CDM TECHNICAL REPORT: CDM-19-05

The TIRAC™ Development Toolkit: Technical Description

(Software Version 1.00)

Russell Leighton
Lakshmi Vempati
Alan Davis
Mark Porczak
Jens Pohl

15th April 2005

CDM Technologies, Inc.
2975 McMillan Avenue, Suite 272
San Luis Obispo, California 93401

Abstract

This report provides a technical description of the Toolkit for Information Representation and Agent Collaboration (TIRAC™) software framework for the development of intelligent decision-support applications. An overview of the transformational forces that have precipitated the need for a development toolkit capable of supporting a distributed, information-centric software environment, and the objectives of TIRAC are contained in a companion CDM Technical Report (CDM-17-04) entitled: “The TIRAC™ Development Toolkit: Purpose and Overview.”

TIRAC is an application development framework and toolkit for distributed decision-support systems incorporating software agents that collaborate with each other and human users to monitor changes (i.e., events) in the state of problem situations, generate, and evaluate alternative plans, and alert human users to immediate and developing resource shortages, failures, threats, and similar adverse conditions. A core component of any TIRAC-based application is a virtual representation of the real world problem (i.e., decision-making) domain. This virtual representation takes the form of an internal information model, commonly referred to as an ontology. By providing context (i.e., data plus relationships) the ontology is able to support the automated reasoning capabilities of rule-based software agents.

Principal objectives that are realized to varying degrees by the TIRAC Toolkit include: support of an ontology-based, distributed, information-centric system environment that limits internal communications to changes in information; ability to automatically “push” changes in information to clients, based on individual subscription profiles that are changeable during execution; ability of clients to generate information queries in addition to their standing subscription-based requests; automatic management of object relationships (i.e., associations) during the creation, deletion, and editing of objects; and, the ability to interface with external data sources through translators and ontological facades.

Most importantly, the TIRAC Toolkit is designed to support the machine generation of significant portions of both the server and client side code of an application. This is largely accomplished with tools that automatically build an application engine by integrating Toolkit components with the ontological properties derived from the internal information model. In this respect, a TIRAC-based application consists of loosely coupled, generic services (e.g., subscription, query, persistence, agent engine), which in combination with the internal domain-specific information model are capable of satisfying the functional requirements of the application field.

A TIRAC-based software development process offers at least four distinct advantages over current data-centric software development practices. First, it provides a convenient structured transition to information-centric software applications and systems in which computer-based agents with reasoning capabilities assist human users to accelerate the tempo and increase the accuracy of decision-making activities. Second, TIRAC allows software developers to automatically generate a significant portion of the code, leaving essentially only the domain-specific user-interface functions and individual agents to be designed and coded manually. Third, TIRAC disciplines the software development process by shifting the focus from implementation to design, and by structuring the process into clearly defined stages. Each of these stages produces a set of verifiable artifacts, including a well defined and comprehensive documentation trail. Finally, TIRAC provides a development platform for achieving interoperability by formalizing a common language and compatible representation across multiple applications within a distributed environment.

The TIRAC™ Development Toolkit: Technical Description

(Software Version 1.00)

Table of Contents

Chapter 1 - Overview

1.1 Introduction

This chapter provides an overview of the TIRAC architecture, design, and implementation. In additional chapters, development toolkit usage, component and service descriptions, and release documentation are provided. The intended audience for this document are software developers and engineers planning to design and develop systems based on the TIRAC framework and architecture. For a more comprehensive executive overview refer to reference [16].

The TIRAC toolkit can be best described as a "meta-framework" in that it contains the tools to facilitate the implementation of a client-server framework for distributed application development. The generated framework provides a common set of information management services for distributed applications. The architecture of this generated framework may be viewed in terms of physical layers and logical tiers. Figure 1.1 illustrates the role of each physical layer with respect to the logical tiers. It is through the exposed physical layers that client applications interact (the presentation tier), with the middle (unexposed) layers providing information management and distribution (the information tier) and the agents and inference engine providing decision support (the logic tier). A variety of applications ranging from relatively low-level services to graphical user interface applications may be built, accessing a common information management system through the low-level object access layer all the way up to a general user interface layer.

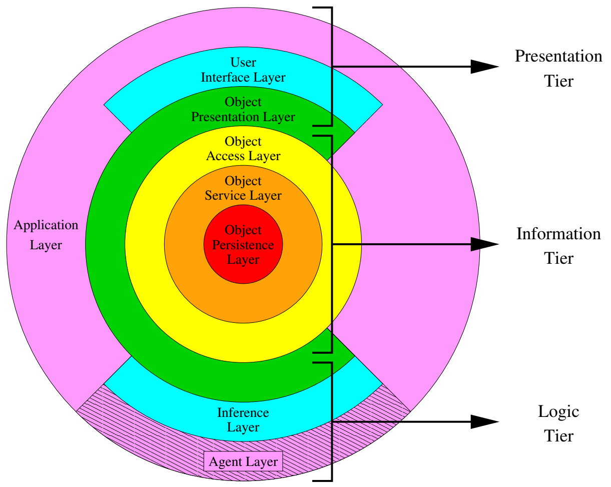


Figure 1.1: Multi-Layered Architecture.

