management. The iterative development model with repeated refinement that was adopted for the design of the mappings was a very good

engineering paradigm for this kind of work. This facilitated the actual implementation of the event type compiler to a large extent.

Thirdly, it was realised how difficult it is to get an initial project proposal right the first time because it has to be written at a stage at which most for the future work is still very vague and many aspects cannot be predicted. For example, as mentioned previously in Section 2.4, it was decided to abandon the original plan of providing a Java code generator for the compiler. Before actually having implemented the C++ code generator, it was difficult to estimate the amount of work necessary for another language mapping with code generation.

5.3 Possible Extensions

This project allows a large number of interesting extensions. The whole area of distributed event systems is still evolving so that many different ideas have yet to be explored.

The current COBEA implementation supports *event mediators* which are entities in the event system that relay events from sources to sinks and registrations from sinks to sources. This allows a further separation of event producers and consumers because the entire management task is performed by the mediator. In its current version, the event type compiler does not support event mediators. However, it should be relatively straight-forward to add mediator support by providing an EventMediatior wrapper class, modifying the event type compiler and adding adequate interface methods to the stub code for event sources and sinks.

The actual COBEA system could be extended in a number of ways. For instance, *complex parameterised filtering expressions* could be allowed so that it would be possible to register for boolean expressions involving a number of event attributes. This extension would require a filtering expression parser and a change of the current signalling code of COBEA and the stub code interface methods. Another conceivable extension is to use *XML* as a fundamental way to express event types and event instances. Such a scheme would integrate nicely with event type repositories and inter-domain applications. Then, dynamic event type checking by an XML parser would be carried out during run-time as opposed to the static type-checking by the C++ compiler. However, this would require a major rewrite of several COBEA core components. The final issue I would like to mention is *access control*. A mechanism where only certain event sinks are allowed to register for a particular event source would allow the implementation of real-world security policies.

Bibliography

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build large scale distributed applications. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.

[2] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[3] S. Davis. An Implementation of the CORBA-Based Event Architecture. CST Part II Project, 1998.

[4] B. Leung. Dynamic Addition of New Event Types in COBEA, a CORBA-Based Event Architectur. CST Part II Project, 1999.

[5] S.-L. Lo and D. Riddoch. *The omniORB2 version 2.8 User's Guide*. AT&T Laboratories Cambridge, September 1999.

[6] C. Ma and J. Bacon. Cobea: A CORBA-Based Event Architecture. In *Proc USENIX COOTS'98*, pages 117–131, 1998.

[7] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 2.3.1, Object Management Group, October 1999.

[8] B. Stroustrup. What is object-oriented programming? (1991 revised version). In *Proc. 1st European Software Festival*, 1991.

[9] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[10] A. Vogel, B. Vasudevan, M. Benjamin, and T. Villalba. *C++ Programming with CORBA*. Wiley Computer Publishing, 1999.

[11] W3C. Extensible Markup Language (XML) 1.0. W3c recommendation, W3C, February 1998.

[12] W3C. XML Schema Part 1: Structures. Working draft, W3C, December 1999.

[13] W3C. XML Schema Part 2: Datatypes. Working draft, W3C, December 1999.

Appendix A

Listing of the COBEA classes and methods

A.1 IDL Definitions in FSrc

```
interface FSrc {
  void reg_event( // Registers an event interest of a sink with this source
    in BaseEventServer::EventHeader e, // Event header to be registered
    in string filter_header, // Filter header (not used)
    in PropertySeq event_body, // Attribute data
    in string filter_body, // String for parameterised filtering
    in string host, // Host name to be registered
    in string server, // Server name to be registered
    in BaseEventServer::Duration d, // Duration (not used)
    in string QoS, // Quality of Service data (not used)
    in string who, // Access control (not used)
    out long uid, // UID used for identification
    out long eid) // EID used for identification
  raises (RegistrationFailed, // Registration has failed
          NameConflict); // Previous Registration with same name

  void dereg_event( // De-registers a previous event interest
    in string user, // Sink name
    in long eid) // EID of registration
  raises (UserNotFound, // Sink name not found
          EventNotFound); // Event name not found
};
```

A.2 IDL Definitions in Snk

```
interface Snk : NotifyPublish {
```

```
  void notify( // Called to notify sink of event occurrence
    in EventHeader e,  // Event header
    in any data)  // Attribute data
    raises (NotConnected); // Called sink not registered

  void disconnect_snk(); // Disconnect a registered sink
}
```

A.3 Implementation of Event Source in FSrc_i

```
virtual void reg_event(...) // Registers an event interest of a sink with this source
  throw (BaseEventServer::RegistrationFailed,
         BaseEventServer::NameConflict);

virtual void dereg_event(...) // De-registers a previous event interest
  throw (BaseEventServer::UserNotFound,
         BaseEventServer::EventNotFound);

void signal( // Signals an event occurence to all registered event sinks
  BaseEventServer::EventHeader& e, // Event header
  const BaseEventServer::PropertySeq& pro); // Attribute data

void new_event( // Registers new event type with event source
  const char* ev_type) // String name of event type
throw (char*);

void delete_event( // Deletes previously registered event type
  const char* ev_type) // String name of event type
throw (char*);
```

A.4 Implementation of Event Sink in Snk_i

```
virtual void notify(...) // Called to notify sink of event occurrence
  throw(BaseEventServer::NotConnected);

virtual void disconnect_snk(); // Disconnects a registered sink
```

Appendix B

Unsupported ODL Keywords

Name of ODL Keyword	Function of keyword
extent	Specifies the set all of instances of a type within a database
key	Defines key to access objects in database
keys	same as key
module	Starts a module declaration
interface	Starts an interface declaration
bag	Data structure of unordered objects that may contain duplicates
union	Union data structure
switch	used for unions
case	used for unions
default	used for unions
readonly	readonly attribute similar to constant
relationship	Defines relationships between objects in database
inverse	Defines inverse relationships between objects in database
exception	Defines an exception
oneway	Defines an oneway method parameter
void	Defines an empty method return type
in	Defines an input method parameter
out	Defines an output method parameter
inout	Defines an input/output method parameter
raises	Declares exceptions raised by a method
context	Database-specific context

