A FLEXIBLE FRAMEWORK FOR DEVELOPMENT OF COMPONENT-BASED DISTRIBUTED SYSTEMS

Arnor Solberg,Tor Neple,Jon Oldevik and Bård Kvalheim

SINTEF Telecom and Informatics
Forskningsveien 1 P.O.Box 124 Blindern, 0314 Oslo, NORWAY
Tel: +47 22 06 73 00 fax: +47 22 06 73 50

{Arnor.Solberg | Tor.Neple | Jon.Oldevik | baardk}@informatics.sintef.no

Abstract: This paper describes a generic framework facilitating the specification and construction of component-based distributed systems. The framework integrates methods for specification of systems with tools supporting the construction of those systems. This is achieved by defining a reference architecture supported by a metamodel, a Component Modelling Language and tools for code-generation. The metamodel is an extension of the UML metamodel. The Component Modelling Language is a lexical description language based on CORBA IDL.

Keywords: Business object framework, component modelling language, UML

1 INTRODUCTION

Over the last few years component-based system development has become increasingly popular. The general concepts of building products by integrating a set of parts with well defined interfaces and characteristics, has been known and used for decades in industries like automotive manufacturing and electronics. The latter is probably the best example of how design, configuration and assembly of standard components provide products of higher quality and lower cost to the consumers. Imagine what a 300$ CD player would cost if it contained no standard off the shelf components.

Within software development, components were first used in the programming of user interfaces. Rapid application development tools such as Microsoft Visual Basic,

L. Kutvonen et al. (eds.), *Distributed Applications and Interoperable Systems II*

Borland Delphi and Powerbuilder allowed the developer to build user interfaces by assembling pre-built components such as grids, buttons, menus and so on. Such components are also available for connecting the user interfaces to different databases and performing different queries in a simple manner.

The new wave of client/server systems based on distributed object technologies with thin clients and distributed application and business logic has sparked initiatives and technologies for component-based server programming. Microsoft COM/DCOM and their Transaction Service, Enterprise Java Beans and CORBA Components are examples of standards and technologies that facilitate the development of such systems.

In software development it is important that the developed systems conform to a defined reference architecture. The quality of the reference architecture will directly influence the quality of the system at hand, including vital aspects such as maintainability and flexibility. Conforming to a well-defined architecture will also assure technical interoperability and facilitate semantic interoperability. It is important to emphasise that the focus on architecture has to be set in the design phase of the development process, and that this focus is held throughout the different phases of system development. Only through architecture-driven design and implementation, the goal of conforming to the architecture can be reached.

In order to make architecture-driven development easier, and to achieve results of higher quality in the lowest amount of time, a framework containing architecture descriptions, standard architecture elements and tools, is of great help.

Most frameworks that exist today are either language, platform or product specific. However, it is useful to be able to abstract away from platform and language issues while creating the system description. The main focus and effort should be on solving the business problems at hand, creating the systems the users need. This paper presents a framework that allows this abstraction, letting the user specify the system using models and lexical descriptions. These descriptions are then used as input to tools that create mappings toward different platforms, languages and middle-ware. The framework contains:

- a reference architecture,
- a metamodel which is a UML[1] metamodel extension,
- a component definition language (CML),
- scripts that generate CML definitions from UML models and
- a code generator that creates CORBA IDL[2], ODMG ODL[3] and code skeletons.

Each of the parts listed above are described in some detail in the following sections. The framework architecture is related to the ISO RM-ODP architecture framework[4-7], the relationships and mappings are described in a separate section of this paper. The last part of the paper contains an example that describes how the framework is used in system development.

2 REFERENCE ARCHITECTURE

The reference architecture is partly developed within the ESPRIT IV project OBOE[8]. The OBOE project focuses on specifying and building a generic and open infrastructure running distributed business objects. It has also been used to build an information system for monitoring, reporting and planning the marine seismic acquisition process. This system is used world wide within Geco Schlumberger, a partner in the OBOE project. Figure 1 depicts the generic service-oriented architecture. It is based on a common three tiered architecture. Within each of the three layers additional layers are added, with the intent to increase flexibility and robustness.