The process of developing specialized frameworks for distributed collaborative decision-support systems can be a significant undertaking. However, through use of the "meta-tools" provided by the TIRAC toolkit the effort required for this process is significantly reduced. The components that comprise the core of TIRAC are shown in figure 1.2. This figure shows the component dependencies as well as their functional grouping. The components shown inside the functional groups are provided (or generated) as part of the TIRAC framework. Those components shown outside of any functional groups are the application specific components. The group highlighted as "Generated Components" comprises those components that are created from the model processing tools. Note that it is the object model that drives the process and is therefore the principle component for defining the structure of a specialized information management system.

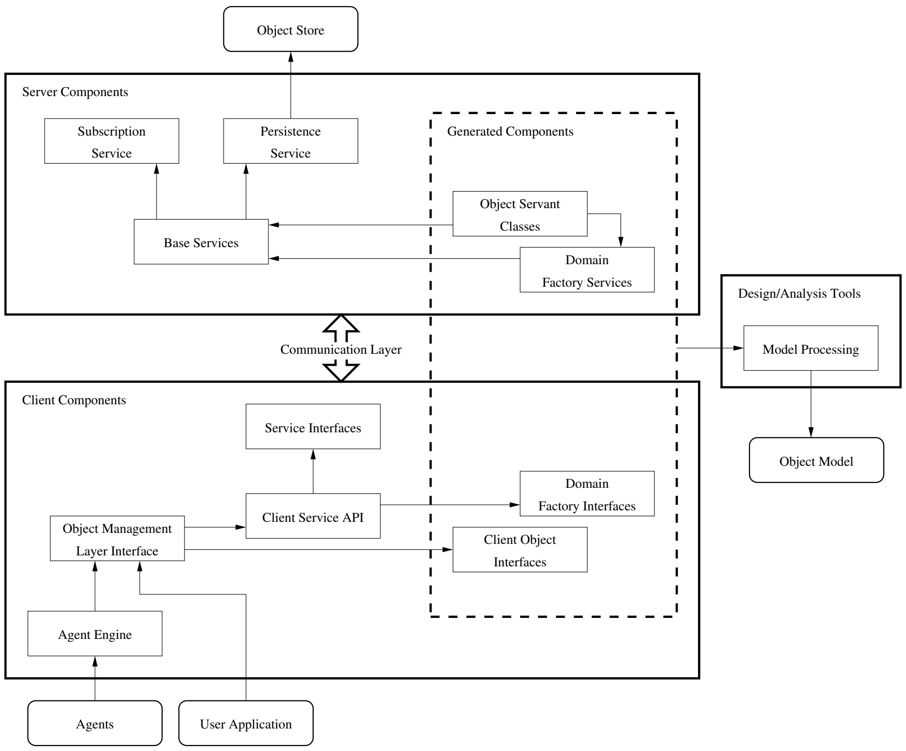


Figure 1.2: TIRAC Component Diagram.

1.2 Requirements

- Architecture
  - Provide capabilities for developing distributed collaborative decision-support systems.
  - Provide tools that support use of information and expertise in a form that conveys high-level meaning and understanding.
  - Provide reusable services common to distributed collaborative decision-support systems.
- Interoperability
  - Provide seamless access to information and services across multiple domains.
  - Provide support for mapping information and data sources to common information domains.
- Support
  - Provide tools to support development of shared components.
  - Provide framework and process to support testing and verification of components and systems.
  - Provide tools for installing and managing components.

1.3 Design and Implementation

The basis for the TIRAC meta-framework is founded on the representation of inherently complex information, providing an object-oriented service architecture for building decision-support systems. Otherwise stated the architecture of the TIRAC meta-framework is based on the recognition that information is naturally complex and that through a rich representation, that captures the real-world complexities of information, the design and implementation of decision-support tools (in the form of software agents or human user interactions through visual interfaces) are considerably simplified.

1.3.1 Object-based Information Representation

Development of systems tailored for specific decision-support applications begin with development of the information model. The TIRAC model processing tools are used to produce the domain specific framework, providing interfaces that correspond to the structure and logic captured by the information model. Generated object interfaces expose methods (accessors and mutators) for interaction with information model features, such as attributes and associations, as defined by the model. Client applications interacting with objects through these interfaces have no specific knowledge of the details required to manage object life-cycles (creation, state maintenance, etc.) and instead focus on information presentation and analysis. Most of the components that comprise a system, built on this architecture, interact at some level as clients to the underlying information management services. For example, both software based agents (decision-support logic modules) as well as user interfaces (providing information presentation to human users) interact through exactly the same object-based interface.

The logic required to manage object interaction resides in servants that are coupled with the client object interfaces through an object request broker (e.g., CORBA ORB). These object servants may be physically separate from their corresponding interfaces (e.g., they may be resident on a host platform separate from the client platform connected through a network) providing location transparency through the underlying information management services, such as the object factories.