Appendix C

XML Schema Definitions of the ODL Data Types and BaseEvent

```
<?xml version = "1.0"?>
<!DOCTYPE schema PUBLIC "-//W3C/DTD XML Schema Version 1.0//EN"
 "http://www.w3.org/XML/Group/1999/09/23-xmlschema/structures/structures.dtd">

<schema>

<!-- Mapping of ODL data types -->

<datatype name = "d_Long" source = "integer">
  <minInclusive value = '-2147483648'/>
  <maxInclusive value = '2417483648'/>
</datatype>

<datatype name = "d_Short" source = "integer">
  <minInclusive value = '-32768'/>
  <maxInclusive value = '32768'/>
</datatype>

<datatype name = "d_ULong" source = "non-negative-integer">
  <maxInclusive value = '4294967295'/>
</datatype>

<datatype name = "d_UShort" source = "non-negative-integer">
  <maxInclusive value = '65535'/>
</datatype>

<datatype name = "d_Float" source = "float"/>
<datatype name = "d_Double" source = "double"/>
<datatype name = "d_Boolean" source = "boolean"/>
```

```
<datatype name = "d_Octet" source = "binary">
  <length value = '8'/>
  <encoding value = 'hex'/>
</datatype>

<datatype name = "d_Char" source = "string">
  <length value = '1'/>
</datatype>

<datatype name = "d_String" source = "string"/>
<datatype name = "d_Date" source = "date"/>
<datatype name = "d_Time" source = "time"/>
<datatype name = "d_Timestamp" source = "timeInstant"/>
<datatype name = "d_Interval" source = "timeDuration"/>

<!-- Definition of BaseEvent and Heartbeat -->

<element name = "BaseEvent">
  <type>
    <element name = "id" type = "d_Long"/>
    <element name = "priority" type = "d_Short"/>
    <element name = "source" type = "d_String"/>
    <element name = "signal_time" type = "d_Timestamp"/>
  </type>
</element>

<element name = "Heartbeat">
  <type>
    <element name = "interval" type = "d_Long"/>
  </type>
</element>
```

Appendix D

An Example COBEA Application — Scenario 1

D.1 AppScen1.odl

```
// AppScen1.odl - Definition
// Written by Peter Pietzuch <prp22@cam.ac.uk> 1999
// Application Scenario No. 1 - Input ODL file

// #include "BaseEvent.odl"

class LocationEvent extends BaseEvent {
  attribute char areaCode;
  attribute string locDescr;
};
class GPSEvent extends LocationEvent {
  attribute struct Coords {
    double x;
    double y;
    double z;
  } gpsLoc;
};
class RoomEvent extends LocationEvent {
  attribute struct RoomInfo {
    string building;
    string room;
    unsigned short no;
  } roomLoc;
};
class EntranceEvent extends RoomEvent {
  attribute string person;
};
class ExitEvent extends RoomEvent {
  attribute string person;
};
```

D.2 AppScen1.h

```
// AppScen1.h
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#ifndef AppScen1_H
#define AppScen1_H

#include "BaseEvent.h"
[...]

class LocationEvent : public virtual BaseEvent {
 public:
  d_Char areaCode; /* 1. property */
  d_String locDescr; /* 2. property */

  virtual const char* getTypeName() {return "LocationEvent";};
  virtual const unsigned int getPropertyCount() {return 2 +
                                          BaseEvent::getPropertyCount();};

 protected:
  LocationEvent() : BaseEvent() {};

  virtual int getProperties(BaseEventServer::Property properties[]);
  int setProperties(const BaseEventServer::Property properties[]);
};

class GPSEvent : public virtual LocationEvent {
 public:
  struct Coords {
    d_Double x;
    d_Double y;
    d_Double z;
  } gpsLoc; /* 1. property */

  virtual const char* getTypeName() {return "GPSEvent";};
  virtual const unsigned int getPropertyCount() {return 1 +
                                          LocationEvent::getPropertyCount();};

 protected:
  GPSEvent() : LocationEvent() {};

  virtual int getProperties(BaseEventServer::Property properties[]);
  int setProperties(const BaseEventServer::Property properties[]);

  friend void operator<<=(CORBA::Any& __a, const Coords& _Coords);
```

```
  friend CORBA::Boolean operator>>=(const CORBA::Any& __a, Coords& _Coords);
};

class RoomEvent : public virtual LocationEvent {
  [...]
};

[...]
#endif
```

D.3 AppScen1.cc

```
// AppScen1.cc
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#include "AppScen1.h"

int LocationEvent::getProperties(BaseEventServer::Property properties[]) {
  int propNum = BaseEvent::getProperties(properties);
  properties[propNum + 0].name = "areaCode";
  properties[propNum + 0].value <<= areaCode;
  properties[propNum + 0].len = sizeof(areaCode);
  properties[propNum + 1].name = "locDescr";
  properties[propNum + 1].value <<= locDescr;
  properties[propNum + 1].len = sizeof(locDescr);
  return LocationEvent::getPropertyCount();
};

int LocationEvent::setProperties(const BaseEventServer::Property properties[]) {
  int propNum = BaseEvent::setProperties(properties);
  properties[propNum + 0].value >>= areaCode;
  properties[propNum + 1].value >>= locDescr;
  return LocationEvent::getPropertyCount();
};

int GPSEvent::getProperties(BaseEventServer::Property properties[]) {
  int propNum = LocationEvent::getProperties(properties);
  properties[propNum + 0].name = "gpsLoc";
  properties[propNum + 0].value <<= gpsLoc;
  properties[propNum + 0].len = sizeof(gpsLoc);
  return GPSEvent::getPropertyCount();
};

int GPSEvent::setProperties(const BaseEventServer::Property properties[]) {
  int propNum = LocationEvent::setProperties(properties);
```

```
  properties[propNum + 0].value >>= gpsLoc;
  return GPSEvent::getPropertyCount();
};

void operator<<=(CORBA::Any& __a, const GPSEvent::Coords& _Coords) {
  int size = 3;
  CORBA::Any* any_array = new CORBA::Any[size];
  any_array[0] <<= _Coords.x;
  any_array[1] <<= _Coords.y;
  any_array[2] <<= _Coords.z;
  AnySeq anyseq(size, size, &(any_array[0]), true);
  __a <<= anyseq;
};

CORBA::Boolean operator>>=(const CORBA::Any& __a, GPSEvent::Coords& _Coords) {
  const AnySeq* anyseq;
  if (!(__a >>= anyseq))
    return false;
  if (!((*anyseq)[0] >>= _Coords.x))
    return false;
  if (!((*anyseq)[1] >>= _Coords.y))
    return false;
  if (!((*anyseq)[2] >>= _Coords.z))
    return false;
  return true;
};

[...]
```