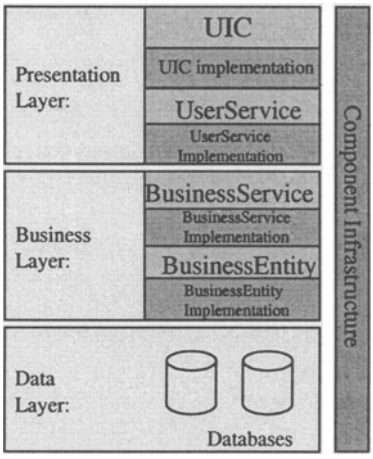

Figure 1. Reference architecture.

The reference architecture separates implementation as distinct parts to explicitly achieve technology transparency. The implementation parts of the architecture apply to the technology viewpoint of the RM ODP framework.

2.1 The tiers of the reference architecture

In the following we describe each of the tiers of the reference architecture shown in Figure 1.

The User Interface Component layer (UIC) includes the user interface components and the user interface control logic. It includes user presentation and handles user interface events such as mouse clicks, keyboard input, etc.

The UIC is typically implemented in a programming language such as Java, Visual Basic or C++. The UIC implementation is basically the realisation of the user interface and the control logic handling user interactions It utilises the services provided by the UserService layer to accomplish its tasks.

The UserService layer provides services needed for a particular application. These services are packaged in one or more UserServices. A UserService is an interface,

which manifests a set of services as operation signatures. The main purpose of the UserService layer is to be a facade that shields the UIC from being aware of any other part of the system. The services are typically derived from the use-cases describing the requirements for the particular application. The UserService layer includes all the services that the UIC needs for satisfying the application users in respect of the specified functional requirements. The services defined in the UserService layer are not distributed. For services to be available on the net they must be defined in the BusinessService layer.

A UserService implementation will typically relate to a chosen component infrastructure technology. By having several implementations of the UserService component, one gets a system that run on different technologies. For instance one UserService might bind to a CORBA ORB, one might bind to Java RMI and one might even be local and for instance wrap local files. The UIC is unaware of this remains untouched in all these cases.

The BusinessService layer also describes a set of services. These services are packaged in what we have called BusinessServices. While the UserService layer describes services needed for a particular application, the BusinessService layer describes common services applicable for several applications. Typically a BusinessService also use services offered by other BusinessServices. The services described in the BusinessService layer are distributed.

The BusinessService implementation is the actual implementation of the services described in the BusinessService layer. The implementation will utilise the chosen component infrastructure to distribute the BusinessService components. To accomplish the offered services, the BusinessService implementation uses the BusinessEntities described in the BusinessEntity layer. Delegation to other available BusinessService components is also typically utilised. A common service operation within a BusinessService is to collect a set of BusinessEntity components and send these components to the requesting client.

The BusinessEntity layer describes the information model for the system. The information model contains the Business Entity components and describes their attributes, operations and relationships. Business Entity components represent things such as customer, vessel, report, car, etc. It is typically the Business Entity components that "travel around" according to client requests.

The BusinessEntity implementation is the actual implementation of the Business Entity components and their relationships. The implementation will utilise the chosen component infrastructure to distribute the Business Entity components. The implementation also handles the wrapping to the actual data storage. There are several possible wrapping techniques, for instance:

- using direct binding to the actual data storage,
- using a de facto database protocol such as ODBC, JDBC or Java Blend or
- using relevant services offered by component technologies, e.g. Persistent State Service offered by CORBA.

The Data Layer describes the mapping to the actual storage.
The Component infrastructure is the infrastructure required to support components in a distributed component-environment. This might for instance be CORBA,

DCOM or Enterprise Java Beans. The component infrastructure will handle component distribution and typically offers services that support some level of technical and semantic interoperability.