1.3.2 Information Services

As with object-based information interaction, basic services, either generated or provided as part of the TIRAC framework, provide interfaces allowing client-level access. The services present an interface exactly analogous to information objects. Services have client interfaces with the service implementation resident in a possibly remote servant that, as with objects, provides for service location transparency. Services may be discovered through the use of a standard lookup service (e.g., name service) enabling the binding of the service to its interface. Services may also interact with other services through their respective client interfaces; therefore, services may also be viewed as clients. It is this mechanism (i.e., interaction through client interfaces) that allows services to be distributed across physical boundaries (e.g., separate processes potentially executing on multiple hosts). In fact, object servants are clients to the basic services, which further blurs the distinction between information objects and services. The base information services and their dependencies (cross-service interactions) are shown in figure 1.3.

Name Service

The name service provides service registration and lookup facilities. It is through the name service that client applications or services establish interfaces to other registered services. All clients (including services) requiring access to services depend on the name service. To reduce diagram complexity, figure 1.3 does not include the name service.

Persistence Service

This service provides access to the persistent store for object state maintenance. It includes support for constrained queries on persisted information.

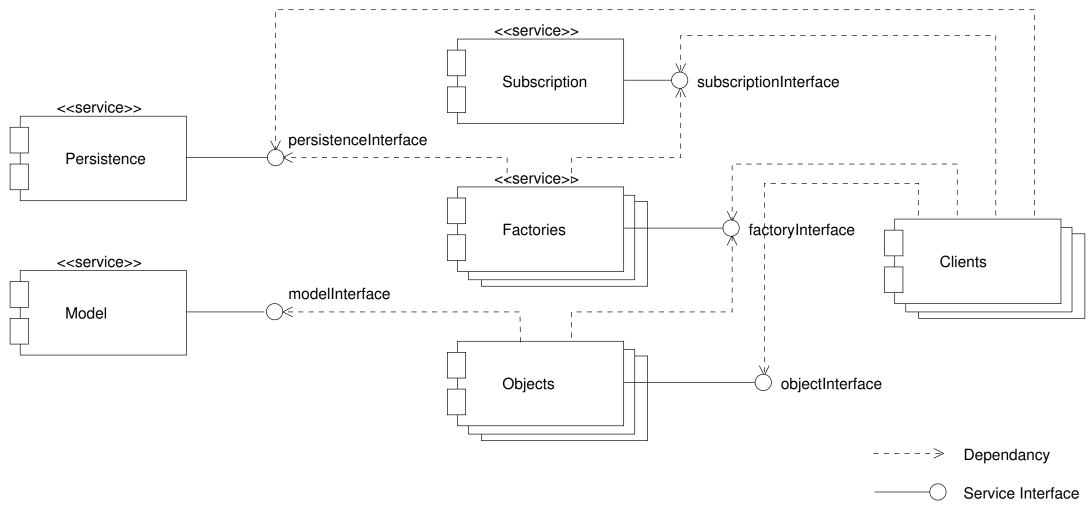


Figure 1.3: TIRAC Services.

Factory Services

The base factory service provides functionality to support interactions with subordinate (domain specific) factory services. The factory service includes support for object life-cycle maintenance and object instance resolution. This service utilizes the persistence service, as a client, to provide object state maintenance. It also utilizes the subscription service to provide notification on object interest satisfaction.

Subscription Service

This service implements an interest and notification management capability enabling clients to register (i.e., subscribe) to interests based on detailed, and possibly very complex criteria. Satisfaction of interest criteria results in notification to the registered client. This capability provides an efficient mechanism for information delivery to clients - providing information based on an asynchronous push rather than through synchronous pull actions (requiring client initiation). This is a particularly important feature with respect to initiation of agent-based information analysis.

Model Service

The model service provides support for object model structure discovery. This service provides object model information (meta-data), which may be utilized to constrain object instance interaction, such as in reference constraints imposed through association management.

1.3.3 Object Management Library

The object management library (OML) is a client application programmer's interface (API) that provides a presentation layer over the client object and service interfaces. This library provides an abstraction layer utilizing run-time class reflection and information properties allowing application development, decoupled from domain model specific interactions. Model meta-data (structure) is utilized within the library methods to constrain object access based on feature (e.g., attribute and association) type. Information is accessed through string values, facilitating user interface development, with specific data type conversion handled automatically based on the feature type information. Object-based interest registration and notification handling are provided through a common listener-based event model.