D.4 AppScen1_Src.h

```
// AppScen1_Src.h
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#ifndef AppScen1SRC_H
#define AppScen1SRC_H

#include "AppScen1.h"
#include "BaseEvent_Src.h"

class LocationEvent_Src : public virtual LocationEvent, public BaseEvent_Src {
 public:
  LocationEvent_Src();
  ~LocationEvent_Src() {};
};
```

```
class GPSEvent_Src : public virtual GPSEvent, public LocationEvent_Src {
 public:
  GPSEvent_Src();
  ~GPSEvent_Src() {};
};

class RoomEvent_Src : public virtual RoomEvent, public LocationEvent_Src {
  [...]
};

[...]

#endif
```

D.5 AppScen1_Src.cc

```
// AppScen1_Src.cc
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#include "AppScen1_Src.h"

LocationEvent_Src::LocationEvent_Src() : LocationEvent(), BaseEvent_Src() {
  inhSpec->push_back("LocationEvent");
};

GPSEvent_Src::GPSEvent_Src() : GPSEvent(), LocationEvent_Src() {
  inhSpec->push_back("GPSEvent");
};

RoomEvent_Src::RoomEvent_Src() : RoomEvent(), LocationEvent_Src() {
  inhSpec->push_back("RoomEvent");
};

[...]
```

D.6 AppScen1_Snk.h

```
// AppScen1_Snk.h
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
```

```
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#ifndef AppScen1SNK_H
#define AppScen1SNK_H

#include "AppScen1.h"
#include "BaseEvent_Snk.h"
#include "EventSink.h"

class LocationEvent_Snk : public virtual LocationEvent, public BaseEvent_Snk {
 public:
  LocationEvent_Snk() : LocationEvent(), areaCode_wildcard(true),
                        locDescr_wildcard(true) {};

  void register_event_interest(EventSink& sink, const BaseEvent_CB* cbfunc);
  bool get_areaCode_wildcard() {return areaCode_wildcard;};
  void set_areaCode_wildcard(bool state) {areaCode_wildcard = state;};
  bool get_locDescr_wildcard() {return locDescr_wildcard;};
  void set_locDescr_wildcard(bool state) {locDescr_wildcard = state;};

 protected:
  const char* getFilterExpression();

 private:
  bool areaCode_wildcard;
  bool locDescr_wildcard;

  static BaseEvent_CB* getCallbackFunction(const char* source_name);
  friend void* cb_LocationEvent(void* vp);
  static map<string, BaseEvent_CB*, less<string> > callback_functions;
};

class GPSEvent_Snk : public virtual GPSEvent, public LocationEvent_Snk {
 public:
  GPSEvent_Snk() : GPSEvent(), gpsLoc_wildcard(true) {};

  void register_event_interest(EventSink& sink, const BaseEvent_CB* cbfunc);
  bool get_gpsLoc_wildcard() {return gpsLoc_wildcard;};
  void set_gpsLoc_wildcard(bool state) {gpsLoc_wildcard = state;};

 protected:
  const char* getFilterExpression();

 private:
  bool gpsLoc_wildcard;

  static BaseEvent_CB* getCallbackFunction(const char* source_name);
  friend void* cb_GPSEvent(void* vp);
  static map<string, BaseEvent_CB*, less<string> > callback_functions;
};
```

```
class RoomEvent_Snk : public virtual RoomEvent, public LocationEvent_Snk {
  [...]
};

[...]

#endif
```

D.7 AppScen1_Snk.cc

```
// AppScen1_Snk.cc
//
// ODL Event Mapping generated by ODL2EventComp
// Written by Peter Pietzuch <prp22@cam.ac.uk> 2000
//
// This file has been machine-generated.
// All changes will be lost. DO NOT EDIT!

#include "AppScen1_Snk.h"

map<string, BaseEvent_CB*, less<string> > LocationEvent_Snk::callback_functions;

void LocationEvent_Snk::register_event_interest(EventSink& sink, const BaseEvent_CB* cb) {
  BaseEventServer::Duration duration = {{0, 0}, {0, 0}}; // this is not used by COBEA
  const char* filter = getFilterExpression();
  LocationEvent_Snk::callback_functions[sink.getSourceName()] = cb;
  BaseEventServer::Property prop[getPropertyCount()];
  getProperties(prop);
  BaseEventServer::PropertySeq* propseq;
  propseq = new BaseEventServer::PropertySeq(getPropertyCount(), getPropertyCount(),
                                             prop, true);
  sink.register_event_interest(getTypeName(), priority, "", *propseq, filter,
                               duration, "", cb_LocationEvent);
};

const char* LocationEvent_Snk::getFilterExpression() {
  string filter;
  filter += BaseEvent_Snk::getFilterExpression();
  if (areaCode_wildcard)
    filter+="*";
  else
    filter+= " ";
  if (locDescr_wildcard)
    filter+="*";
  else
    filter+= " ";
  return filter.c_str();
};

BaseEvent_CB* LocationEvent_Snk::getCallbackFunction(const char* source_name) {
```

```
  return LocationEvent_Snk::callback_functions[source_name];
};

void* cb_LocationEvent(void* vp) {
  callback_data* cbd = (callback_data*) vp;
  const char* src_id = cbd->src_id.c_str();

  BaseEventServer::PropertySeq* pss;
  cbd->data >>= pss;
  BaseEventServer::Property* properties;
  properties = &(*pss)[0];

  LocationEvent_Snk* o_LocationEvent = new LocationEvent_Snk();
  o_LocationEvent->setProperties(properties);
  o_LocationEvent->id = cbd->n;
  o_LocationEvent->source = src_id;
  o_LocationEvent->signal_time = d_Timestamp::current();

  BaseEvent_CB* cb = LocationEvent_Snk::getCallbackFunction(src_id);
  cb(o_LocationEvent);
  delete o_LocationEvent;

  return vp;
};

map<string, BaseEvent_CB*, less<string> > GPSEvent_Snk::callback_functions;

void GPSEvent_Snk::register_event_interest(EventSink& sink, const BaseEvent_CB* cb) {
  [...]
};

const char* GPSEvent_Snk::getFilterExpression() {
  [...]
};

BaseEvent_CB* GPSEvent_Snk::getCallbackFunction(const char* source_name) {
  return GPSEvent_Snk::callback_functions[source_name];
};

void* cb_GPSEvent(void* vp) {
  [...]
};

[...]
```