3 THE METAMODEL

The UML standard offers the possibility of making extensions to the UML metamodel. At the model level these extensions appear as stereotypes. By using this mechanism, desired concepts may be integrated into the UML models. Commercially available UML tools like Rational Rose that enables integration through API's or scripting, make it possible to perform model checking and specialised code generation to support specific needs. Thus, UML metamodel extensions facilitate a flexible way of supporting context, domain or architecture-driven concepts at the model level.

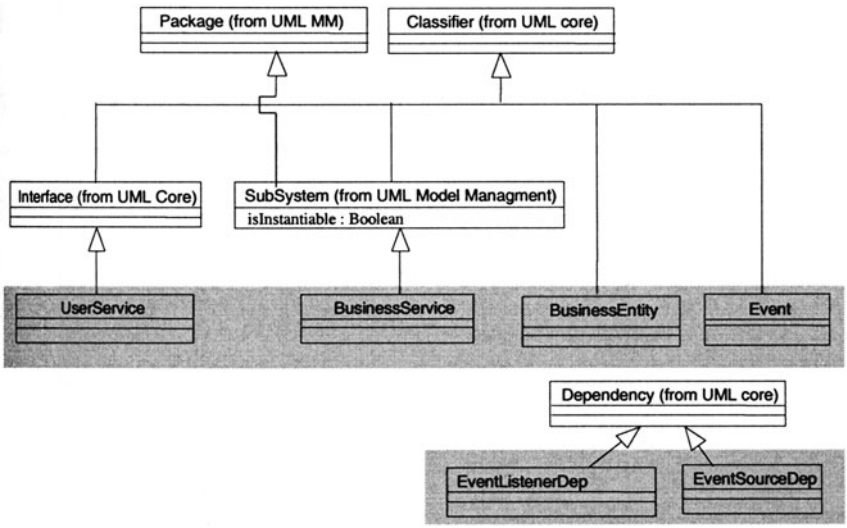

Figure 2. The metamodel.

The metamodel defined in Figure 2 extends the UML metamodel with concepts that corresponds to the reference architecture. This includes the UserService, the BusinessService and the BusinessEntity concepts. In addition the metamodel supports an event model for handling business events. This includes the Event, the EventListenerDep and the EventSourceDep concepts. The new concepts are marked with grey background in Figure 2.

3.1 The metamodel concepts

The UserService is a subclass of Interface from the UML core package. A UserService is only a collection of operations with a name.

The BusinessService inherits SubSystem from UML Model Management. A Business Service logically contains a set of BusinessEntities, and serves as the controller of the BusinessEntity interactions. A BusinessService component is instantiable and access transparent. Access transparent means that the BusinessService components are registered and available on the net, so a BusinessService component will be a CORBA object in a CORBA implementation. A restriction defined for the BusinessService components is that they only may have relationships to BusinessEntities or to other BusinessServices. This to ensure the described independence between the layers defined in the reference architecture. The following OCL[9] statement defines this formally:

```
self.allOppositeAssociationEnds -> forAll (a | a.type.oclIsTypeOf
     (BusinessService) or a.type.oclIsTypeOf (BusinessEntity))
```

The BusinessEntity inherits Classifier from the UML core package. A BusinessEntity component is persistent and is access-transparent. A restriction defined for the BusinessEntity components is that they may only have relationships to other BusinessEntities. This again to ensure that the BusinessEntity components are independent of the rest of the system. This is defined in OCL as follows:

```
self.allOppositeAssociationEnds -> forAll(a |
     a.type.oclIsTypeOf(Entity))
```

The metamodel supports an event model that includes the concepts *Event*, *EventListenerDep* and *EventSourceDep*. The event model is based on the JavaBeans event model. However, the event model has only an event hierarchy, not an event interface hierarchy as in JavaBeans.