1.3.4 Agent-based Information Analysis

Agents can be viewed as logical software units that encapsulate patterns and associated actions implementing behavior that supports human decision making. The rules (patterns and actions) that make up agents can be viewed as an integral part of the overall model representing the complete knowledge domain (information structure and business logic). Whereas simple logic may be directly incorporated into the information objects themselves, agents afford the ability to define complex domain logic (e.g., behavior and reasoning that spans object boundaries). Agents may be implemented utilizing a rule-based inference engine (e.g., Jess, CLIPS, etc.), which provides an information analysis environment based on pattern recognition and related action initiation. The TIRAC toolkit provides this capability in the form of an agent engine which serves as a client to a TIRAC based information management system. This agent engine provides the bridge between the object based information structure provided by the information management system and the agent knowledge base required by agent logic modules developed for use within the inferencing environment. The agent engine may also be viewed as a logical collection of agents that comprise logic specific to an application knowledge domain. This grouping of domain knowledge and agents is referred to as an agent session. Agent sessions may also be distributed (they are clients after all) providing common domain business logic, possibly affecting information state, and providing decision support to other clients interacting with the information system. It is helpful to view agents in this context, as collaborators in the analysis of the information state provided by the system, just as human users collaborate through their client interfaces with the system. An integral part of the bridge function that the agent engine provides is the initiation of agent interests based on agent rule patterns. On initialization the agent engine determines agent interests and registers subscriptions (through the subscription service interface). Subsequent interest satisfaction results in notification to the agent engine, which in turn updates the agent knowledge base. This update may in turn cause agent rule activation resulting in possible agent action.

1.3.5 Model Processing Tools

The model processing tools are used to generate a collection of artifacts required, in combination with the base services, for building a system tailored to a specific knowledge domain. Given an object model in the form of XMI (XML Metadata Interchange) compliant with the UML (Unified Modeling Language) meta-model the following artifacts are generated:

- CORBA interfaces defined in IDL (Interface Definition Language)
- Domain factory service for each model package (namespace)
- Object servants
- Client wrappers (classes wrapping calls to CORBA client interfaces)
- Properties for object management information
- Model documentation

1.4 Development Environment

The TIRAC software development environment is provided as a collection of tools. There is no “all-in-one” integrated development environment provided. In fact, where ever possible, off-the-shelf tools and components are utilized in the development and system execution environment. It is through the use of open standards (such as CORBA, XMI, UML, JDBC, etc.) that gives the user of the TIRAC development environment choices in the tools and components fitting their unique needs. The following chapters discuss specific usage of the various tools that make up the TIRAC toolkit, but refrain from specific discussions on configuration and use of off-the-shelf applications (e.g., object modeling tools, databases, etc.) deferring instead to the documentation pertaining to the appropriate tool or component.

Because of the complexities involved in the development process of a complete system based on the TIRAC framework, the next chapter discusses usage in terms of specific examples with the subsequent chapters focused on tool and service details. The intended audience of this document are software developers and engineers interested in the development of distributed, collaborative decision-support systems. It is assumed that the reader has some level of knowledge in distributed systems, the Java™ programming language, the extensible markup language (XML), the unified modeling language (UML), rule-based languages (Jess™ and/or CLIPS), and specifics of any underlying operating system required.

Chapter 2 - Toolkit Usage

2.1 Introduction

This chapter outlines the basic steps required to use the TIRAC development toolkit through the use of a simple example. The full development process is described, starting with the construction of an object model up to generation of a complete information management framework. This chapter is intended to be a starting point for developers to get acquainted with the core development environment, and is therefore not meant to be all inclusive. The API documentation and example source should be consulted prior to beginning any serious development effort utilizing the TIRAC toolkit.

2.2 Creating An Object Model Document

The starting point for creating a framework, using the TIRAC toolkit, is the development of an object model describing the information domain specific to a project application. This section describes this process illustrated through a description of a simple object model in XMI-UML. The object model is shown in figure 2.1 in terms of a UML class diagram.

This model, while not all inclusive, illustrates several features that require special attention due to the stronger constraints and assumptions imposed by the processing tools (see section 3.3.2 for a comprehensive list of additional modeling requirements).

The framework model processing tools require input in the form of XMI-UML (an XMI document adhering to the UML meta-model - refer to chapter 3 for more information). The creation of this XMI document may be facilitated through the use of graphical modeling tools, such as Describe® [6], Rational® [8], and Poseidon for UML® [7]. For the purposes of this example, discussion will be restricted to the description of the required XMI elements. Dependent on the specific features of the graphical modeling tool, XMI export may be supported. However, the specific XMI produced may not adhere completely to the specification for UML 1.4. If there are incompatibilities it may be necessary to develop a translation (e.g., an XSL transform) that will take the exported XMI and produce a compliant XMI document. One such example is the XSL transform provided to correct a minor problem in exported XMI from the Poseidon UML modeling tool.

Sections of XMI-UML are shown below for clarity in the discussion. The full model document, for this example, may be referenced in Appendix C.1.

2.2.1 XMI header and meta-model specification

The root element for the model document must be an ***XMI*** element and must contain at least the following exactly as shown:

```
<?xml version="1.0"?>
<XMI xmlns:UML="org.omg.xmi.namespace.UML" xmi.version="1.1">
  <XMI.header>
    <XMI.metamodel xmi.version="1.4" xmi.name="UML"/>
  </XMI.header>
  <XMI.content>
    ...
  </XMI.content>
</XMI>
```