D.8 AppScen1.xsd

```
<?xml version="1.0"?>
<!--XML Schema Event Hierarchy Definition-->
<!---->
```

```
<!--ODL Event Mapping generated by ODL2EventComp-->
<!---->
<!--Written by Peter Pietzuch <prp22@cam.ac.uk> 2000-->
<!---->
<!--This file has been machine-generated.-->
<!--All changes will be lost.-->
<!--DO NOT EDIT!-->
<!---->
<schema>
  <element name="LocationEvent">
    <type source="BaseEvent" derivedBy="extension">
      <element name="areaCode" type="d_char"/>
      <element name="locDescr" type="d_string"/>
    </type>
  </element>
  <element name="GPSEvent">
    <type source="LocationEvent" derivedBy="extension">
      <element name="gpsLoc">
        <type name="Coords">
          <element type="d_double" name="x"/>
          <element type="d_double" name="y"/>
          <element type="d_double" name="z"/>
        </type>
      </element>
    </type>
  </element>
  <element name="RoomEvent">
    <type source="LocationEvent" derivedBy="extension">
      <element name="roomLoc">
        <type name="RoomInfo">
          <element type="d_string" name="building"/>
          <element type="d_string" name="room"/>
          <element type="d_ushort" name="no"/>
        </type>
      </element>
    </type>
  </element>
  <element name="EntranceEvent">
    <type source="RoomEvent" derivedBy="extension">
      <element name="person" type="d_string"/>
    </type>
  </element>
  <element name="ExitEvent">
    <type source="RoomEvent" derivedBy="extension">
      <element name="person" type="d_string"/>
    </type>
  </element>
</schema>
```

Appendix E

Dummy Application Framework — Scenario 1

E.1 TestSourceAP1.cc

```
// This program is a framework for an event source in the Application Scenario 1
//
// Written by prp22@cam.ac.uk 14.1.2000

#include "AppScen1_Src.h"
#include "EventSource.h"

#include <iostream>
#include <omniORB2/CORBA.h>

int main(int argc, char* argv[]) {
  cout << "Building event source 'Source1AP1'" << endl;
  EventSource source1(argc, argv, "Source1AP1", "localhost", 5);

  cout << "Building BaseEvent" << endl;
  BaseEvent_Src baseevent;
  cout << "Registering BaseEvent with 'Source1AP1'" << endl;
  source1.register_new_event(baseevent);

  cout << "Building GPSEvent" << endl;
  GPSEvent_Src gpsevent;
  cout << "Populating GPSEvent with values" << endl;
  gpsevent.areaCode = 'A';
  gpsevent.locDescr = "Car Parking";
  gpsevent.gpsLoc.x = 123.456;
  gpsevent.gpsLoc.y = 42.000;
  gpsevent.gpsLoc.z = 67.000;
  cout << "Registering GPSEvent with 'Source1AP1'" << endl;
  source1.register_new_event(gpsevent);
```

```
  cout << "Building RoomEvent" << endl;
  RoomEvent_Src roomevent;
  cout << "Registering RoomEvent with 'Source1AP1'" << endl;
  source1.register_new_event(roomevent);

  cout << "Building EntranceEvent" << endl;
  EntranceEvent_Src entranceevent;
  cout << "Populating EntranceEvent with values" << endl;
  entranceevent.areaCode = 'B';
  entranceevent.locDescr = "Building";
  entranceevent.roomLoc.building = "Library";
  entranceevent.roomLoc.room = "Reading Room";
  entranceevent.roomLoc.no = 1;
  entranceevent.person = "Fred";
  cout << "Registering EntranceEvent with 'Source1AP1'" << endl;
  source1.register_new_event(entranceevent);

  cout << "Building ExitEvent" << endl;
  ExitEvent_Src exitevent;
  cout << "Populating ExitEvent with values" << endl;
  [...]
  cout << "Registering ExitEvent with 'Source1AP1'" << endl;
  source1.register_new_event(exitevent);

  cout << endl << "Signalling Events..." << endl << endl;

  while(true) {
    cout << "Signalling... GPSEvent" << endl;
    gpsevent.signal_event(source1);
    omni_thread::sleep(1);
    cout << "Signalling... EntranceEvent (person = 'Fred')" << endl;
    entranceevent.person = "Fred";
    entranceevent.signal_event(source1);
    omni_thread::sleep(1);
    cout << "Signalling... EntranceEvent (person = 'John')" << endl;
    entranceevent.person = "John";
    entranceevent.signal_event(source1);
    omni_thread::sleep(1);
    cout << "Signalling... ExitEvent (person = 'Alison')" << endl;
    exitevent.signal_event(source1);
    omni_thread::sleep(1);
    };
  exit(0); return 0;
};
```