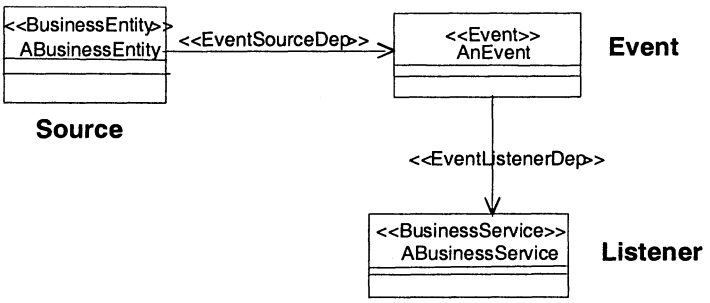

Figure 3. Event Model example.

An event is represented as an object. An event can only have attributes. The only methods in an event object is the constructor and get() operations for the attributes. The attributes are read-only. An event is dependent of an event-source to be instantiated, and event listeners register themselves to be notified of events. BusinessServices and BusinessEntities may be event sources and event listeners, while UserService may register to be an event listener. The event model is illustrated with the example in Figure 3.

The figure shows a BusinessEntity that generate events of type AnEvent and a BusinessService that has registered to be a listener of that event.

4 COMPONENT MODELLING LANGUAGE (CML)

A part of the framework is the Component Modelling Language, which is a lexical description language for describing systems in a technology independent manner. CML is a superset of CORBA 2.0 IDL. The new concepts in CML are: Relationships, Events and the concepts UserService, BusinessService and BusinessEntity.

Since IDL is only an interface description language, it does not have the possibility to describe anything else than the interfaces used in a system. CML, which includes these new concepts, has more descriptive power than IDL. Below we will describe the IDL extensions in CML.

Relationships in CML corresponds semantically and syntactically to the relationship concept used in ODMG's ODL. In CML there are three different appearances of relationships: BusinessEntity – BusinessEntity, BusinessService – BusinessEntity, BusinessService – BusinessService. The semantics of all three appearances are the same. From the system programmer's point of view, it should be perfectly transparent whether it is an entity – entity or service – entity relationship. In addition to the three appearances, there are four relationship types as defined in the ODMG standard: list, bag, set and dictionary.

The syntax to describe a relationship in CML is defined in the following way (BNF grammar):

```
<relationship_dcl>     ::= "Relationship" <relationship_type>
                           <identifier> ["inverse" <scoped_name>]

<relationship_type>    ::= <param_type_spec>
                           | "list" "<" <param_type_spec> ">"
                           | "bag" "<" <param_type_spec> ">"
                           | "set" "<" <param_type_spec> ">"
                           | "dictionary" "<" <param_type_spec> ","
                           <param_type_spec> ">"
```

The identifier, scoped_name and param_type_spec are derived from the CORBA 2.0 IDL specification (later productions that end with *_from_corbaidl* are also from CORBA 2.0 IDL). This part of the CML grammar also illustrates the four different types of relationship that exist in CML. The semantics of the inverse relationship are identical to the inverse relationship in the ODMG standard.

The BusinessServices are full-blown distributed components that the client typically binds to. This means that BusinessServices may include methods, attributes and relationships. A BusinessService might also be an event source and an event listener. The CML syntax is defined in the following way:

```
<businessService_dcl>     ::= "BusinessService" <identifier>
                              [<inheritance_spec_from_corbaidl>]
                              "{" <businessService_body_dcl> "}"

<businessService_
body_dcl>                 ::= <export_businessService>*
```

```
<export_businessService> ::=  <type_dcl_from_corbaidl> ";"
                            | <const_dcl_from_corbaidl> ";"
                            | <except_dcl_from_corbaidl> ";"
                            | <attr_dcl_from_corbaidl> ";"
                            | <op_dcl_from_corbaidl> ";"
                            | <signal_dcl> ";"
                            | <subscribe_dcl> ";"
                            | <relationship_dcl> ";"
```

The Business Entities are the "data objects" within a system. The syntax in CML is defined in the following way:

```
<businessEntity_dcl>      ::=  "BusinessEntity" <identifier>
                               [<inheritance_spec_from_corbaidl>]
                               "{" <businessEntitybody_dcl> "}"

<businessEntitybody_dcl> ::= <export_businessEntity>*

<export_businessEntity>  ::= <export_businessService>
```