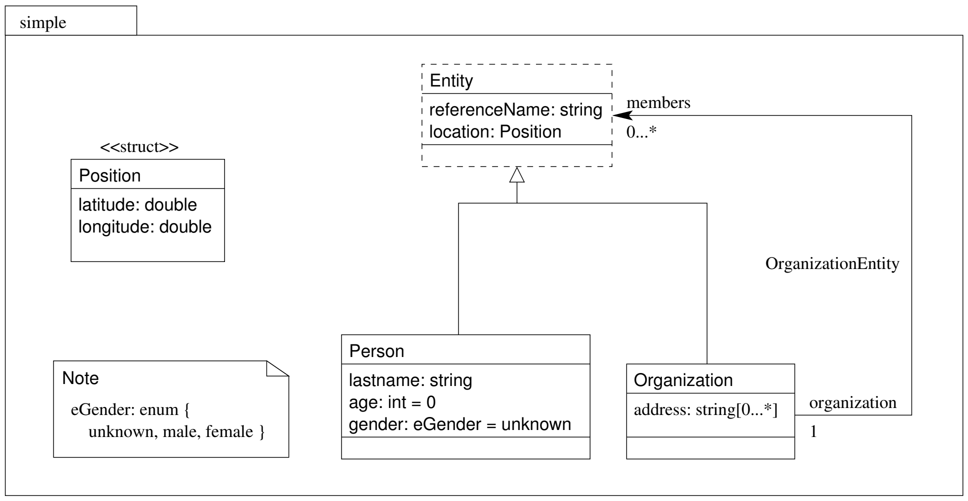


Figure 2.1: Simple Object Model.

2.2.2 Model element

A single ***Model*** element is required as the root namespace for the domain. The ***Model*** element must have a ***name*** attribute, whose value is unique from any enclosed package or referenced external model, and must contain owned elements. The following example defines a model (named “exampleModel”) that is defined as the root namespace containing (i.e., owning) all model elements.

```
<UML:Model xmi.id="ExampleModel" name="exampleModel">
  <UML:Namespace.ownedElement>
    ...
  </UML:Namespace.ownedElement>
</UML:Model>
```

Note: The model name must coincide with the name given to the XMI document without the .xmi suffix. Additionally, the ***xmi.id*** used by many of the ***XMI*** elements is required to be unique and will typically be generated by the modeling tool that produces the XMI file. In these examples, descriptive words will be used for the ***xmi.id*** for ease of reference.

2.2.3 Package element

The ***Model*** namespace may contain (as owned elements) any number of ***Package*** elements. Each ***Package*** element must have a ***name*** (unique from any enclosing or enclosed ***Package*** or ***Class*** elements), a ***namespace*** stereotype (i.e., a ***Stereotype*** element must be defined with its ***name*** attribute set to “namespace”), and must contain owned elements. The following example defines a package (named “simple”) that corresponds to the UML package depicted in figure 2.1.

```
<UML:Package xmi.id="SimplePackage" name="simple"
             stereotype="Namespace">
  <UML:Namespace.ownedElement>
    ...
  </UML:Namespace.ownedElement>
</UML:Package>
...
<UML:Stereotype xmi.id="Namespace" name="namespace"/>
```

Note: If a ***Package*** element is not stereotyped as a "namespace" the model processing tools will not process the package. In fact, if the package has any stereotype not defined as "namespace" it will not be processed. This may be useful in cases where packages may be defined as place-holders of content that should not be included in the generated system but could easily be "turned on" later by setting the stereotype to "namespace."

2.2.4 Class element

The ***Model*** namespace and/or ***Package*** may contain (as owned elements) any number of ***Class*** elements. Each ***Class*** element must have a ***name*** unique from any other ***Class*** or ***DataType*** elements defined within the immediate enclosing namespace. ***Class*** elements may reference a generalization if the class participates in one (i.e., it inherits from another class). Also, a ***Class*** element may be defined as abstract (i.e., non-instantiable). Additionally, ***Class*** elements may contain features such as attributes and operations. The following example defines an abstract class (named "Entity") that corresponds to the Entity class depicted in figure 2.1.

```
<UML:Class xmi.id="EntityClass" name="Entity" isAbstract="true">
  <UML:Classifier.feature>
    ...
  </UML:Classifier.feature>
</UML:Class>
```

A ***Class*** element may also contain ***Enumeration*** elements as owned elements. The following example defines a concrete class (named "Person") which extends the Entity class previously defined. In addition, the example illustrates the definition of an enumeration defined within the scope of the class.

```
<UML:Class xmi.id="PersonClass" name="Person"
           isAbstract="false" generalization="PersonEntity">
  ...
  <UML:Namespace.ownedElement>
    <UML:Enumeration xmi.id="GenderType" name="eGender">
      ...
    </UML:Enumeration>
  </UML:Namespace.ownedElement>
</UML:Class>
```

2.2.5 DataType and Enumeration elements

The ***Model*** namespace and/or ***Package*** may contain (as owned elements) any number of ***DataType*** and ***Enumeration*** elements. Additionally, ***Class*** elements may contain (as owned elements) any number of ***Enumeration*** elements. Data types used to define primitive types may be defined, refer to chapter 3 section 3.3.2 for a list of supported primitive types. For example, the following defines a ***DataType*** element named "int."

```
<UML:DataType xmi.id="IntType" name="int"/>
```