E.2 TestSinkAP1.cc

```
// This program is a framework for an event sink in the Application Scenario 1
//
// Written by prp22@cam.ac.uk 14.1.2000
```

```
#include "AppScen1_Snk.h"
#include "EventSink.h"

#include <iostream>

void* disconnect_fn(void*);
void gpsevent_cb(BaseEvent_Snk*);
[...]
void heartbeat_cb(Heartbeat_Snk*);

void gpsevent_cb(BaseEvent_Snk* bevent) {
  cout << "Received GPSEvent      ";
  cout << "(id = " << bevent->id << ")";
  cout << "(signal_time = " << bevent->signal_time << ")";

  GPSEvent_Snk* gevent = (GPSEvent_Snk*) bevent;
  cout << "(areaCode = " << gevent->areaCode << ")";
  cout << "(locDescr = " << gevent->locDescr << ")" << endl;
  cout << "                        (x = " << gevent->gpsLoc.x << ")";
  cout << "(y = " << gevent->gpsLoc.y << ")" << endl;
  cout << "                        (z = " << gevent->gpsLoc.z << ")" << endl;
};

void roomevent_cb(BaseEvent_Snk* bevent) {
  [...]
};

void entranceevent_cb(BaseEvent_Snk* bevent) {
  cout << "Received EntranceEvent ";

  EntranceEvent_Snk* enevent = (EntranceEvent_Snk*) bevent;
  cout << "(areaCode = " << enevent->areaCode << ")";
  cout << "(locDescr = " << enevent->locDescr << ")" << endl;
  cout << "                        (roomLoc.building = " << enevent->
                                                    roomLoc.building << ")";
  cout << "(roomLoc.room = " << enevent->roomLoc.room << ")" << endl;
  cout << "                        (roomLoc.no = " << enevent->roomLoc.no << ")";
  cout << "(person = " << enevent->person << ")" << endl;
};

void exitevent_cb(BaseEvent_Snk* bevent) {
  [...]
};

void heartbeat_cb(Heartbeat_Snk* bevent) {
  [...]
};

void* disconnect_fn(void* vp) {
  [...]
};
```

```
int main(int argc, char** argv) {
  cout << "Building event sink 'Sink1AP1' bound to 'Source1AP1'" << endl;
  EventSink sink1 (argc, argv, "Sink1AP1", "localhost", "Source1AP1",
                                           "localhost", disconnect_fn);

  cout << "Building Heartbeat" << endl;
  Heartbeat_Snk heartbeat;
  cout << "Registering Heartbeat (Sink1AP1->Source1AP1)" << endl;
  heartbeat.register_event_interest(sink1, heartbeat_cb);

  cout << "Building event GPSEvent" << endl;
  GPSEvent_Snk gpsevent;
  cout << "Registering GPSEvent (Sink1AP1->Source1AP1)" << endl;
  gpsevent.register_event_interest(sink1, gpsevent_cb);

  cout << "Building event RoomEvent" << endl;
  RoomEvent_Snk roomevent;
  cout << "Registering RoomEvent (Sink1AP1->Source1AP1)" << endl;
  roomevent.register_event_interest(sink1, roomevent_cb);

  cout << "Building event EntranceEvent" << endl;
  EntranceEvent_Snk entranceevent;
  cout << "Setting up paramterised filtering (person = 'Fred')" << endl;
  entranceevent.person = "Fred";
  entranceevent.set_person_wildcard(false);
  cout << "Registering EntranceEvent (Sink1AP1->Source1AP1)" << endl;
  entranceevent.register_event_interest(sink1, entranceevent_cb);

  cout << "Building event ExitEvent" << endl;
  ExitEvent_Snk exitevent;
  cout << "Registering ExitEvent (Sink1AP1->Source1AP1)" << endl;
  exitevent.register_event_interest(sink1, exitevent_cb);

  cout << "Sleeping for 15 seconds" << endl;
  omni_thread::sleep(15);

  cout << "Deregistering Heartbeat (Sink1AP1)" << endl;
  heartbeat.deregister_event_interest(sink1);
[...]
  cout << "Deregistering ExitEvent (Sink1AP1)" << endl;
  exitevent.deregister_event_interest(sink1);

  omni_thread::sleep(3);

  cout << "Exiting main task" << endl;
  exit(0); return 0;
};
```

Appendix F

An Example COBEA Application — Scenario 2

F.1 Description of the Scenario

Informal Description: The second application scenario deals with a surveillance system that is composed of a number of sensors. These sensors are distributed around a building and can send messages to base stations. Different types of sensors are used, namely magnetic, optical and electronic sensors. The purpose of the sensor system is to detect intruders and to pass measurement data on to the main system for processing. Different types of data is measured by different sensors. When intruders are detected, additional information about them is available that needs to be transfered to the main system, as well.

The task is to design a system that controls all the sensors and is able to react quickly to the presence of intruders. All the measured data shall be correctly transfered from the sensors to the main system.

ODL Declaration: The event classes that can be used to implement this application scenario are shown in Figure F.1. All the events used in this example are generated by sensors and thus derived from a general SensorEvent. This type contains the time stamp of the measurement and the kind of sensor that performed it. Three events are derived from the SensorEvent: AlarmEvent signals the detection of intruders and contains a series of nested structures to store data. The two events InfoEvent_A and InfoEvent_B are different informational sensor events that can be distinguished by constant attributes. They carry a set of integer or floating-point data.

The entire ODL declaration of the file AppScen2.odl is listed in the following Section. It can be seen that this application scenario is more complex than the first one because of the usage of nested structures, enumerations, constants, type definitions and expressions.

C++ and XML Stub Code: The stub code for this scenario is written into seven files whose names begin with AppScen2. Especially the C++ stub code is much more complex than the code for

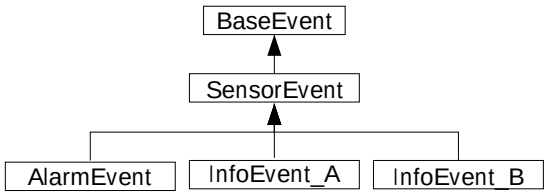

Figure F.1: ODL Classes and Inheritance Relations for Application Scenario 2

the first scenario because several marshalling operators for the user-defined data structures have to be synthesised by the event type compiler. Partly because of this additional complexity, it is not possible to equivalently express the output in XML Schema. This means that the compiler issues several warning messages when generating the XML output which is then invalid.