The UserService is an interface defining different services for an application. The implementation of a UserService marshals the request from the client application typically to a BusinessService. The syntax in CML is as follows:

```
<userService_dcl>      ::= "UserService" <identifier>
                           [<inheritance_spec_from_corbaidl>]
                           "{" <userservicebody_dcl> "}"
<userServicebody_dcl>  ::= <export_userService>*
<export_userService>   ::= <subscribe_dcl> ";"
                           |<op_dcl_from_corbaidl> ";"
```

The Event model in CML is based on Java 1.1 event model. In our framework BusinessServices and BusinessEntities can both subscribe to and generate events, whilst UserServices only may subscribe to events. The BNF syntax for event, event production and event subscription is:

```
<event_dcl>         ::= "event" <identifier> "{" <eventbody_dcl> "}"

<eventbody_dcl>     ::= <eventexport>*

<eventexport>       ::= <event_attr_dcl> ";"

<event_attr_dcl>    ::= <param_type_spec> <declarators_from_corbaidl>
<signal_dcl>        ::= "Signal" <scoped_event_name>

<subscribe_dcl>     ::= "Subscribe" <scoped_event_name>
```

5 PARSING AND CODE GENERATION TOOLS

The framework includes parser and code generation tools, both for parsing UML models, generating CML and parsing CML. Currently the CML parser generate IDL, Java and ODMG's ODL.

This means that code is generated for the chosen component infrastructure binding based on a UML model. Java skeletons and the persistence binding to an ODL based database are also generated.
The UML parser is built using the scripting language in Rational Rose, which parses the active model and generates CML.

The CML parser is built using JavaCC (Java Compiler Compiler). The parser has been built using the IDL grammar as well as the grammar for the new concepts of CML described in section 4.

5.1 Mapping

The CML concepts are mapped to IDL using the following mapping strategies:

- The UserService, BusinessService and BusinessEntity concepts are mapped to interfaces in IDL.
- Relationships are mapped to IDL using iterators.
- Any CML interface that signal events is transformed into the equivalent IDL interface derived from the Notification service structured event supplier interface.
- Any CML interface that subscribes to events is mapped to the equivalent IDL interface derived from the Notification service structured event consumer interface.
- Subscribe and unsubscribe methods for handling event subscription appears within in the supplier's interface in the IDL file to make these methods accessible on the ORB.

The added CML concepts are mapped to ODL in the following way:

- The UserService, BusinessService and BusinessEntity concepts are mapped to interfaces in ODL.
- Relationships in CML are mapped to ODL relationships (their semantics are the same).

The added CML concepts are mapped to Java in the following way:

- The UserService, BusinessService and BusinessEntity concepts are mapped to Java classes.
- Relationships in CML are mapped to Java using hash tables and vectors.
- The CML event model is mapped to Java by creating a Java class for the event, a Java interface for handling the event and event supplier and consumer classes that derive and use the CORBA notification service.

6 THE RELATIONSHIP WITH RM-ODP

ISO RM-ODP[4-7] defines a set of frameworks within which support for distribution, interworking, interoperability and portability can be integrated. ODP standardisation considers distributed systems spanning many organisations and

technological boundaries. This section will study how the architectural framework of ODP can be related to the reference architecture described in section 2 and 3.

In general, an ODP system can be described in terms of related, interacting objects. The ODP foundation is defined by a set of basic modelling concepts, specification concepts and structuring concepts, being the building blocks upon which the viewpoints, the viewpoint languages, the conformance framework and the distribution framework is based. RM ODP defines the architectural framework for structuring ODP systems in terms of viewpoint specifications and distribution transparencies. An ODP system is specified in terms of a set of related but separated *viewpoints*. Five viewpoints are defined in ODP: enterprise, information, computational, engineering and technology, each associated with a viewpoint language that defines a set of concepts for each viewpoint.