Struct data types (data type classifiers with structural features only) may be defined using a ***DataType*** element stereotyped as "struct" (i.e., a ***Stereotype*** element must be defined with its ***name*** attribute set to "struct"). Struct data types must define one or more attributes. The following example defines a ***DataType*** element representing a struct with the fields defined as attributes (classifier features) corresponding to the Position data type depicted in figure 2.1.

```
<UML:DataType xmi.id="PositionType" name="Position" stereotype="Struct">
  <UML:Classifier.feature>
    <UML:Attribute name="latitude" visibility="public"
                   changeability="changeable" type="DoubleType"/>
    <UML:Attribute name="longitude" visibility="public"
                   changeability="changeable" type="DoubleType"/>
  </UML:Classifier.feature>
</UML:DataType>
...
<UML:Stereotype xmi.id="Struct" name="struct"/>
```

Note: Data types may not participate in any generalization (i.e., data types may not extend or be extended by other data types).

Enumerations (data types constrained to a fixed set of literal values) may be defined using an ***Enumeration*** element. ***Enumeration*** elements must contain at least one enumeration literal. For example, the following defines an enumeration data type named “eGender” with a set of enumeration literals.

```
<UML:Enumeration xmi.id="GenderType" name="eGender">
  <UML:Enumeration.literal>
    <UML:EnumerationLiteral name="unknown"/>
    <UML:EnumerationLiteral name="male"/>
    <UML:EnumerationLiteral name="female"/>
  </UML:Enumeration.literal>
</UML:Enumeration>
```

2.2.6 Attribute element

Attribute elements may be defined as features within a ***Class*** element. Each ***Attribute*** element must have a ***name*** that is unique from any ***Attribute*** element defined within the scope or inherited scope of the class, a ***visibility*** (“public,” “protected,” “private”), a ***changeability*** (“changeable,” “frozen”), and a ***type*** reference. The following example defines a simple modifiable (changeable) attribute with ***name*** “referenceName” and ***type*** referring to a previously defined data type (in this case the “StringType” data type).

```
<UML:Attribute name="referenceName" visibility="public"
               changeability="changeable" type="StringType"/>
```

Attribute elements may also define an initial value. For example:

```
<UML:Attribute name="gender" visibility="public"
               changeability="frozen" type="GenderType">
  <UML:Attribute.initialValue>
    <UML:Expression body="unknown"/>
  </UML:Attribute.initialValue>
</UML:Attribute>
```

Attribute elements may also define their ***multiplicity*** (range of cardinality or number of element values). Supported multiplicity ranges currently include 0...1 or 0...* (upper range set to -1). For example, the following defines an address attribute typed as a string but having any number of values (unbounded multiplicity range).

```
<UML:Attribute name="address" visibility="public"
               changeability="changeable" type="StringType">
  <UML:StructuralFeature.multiplicity>
    <UML:Multiplicity>
      <UML:Multiplicity.range>
        <UML:MultiplicityRange lower="0" upper="-1"/>
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:StructuralFeature.multiplicity>
</UML:Attribute>
```

2.2.7 Association element

The ***Model*** namespace may contain, as owned elements, any number of ***Association*** elements. ***Association*** elements must have a ***name*** that is unique from any other ***Association*** element and must contain association connections (***AssociationEnd*** elements). For example, the following defines an ***Association*** element named "OrganizationEntity" that relates entities (as members) to an organization through association connections.

```
<UML:Association xmi.id="OrganizationEntityAssoc"
                 name="OrganizationEntity">
  <UML:Association.connection>
    ...
  </UML:Association.connection>
</UML:Association>
```

2.2.8 AssociationEnd element

Association elements must contain, as connections, two ***AssociationEnd*** elements. ***AssociationEnd*** elements must have a ***name*** that is unique from any ***AssociationEnd*** elements defined within the scope of the referring class and its scope of inheritance, navigability, a reference to the participating ***Class*** element and must contain a definition of its ***multiplicity*** (range of cardinality or number of element values). Supported multiplicity ranges include 0...1 or 0...* (upper range set to -1). Additionally, ***AssociationEnd*** elements may be defined as an aggregation (***aggregation*** attribute set to "aggregate"). The following example defines an ***AssociationEnd*** element, describing the "members" role that an entity plays with respect to an organization. The multiplicity range is defined as unbounded (any number of entities can be members of an organization).

```
<UML:AssociationEnd name="members" isNavigable="true"
                    aggregation="none" participant="EntityClass">
  <UML:AssociationEnd.multiplicity>
    <UML:Multiplicity>
      <UML:Multiplicity.range>
        <UML:MultiplicityRange lower="0" upper="-1"/>
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:AssociationEnd.multiplicity>
</UML:AssociationEnd>
```

2.2.9 Generalization element

Generalization links are defined using this element. ***Generalization*** elements must define a ***parent*** and ***child*** referencing defined ***Class*** elements. The following example defines the ***Generalization*** link between the entity and person classes. All child classes (classes extending other classes) must reference an appropriate generalization (see example PersonClass above).