F.2 AppScen2.odl

```
// AppScen2.odl - Definition
//
// Written by Peter Pietzuch <prp22@cam.ac.uk> 1999
// Application Scenario No. 2 - Input ODL file

// #include "BaseEvent.odl"

class SensorEvent extends BaseEvent {
  attribute interval i;
  attribute timestamp ts;

  attribute enum SensorType {
    optical,
    acoustic,
    magnetic
  } sensortype;
};

class AlarmEvent extends SensorEvent {
  enum Importance {
    high,
    medium,
    low
  };

  struct Intruder {
    string name;
    unsigned long referenceNo;
  };

  attribute struct AlarmDescr {
```

```
    Importance priority;

    string location;
    list<string> intruders;
    any additionalData;

    struct IntruderDescr {
      char internalCode;
      list<Intruder> intruderList;
    } intruderDescr;
  } descr;
};

class InfoEvent_A extends SensorEvent {
  const string info_type = "A";
  const double info_id = 100 + 1;

  typedef set<long> LongSet;
  attribute LongSet data_values;
};

class InfoEvent_B extends SensorEvent {
  const string info_type = "B";
  const double info_id = 100 + 2;

  typedef set<double> DoubleSet;
  attribute DoubleSet data_values;
};
```

Appendix G

Dummy Application Framework — Scenario 2

G.1 TestSourceAP2.cc

```
// This program is a framework for the Application Scenario 2
//
// Written by prp22@cam.ac.uk 23.1.2000

#include "AppScen2_Src.h"
#include "EventSource.h"
[...]
#include <omniORB2/CORBA.h>

int main(int argc, char* argv[]) {
  cout << "Building BaseEvent" << endl;
  BaseEvent_Src baseevent;

  cout << "Building event source 'Source1AP2'" << endl;
  EventSource source1(argc, argv, "Source1AP2", "localhost", 5);

  cout << "Registering BaseEvent with 'Source1AP2'" << endl;
  source1.register_new_event(baseevent);

  cout << "Building SensorEvent" << endl;
  SensorEvent_Src sensorevent;
  cout << "Registering SensorEvent with 'Source1AP2'" << endl;
  source1.register_new_event(sensorevent);

  cout << "Building AlarmEvent" << endl;
  AlarmEvent_Src alarmevent;
  cout << "Populating AlarmEvent with values" << endl;
  alarmevent.ts = d_Timestamp(2000, 2, 1, 15, 00);
  alarmevent.sensortype = AlarmEvent_Src::optical;
  alarmevent.descr.priority = AlarmEvent::high;
```

```
  alarmevent.descr.location = "Warehouse 1";
  alarmevent.descr.intruders.insert_element("Thief A");
  alarmevent.descr.intruders.insert_element("Thief B");
  AlarmEvent_Src::Intruder myIntruder1, myIntruder2;
  myIntruder1.name = "Joe";
  myIntruder1.referenceNo = 123456;
  [...]
  alarmevent.descr.intruderDescr.internalCode = 'X';
  alarmevent.descr.intruderDescr.intruderList.insert_element(myIntruder1);
  alarmevent.descr.intruderDescr.intruderList.insert_element(myIntruder2);
  cout << "Registering AlarmEvent with 'Source1AP2'" << endl;
  source1.register_new_event(alarmevent);

  cout << "Building InfoEvent_A" << endl;
  InfoEvent_A_Src infoevent_a;
  cout << "Populating InfoEvent_A with values" << endl;
  infoevent_a.ts = d_Timestamp(2000, 2, 1, 16, 00);
  infoevent_a.sensortype = InfoEvent_A::magnetic;
  infoevent_a.data_values.insert_element(1);
  [...]
  cout << "Registering InfoEvent_A with 'Source1AP2'" << endl;
  source1.register_new_event(infoevent_a);

  cout << "Building InfoEvent_B" << endl;
  [...]
  cout << "Registering InfoEvent_B with 'Source1AP2'" << endl;
  source1.register_new_event(infoevent_b);

  cout << "Signalling events........" << endl;

  while(true) {
    [...]
  };
  exit(0); return 0;
};
```

G.2 TestSinkAP2.cc

```
// This program is a framework for the Appication Scenario 2
//
// Written by prp22@cam.ac.uk 23.1.2000

#include "AppScen2_Snk.h"
#include "EventSink.h"
[...]
#include <iostream>

void* disconnect_fn(void*);
[...]
void heartbeat_cb(Heartbeat_Snk*);
```

```
void sensorevent_cb(BaseEvent_Snk* bevent) {
  cout << "Received SensorEvent   ";
  SensorEvent_Snk* sevent = (SensorEvent_Snk*) bevent;
  cout << "(ts = " << sevent->ts << ")";
  cout << "(sensortype = " << sevent->sensortype << ")" << endl;
};

void alarmevent_cb(BaseEvent_Snk* bevent) {
  cout << "Received AlarmEvent    ";
  AlarmEvent_Snk* aevent = (AlarmEvent_Snk*) bevent;
  cout << "(ts = " << aevent->ts << ")";
  cout << "(sensortype = " << aevent->sensortype << ")" << endl;
  cout << "                       (descr.priority = " << aevent->descr.priority << ")";
  cout << "(descr.location = " << aevent->descr.location << ")" << endl;;

  d_Iterator<d_String> it = aevent->descr.intruders.begin();
  cout << "                       (descr.intruder = ";
  while (it != aevent->descr.intruders.end()) {
    cout << (*it) << " "; it++;
  };
  cout << ")" << endl;
[...]
};

[...]

int main(int argc, char** argv) {
  cout << "Building event sink 'Sink1AP2' bound to 'Source1AP2'" << endl;
  EventSink sink1 (argc, argv, "Sink1AP2", "localhost", "Source1AP2",
                                           "localhost", disconnect_fn);

  cout << "Building Heartbeat" << endl;
  Heartbeat_Snk heartbeat;
  cout << "Registering Heartbeat (Sink1AP2->Source1AP2)" << endl;
  heartbeat.register_event_interest(sink1, heartbeat_cb);

  cout << "Building event SensorEvent" << endl;
  [...]
  cout << "Registering InfoEvent_B (Sink1AP2->Source1AP2)" << endl;
  infoevent_b.register_event_interest(sink1, infoevent_b_cb);

  cout << "Waiting for events............" << endl;
  omni_thread::sleep(15);

  [...]

};
```