The enterprise viewpoint is concerned with the purpose, scope and policies of the enterprise related to the specified system or service. It covers the role of the system in the business, and the human user roles and business policies related to the service.

The information viewpoint is concerned with the semantics of information and information processing. It covers the information held by a system and the information processing carried out by the system.

The computational viewpoint is concerned with the interaction patterns between the components (services) of the system, described through their interfaces. It covers the service interfaces as seen from a client and the interactions between components.

The engineering viewpoint is concerned with the design of distribution-oriented aspects, i.e. the infrastructure required to support distribution and provide distribution transparencies. The main concern is the support of interactions between computational objects. The following transparencies are defined by ODP: access, location, persistence, transaction, failure, migration, replication and relocation

The technology viewpoint is concerned with the provision of an underlying infrastructure. A technology specification defines how a system is structured in terms of hardware and software components, and underlying supporting infrastructure.

ODP provides a reference model for distributed systems, and it is timely to identify correlation points between this and the logical architecture described in section 2 and 3. This will help assuring that standard models and ways of thinking are preserved. Figure 4 gives a high-level perspective on this relationship.

The enterprise viewpoint drives requirements to all levels of the architecture. The information viewpoint is represented by the business entities (the persistent objects) in the architecture. The computational viewpoint is represented by the user business services. The engineering viewpoint is represented by requirement-statements for distribution transparencies that are described jointly with the architecture. The technology viewpoint is represented by implementations, explicit infrastructure mappings and other technology choices.

From an architectural perspective, it is interesting to analyse how the multi-tier architecture (and the framework) can accommodate engineering requirements (transparency requirements) and map these to underlying services supported by the component infrastructure. We will se that many distribution transparencies can be supported directly by such mappings.

Access transparency can be directly provided by the component infrastructure, e.g. by CORBA, Java or DCOM.

Location transparency can be partly provided by infrastructure services like naming or trader services, e.g. CORBA Naming Service or Java Naming and Directory Interface (JNDI).

Persistence transparency can be provided by support from persistence services or automatic generation of database language mappings. In the reference architecture, all BusinessEntities are assumed to be persistent. Automatic mappings to ODL databases are performed.

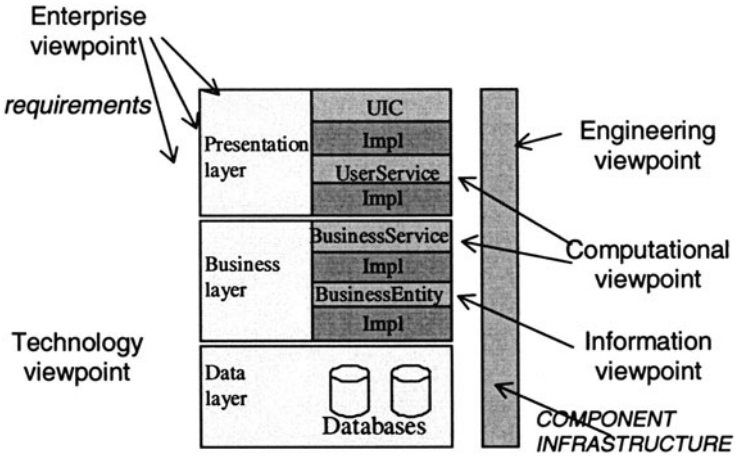

Figure 4. Relationship between the logical architecture and RM-ODP.

Transaction transparency can be provided by support from transaction services, e.g. CORBA, Microsoft or Java Transactions Service. Also, transactional support in underlying databases may be used. During analysis, transaction requirements can be identified in modelling (e.g. as tagged values on services/entities). This can be used to create automatic mappings to the chosen transaction technology.

Similar strategies can be applied for supporting additional transparencies and services in the framework, adding value to the design process as well as the flexibility of the architecture.