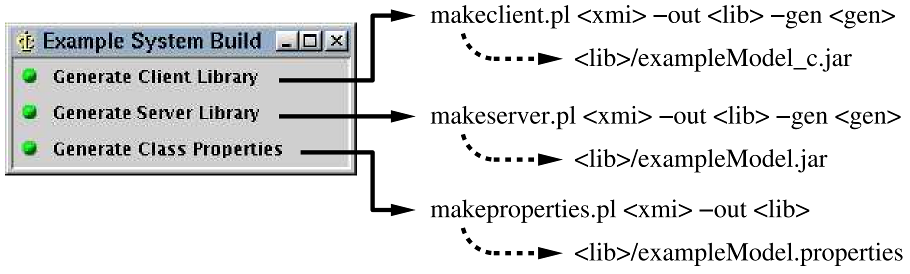
```
<UML:Generalization xmi.id="PersonEntity" parent="EntityClass"
                    child="PersonClass"/>
```

2.3 Processing The Object Model

Having produced an object model, in the form of an XMI-UML document, the model may now be processed into a TIRAC framework utilizing the model processing tools provided in the toolkit. The toolkit provides convenience scripts (see Appendix A.9 for script usage) for invoking the processing tools to produce the individual artifacts that ultimately constitute the domain specific framework. Processing the object model into an executable system will be described using the execution framework (see Appendix D) and the requisite XML suite documents, which utilize the convenience scripts provided.

2.3.1 Object model processing suite

The following describes the process cases that make up an example build suite defined in the execution suite XML schema. This example build suite (see Appendix C.2) will take in the example object model (shown in figure 2.1 and provided in XMI form in Appendix C.1) and produce the client interface library, server support library (object servants, factories, etc.), and object management class properties.


Example System Build
Generate Client Library
Generate Server Library
Generate Class Properties
makeclient.pl <xmi> –out <lib> –gen <gen>
<lib>/exampleModel_c.jar
makeserver.pl <xmi> –out <lib> –gen <gen>
<lib>/exampleModel.jar
makeproperties.pl <xmi> –out <lib>
<lib>/exampleModel.properties

build suite

The build suite defines a series of execution cases that, when processed, will result in a complete system framework that may be executed (services started up) with interacting generic client applications. The following sections describe specific execution cases that make up this build suite.

client interface library

This process case executes the convenience Perl script ***makeclient.pl***, taking the example object model XMI document as input, and producing the client interface library jar as an output artifact.

```
<command id="perl"><exec>perl</exec></command>
<case id="client" name="Generate Client Library">
  <command ref="perl">
    <exec path="uml.path">makeclient.pl</exec>
    <option id="out"><path ref="lib.path"/></option>
    <option id="gen"><path ref="gen.path"/></option>
```

```
      <option id="seq">object</option>
      <arg><path ref="xmi.path">exampleModel.xmi</path></arg>
    </command>
    <artifact path="lib.path">
      exampleModel_c.jar
    </artifact>
  </case>
```

This case is equivilent to executing the following[1]:

```
  perl <uml.path>/makeclient.pl <xmi.path>/exampleModel.xmi ↪
     -out <lib.path> -gen <gen.path> -seg=object
```

where “<uml.path>” is a defined path to the installed model processing scripts, “<xmi.path>” is a defined path to the example ***xmi*** directory, “<lib.path>” is a defined path to the build suite ***lib*** directory (i.e., ***example/build/lib***), and “<gen.path>” is a path to the current suite output directory (i.e., ***example/build/simplemodel/gen*** defined by the ***path*** “.”). See chapter 3, section 3.3.3 for more detailed description of the model processing tools and convenience Perl scripts.

servant class library

This process case executes the convenience Perl script ***makeserver.pl***, taking the example object model XMI document as input, and producing the servant class library jar as an output artifact.

```
  <case id="server" name="Generate Server Library">
    <command ref="perl">
      <exec path="uml.path">makeserver.pl</exec>
      <option id="out"><path ref="lib.path"/></option>
      <option id="gen"><path ref="gen.path"/></option>
      <option id="seq">object</option>
      <arg><path ref="xmi.path">exampleModel.xmi</path></arg>
    </command>
    <artifact path="lib.path">
      exampleModel.jar
    </artifact>
  </case>
```

This case is equivilent to executing the following (see above for path descriptions):

```
  perl <uml.path>/makeserver.pl <xmi.path>/exampleModel.xmi ↪
     -out <lib.path> -gen <gen.path> -seg=object
```

object management class properties

This process case executes the convenience Perl script ***makeproperties.pl*** taking the example object model XMI document as input, and producing the class properties file as an output artifact.

[1] The ↪symbol at the end of a line indicates that the following line is a continuation (i.e., no carriage return).

```
<case id="properties" name="Generate Class Properties">
  <command ref="perl">
    <exec path="uml.path">makeproperties.pl</exec>
    <option id="out"><path ref="lib.path"/></option>
    <arg><path ref="xmi.path">exampleModel.xmi</path></arg>
  </command>
  <artifact path="lib.path">
    exampleModel.properties
  </artifact>
</case>
```

This case is equivilent to executing the following (see above for path descriptions):

```
perl <uml.path>/makeproperties.pl <xmi.path>/exampleModel.xmi ↪
   -out <lib.path>
```

defclasses (COOL)

This process case executes the convenience Perl script makecool.pl taking the example model XMI document as input, and producing the COOL defclass definition file as an output artifact.