G.3 Screenshot of Application Scenario 2

Event Source

```
Building event source 'Source1AP2'
Building BaseEvent
Registering BaseEvent with 'Source1AP2'
Building SensorEvent
Registering SensorEvent with 'Source1AP2'
Building AlarmEvent
Registering AlarmEvent with 'Source1AP2'
Building InfoEvent_A
Registering InfoEvent_A with 'Source1AP2'
Building InfoEvent_B
Registering InfoEvent_B with 'Source1AP2'

Signalling events.............

Signalling... InfoEvent_A (Source1AP2)


Signalling... InfoEvent_B (Source1AP2)



Signalling... AlarmEvent (Source1AP2)
```

Event Sink

```
Building event sink 'Sink1AP2' bound to 'Source1AP2'
Building Heartbeat
Registering Heartbeat (Sink1AP2->Source1AP2)
Building event SensorEvent
Registering SensorEvent (Sink1AP2->Source1AP2)
Building event AlarmEvent
Registering AlarmEvent (Sink1AP2->Source1AP2)
Building event InfoEvent_A
Registering InfoEvent_A (Sink1AP2->Source1AP2)
Building event InfoEvent_B
Registering InfoEvent_B (Sink1AP2->Source1AP2)

Waiting for events.........

Received InfoEvent_A   (ts = 01/02/2000 - 16:00:00)(sensortype = 2)
                       (info_type = A)(info_id = 101)
                       (data_values = 1 2 3 )
Received SensorEvent   (ts = 01/02/2000 - 16:00:00)(sensortype = 2)
Received InfoEvent_B   (ts = 01/02/2000 - 17:00:00)(sensortype = 1)
                       (info_type = B)(info_id = 102)
                       (data_values = 1.1 2.2 3.3 )
Received SensorEvent   (ts = 01/02/2000 - 17:00:00)(sensortype = 1)
Received AlarmEvent    (ts = 01/02/2000 - 15:00:00)(sensortype = 0)
                       (descr.priority = 0)(descr.location = Warehouse 1)
                       (descr.intruder = Thief A Thief B )
                       (descr.intruderDescr.internalCode = X)
                       (descr.intruderDescr.intruderList = Joe 123456
                                                           Sandy 654321)
```

Appendix H

Part II Project Proposal

Introduction

The Opera research group in the computer lab has developed an event architecture within an object-oriented distributed programming (OODP) environment based on CORBA. It is called *COBEA*[1] and allows the usage of events for building large scale distributed applications.

This architecture bases on the publish-register-notify paradigm soq that clients that are interested in a specific class of events can register and are notified of any occurrence of these events by servers. At the moment events are only implicitly specified. Clients use a library as a low-level interface to an event server so that they can register and deregister their interests.

However, it would be desirable to use *ODL*[2] specified by the ODMG[3] as an event specification language in order to describe events in a standard way. A mapping from these ODL event specifications to Java and C++ code would allow events to be implemented as objects with a type and certain attributes.

Description

The ODL Stub Generator

The main aim of this project is to develop an *ODL event type compiler* that outputs stub code in C++ or Java. This requires the design of a mapping from event types to ODL. Clients can then use the stub methods generated by the compiler as an interface to the event server, for example, to register/deregister call-back methods for event notification. This would facilitate the process of application and server development and give support for strong-typing. Furthermore, application developers would not need to marshal and unmarshal events themselves. They could work on the

[1] COrba-Based Event Architecture
[2] Object Definition Language
[3] Object Data Management Group

ODL mapping of the events and have a consistent API to use. The stub code would interface with the lower-level library.

The figure H.1 gives a simplified view of the resulting architecture of the event system.

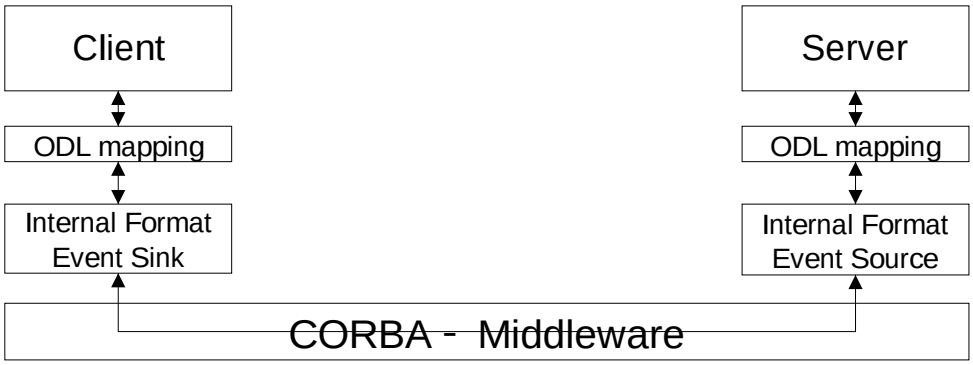

Figure H.1: The Event System Architecture

The event type compiler takes an ODL event type definition as input and generates stub code in Java and C++. It consists of the standard compiler phases as shown in figure H.2. The code generation phase of the event type compiler can be implemented as a plug-in module so that both C++ and Java stub code can be generated.

The compiler will be implemented in C++. The process of implementing the lexer and parser will be simplified by using the Unix tools *lex* and *yacc*. Appropriate data structures for storing the abstract syntax tree generated by the parser have to be designed.

The stub code that has been generated is the interface used by clients and servers to communicate with the event system. It must include facilities for registration and deregistration of event interests of clients and a way to notify about the occurrence of events.