7 USING THE FRAMEWORK

This section demonstrates how to specify and construct a component-based, distributed system based on the framework. The system to construct is a simple car rental system, handling car reservations. The system also handles overdue events if a car is not checked in or not checked out according to the dates specified in the car reservation. A car reservation comprises a customer, a car and a period of time. The UserService and the BusinessServices defined in the system offers the services needed for handling reservations of cars. This includes making reservations and checking in and out cars.

7.1 Model

The simple car rental system includes one UserService: *CarRental*, two BusinessServices: *RentalService* and *CustomerService*, three BusinessEntities: *Reservation*, *Customer* and *Car*, and two events: *NotCheckedIn* and *NotCheckedOut*. These components, their attributes, relationships and operations are modelled in UML using Rational Rose. The UML model is shown in Figure 5.

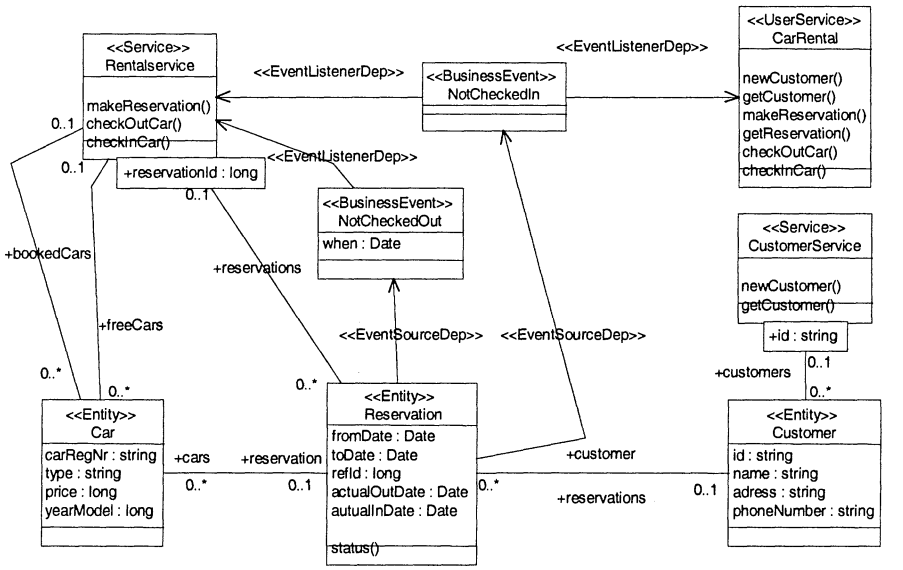

Figure 5. The UML model of the car rental example.

Note that the concepts defined in the framework appear as stereotypes in the model.

7.2 CML code

The UML parser is now used to generate CML. Parts of the generated CML code is listed below.

```
// CML mapping generated from Rational Rose BOF MetaModel
// File : 'I:\PROJECTS\OBOE\Metamodel\CarRental.cml'
// Date : '03-oct-98'
businessEntity Reservation {
 relationship list <Car> cars;
 relationship Customer customer inverse Customer::reservations;
 attribute Date fromDate;
   // The remaining attribute declarations are left out
 signal NotCheckedIn;
 signal NotCheckedOut;
 string status(); };

businesService Rentalservice {
 subscribe NotCheckedIn;
 subscribe NotCheckedOut;
```

```
 relationship list <Car> freeCars;
  // The remaining relationship declarations are left out
   long makeReservation(in CustomerService::Customer theCustomer, in Car
   theCar, in Date from, in Date to);
  // The remaining method declarations are left out };
event NotCheckedOut {
 Date when; };
// The remaining CML code is left out
```

7.3 IDL, Java and ODL

Using tools included in the framework, IDL, ODL and Java class skeletons will be generated based on the CML file. The IDL then is compiled using a Java IDL compiler, generating stubs and skeletons. The ODL file is the database schema and is used to generate the implementation of the data layer for the system. The appropriate Java class skeletons are also generated. These skeletons also include event handling.