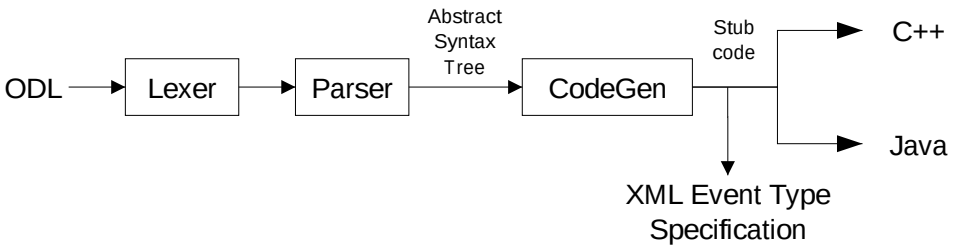

Figure H.2: The ODL Stub Generator

Registration of event superclasses

Moreover, a scheme should be implemented where it is possible for clients to register an event with its *superclass* and then be notified about any subclass events of that superclass. The main advantage of this would be the possibility of applications to register a general event if all possible derivatives of this event are of interest to the application.

Generation of Parsed Event Type Definitions

If the event system is used in connection with an event type repository or with cross-domain applications, *parsed event type definitions* will be required. Such event type definitions can be produced by the stub generator during the code generation phase and output as an *XML*[4] file. A suitable format has to be designed so that the type definitions can be imported into a type repository or used by applications that cross domain boundaries to query event types.

Possible Extensions

This section contains two sensible extensions of the main project idea as described above. The decision which of these extensions will be included into the project will be done at a later project stage as indicated in the project schedule.

Parser for Event Filtering Expressions

Clients that register their interest in particular events can use *event filtering expressions* in order to receive events constrained by certain parameters. For this it would be necessary to write a simple parser for event filtering expressions which would be used during the registration of clients. It parses the event filtering expression so that it can be determined whether an event is of interest and the corresponding application needs to be notified. This would involve the modification of current code.

Interfaces for Event Access Control

A further extension would be the definition of suitable interfaces in the stub code that can be used for an *access control scheme*. This would allow the association of certain access restrictions with particular events so that only certain clients are allowed to register for these events.

Work to be undertaken

- Literature study and initial preparation. (CORBA, ODL, IDL, COBEA)

[4] eXtensible Markup Language

- A suitable mapping of event types to ODL has to be investigated. For example, this involves the specification of an event base class.
- The stub code that is to be produced by the event type compiler out of an ODL event definition has to be defined and hand-coded.
- A simple client and server application has to be built that can be used for testing and evaluation of the whole event system.
- A lexer and parser for the ODL event definitions is required which outputs an abstract syntax tree needed for code generation.
- A code generator is required that transforms the parse tree into C++ or Java stub code.
- A suitable format for the XML event type specifications has to be designed. This involves a mapping from ODL to XML.
- A code generator is required that transforms the parse tree into XML event type definitions.
- Depending on the progress of the project one of the possible extensions can be implemented.

Starting Point

Programming Languages

I have a good knowledge of Java because of the group project done last year.

In addition to that I have knowledge of C++ although I have never coded a larger project in this language.

The usage of the compiler tools *lex* and *yacc* is familiar to me because of the "Compiler Construction" lecture course.

Middleware – CORBA – COBEA

Although I am familiar with the basic concepts of Middleware and CORBA because of the "Concurrent Systems" lecture course, it will be necessary for me to spend a considerable amount of time to understand and learn the detailed usage of CORBA in distributed applications. In particular, I will have to familiarize myself with IDL and ODL and it will be necessary for me to understand the structure, design and implementation of COBEA.

Timetable

The following timetable gives the overall schedule for the project. It is subdivided into working packages, each a *fortnight* long.

11. October – 24. October	**Michaelmas Term** First project idea Contacting supervisor Writing Project Proposal
Milestones:	*Project Proposal with timetable* *Submission of Project Proposal* *Approval from Overseers*
25. October – 7. November	 Reading literature Introduction to CORBA & IDL Introduction to ODL Introduction to COBEA Preparation and Setup of college computer Study of the current Event System
Milestones:	*Production of a model CORBA application*
8. November – 21. November	 Initial version of the *event type→ODL* mapping Initial version of hand-coded stubs Implementation of the demo client-server application
Milestones:	*First draft of* event→ODL *mapping* *First draft of* ODL→C++/Java *mapping*
22. November – 5. December	 Refinement of ODL event definitions Refinement of hand-coded stubs Revision on the usage of lex & yacc Implementation of Lexer & Parser for ODL
Milestones:	*Final version of ODL event definitions* *Final version of hand-coded stub code* *Working Lexer and Parser for ODL*
6. December – 16. January	**Christmas Vacation** (used for revision)
17. January – 23. January	**Lent Term** Implementation of C++/Java code generator Testing and Evaluation of code generation
Milestones:	*Working version of code generator*
24. January – 30. January	 Definition of *ODL→XML* mapping Implementation of XML code generator Writing of Progress Report
Milestones:	*Submission of Progress Report*
4. February	**Progress Report Submission Deadline**

31. January – 13. February	Continuous improvement of code generation Decision about possible extension Implementation of extension
Milestones:	*Decision about extension* *Final version of stub code generator*
14. February – 27. February	Writing Introduction and Preparation Chapters
Milestones:	*First part of Dissertation written*
28. February – 12. March	Writing Implementation and Evaluation Chapters
Milestones:	*Second part of Dissertation written* *First draft of Dissertation completed*
13. March – 23. April	**Easter Vacation** (used for exam revision)
24. April – 7. May	**Easter Term** Submission of Dissertation to supervisor Feedback from supervisor Preparation of diagrams Preparation of appendices Proof-reading of Dissertation
Milestones:	*Final version of Dissertation*
8. May – 19. May	Small changes and additions Final review of Dissertation Submission of Dissertation
Milestones:	*Submission of Dissertation*
19. May	**Dissertation Submission Deadline**

Special Resources

The main development will be done on my own computer in College. The necessary software tools for the development work will be installed on it. It is moreover possible to run a local version of the event system on this machine.

The used computer runs Linux OS and uses a streamer as a backup facility. Moreover, regular backups to the Pelican Archive Service provided by the Computing Service will be done.

At a further stage of the project and particularly during testing I will use Opera's linux machines on which an account and 25 MB of disc space will be provided for this project.

Supervisor

Walt Yao (Walt.Yao@cl.cam.ac.uk), PhD student in the Opera group, will be supervising this project.