Parts of the generated IDL is listed below.

```
/* IDL file generated from CMLParser - Tue Jan 26 13:21:01 CET 1999*/
module CarRental{ // forward declarations of the interfaces:
 interface Reservation;
 interface RentalService;
 // The remaining interface declarations are left out

 interface ReservationDictionaryIterator{
  Reservation nextReservation();
  boolean hasMoreReservations();};
 interface CarListIterator {
  Car nextCar();
  boolean hasMoreCars(); };
 // The remaining iterator declarations are left out

 // Event interfaces (the NotChecedIn event declaration are left out)
 interface NotCheckedOutConsumer:CosNotifyComm::StructuredPushConsumer
 { };
 interface NotCheckedOutSupplier:CosNotifyComm::StructuredPushSupplier{
  boolean addNotCheckedOutConsumer(in NotCheckedOutConsumer consumer);
  boolean removeNotCheckedOutConsumer(in NotCheckedOutConsumer
  consumer); };

 interface Reservation : NotCheckedInSupplier, NotCheckedOutSupplier{
  CarListIterator getCars();
  Customer getCustomer();
  long getFromDate(); void setFromDate(in long fromDate);
  // The remaining declarations are left out };

 interface RentalService {
  attribute NotCheckedOutConsumer notCheckedOutConsumer;
  ReservationDictionaryIterator getReservations();
  long makeReservation(in Customer theCustomer, in Car theCar,
  in long from, in long to);
  void checkOutCar(in long reservationId);
  void checkInCar(in long reservationId);
  // the remaining declarations. are left out};
// The remaining IDL are left out
};
```

8 SUMMARY

This paper has presented a framework for architecture-driven development of component-based distributed systems. The framework has been used with success in the OBOE project mentioned in the text. As illustrated, the innovations presented here facilitate easier and more flexible architecture-driven development, aiming at supporting the central concepts of business object and component architectures.

Based on the experiences from this project, and internal usage of the framework, we intend to develop the ideas and tools further. Among the issues we are working on is incorporating support for transaction management, and mappings toward other infrastructures. The work on using the framework toward a Microsoft DCOM environment with Microsoft Transaction Server will start shortly. Other target environments such as Enterprise Java Beans and CORBA components will also be investigated. Further development will also be done to enable the automatic management and mapping of changes to implementation and models at all levels.

Currently the tool that converts from UML to CML is specific for Rational Rose98. In the next iteration we will make a tool that generates CML from a XMI representation of the UML model. This work will start when the OMG XMI (XML Model Interchange) standard has stabilised.

References

[1] UML CONSORTIUM, UML Semantics, *Rational Software Corporation Version 1.1*, 1 September 1997.

[2] OMG, *Object Management Architecture Guide*, Third ed: John Wiley & Sons, Inc., 1995.

[3] CATTELL R., BARRY D., BARTELS D., BERLER M., EASTMAN J., GAMERMAN S., JORDAN D., SPRINGER A., STRICKLAND H., AND WADE D., The Object Database Standard: ODMG 2.0, pp. 288 *The Morgan Kaufmann Series in Data Management Systems*, J. Gray, Ed. San Francisco: Morgan Kaufmann Publishers, 1997.

[4] ISO/IEC JTC1/SC21, *Basic reference model of open distributed processing, part 1: Overview,* ITU-T X.901 - ISO/IEC 10746-1, August 1995.

[5] ISO/IEC JTC1/SC21, *Basic reference model of open distributed processing - part 2: Foundations,* ITU-T X.902 - ISO/IEC 10746-2, August 1994.

[6] ISO/IEC JTC1/SC21, *Basic reference model of open distributed processing, part 3: Architecture,* ITU-T X.903 - ISO/IEC 10746-3, 1995.

[7] ISO/IEC JTC1/SC21, *Basic reference model of open distributed processing, part 4: Architectural Semantics,* ITU-T X.904 - ISO/IEC 10746-4, 1995.

[8] OBOE, *OBOE whitepaper,* ESPRIT project no 23.233 revision 0.7, 1999.

[9] UML CONSORTIUM, Object Constraint Language Specification, *Object Management Group Version 1.1*, 1 September 1997.