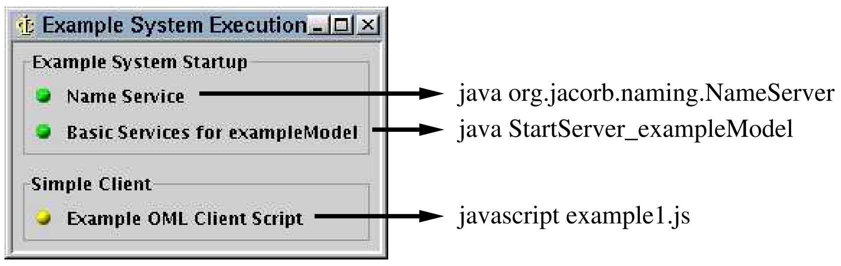
```
<case id="defclasses" name="Generate COOL Defclasses">
  <command ref="perl">
    <exec path="uml.path">makecool.pl</exec>
    <option id="out"><path ref="kb.path"/></option>
    <arg><path ref="xmi.path">exampleModel.xmi</path></arg>
  </command>
  <artifact path="kb.path">exampleModel.kbc</artifact>
</case>
```

This case is equivilent to executing the following (see above for path descriptions):

```
perl <uml.path>/makecool.pl <xmi.path>/exampleModel.xmi ↪
   -out <kb.path>
```

2.4 System Execution

After building a system, the services may then be started, along with clients interacting with the information served by these domain specific services. As examples, some simple execution suites are provided (shown in Appendix C.3), illustrating startup of the base services and model domain services (factories), as well as, the execution of some simple clients.


Example System Execution
Example System Startup
Name Service
Basic Services for exampleModel
Simple Client
Example OML Client Script
java org.jacorb.naming.NameServer
java StartServer_exampleModel
javascript example1.js

2.4.1 Example service execution suite

The execution suite used to startup the basic services is described below.

name service

Startup of the name service is provided through the use of an included execution suite. The first process case configures the JacORB CORBA ORB (specifies name service IOR URL) by creating a Java property file. The property file is generated from a property template file (a file containing core property settings with placeholder tokens for those property settings requiring configuration) using the ***MakeFile*** tool. In this case the token ***nsref*** is specified as an argument to the ***MakeFile***, which processes the template file (***jacorb.properties.tpl***), replacing all occurances of the token ***nsref*** with the specified value. This Java tool is executed within the suite execution process - inheriting the launcher classpath - through specification of the Java application main class and the absense of a defined classpath. If a classpath had been defined then the suite execution launcher would have launched the case as a separate process.

```
<case id="jacorbproperties">
  <command>
    <class>com.cdmtech.core.tool.build.MakeFile</class>
    <arg><path ref="tpl.path">jacorb.properties.tpl</path></arg>
    <arg><path ref="lib.path">jacorb.properties</path></arg>
    <arg>nsref=<url protocol="file" ref=".">NS_Ref</url></arg>
  </command>
  <artifact path="lib.path">jacorb.properties</artifact>
</case>
```

This case is equivilent to executing the following (see above for path descriptions):

```
java com.cdmtech.core.tool.build.MakeFile ↪
   <tpl.path>/jacorb.properties.tpl ↪
   <lib.path>/jacorb.properties nsref=<suite.url>/NS_Ref
```

where “<tpl.path>” is a defined path to the installed file templates, “<lib.path>” is a defined path to the execution suite ***lib*** directory (i.e., ***example/exec/lib***), and “<suite.url>” is a URL to the current suite output directory (i.e., ***file:<suite root>/example/exec*** defined by the protocol and ***path*** “.”). Note that this results in the creation of a JacORB properties file with just the single token ***nsref*** replaced, defining the value for the ***ORBInitRef.NameService*** property. Under most circumstances this will be the only property requiring modification in this file.

The subsequent case starts the name service as an asynchronous process after the case condition is satisfied (the string “JacORB V 2.1” is matched in the standard output of the process). Note the use of the artifact element to provide for conditional case execution. If the specified artifact exists prior to case execution then the case process will not be started. This particular artifact (a file named with the case ***id*** appended with “.run”) is created when the case process is started. Hence, if a previous case has been successfully executed that starts up the name service, then this case will not execute (only one name service is required for system execution). The option element is used here to pass options to the Java run-time, setting the ***boot.classpath*** to a previously defined class path. The argument passed to the name service specifies the path to a file that will contain a serialized Initial Object Reference (IOR) to the name service. This IOR is utilized by clients needing access to the name service. This can be thought of as a boot strap process since once client applications have access to the name service, references to all other registered services can be obtained through the name service. This name service IOR file can be accessed through various protocols using a URL defined in the property file generated by the previous execution case. This property file, and therefore the URL reference to the name service IOR, will be loaded and used by the JacORB ORB after initialization by the client application. All of this may seem rather complex, but for the most part is handled transparently when client applications utilize the client API provided by the TIRAC framework.

```
<case id="nameserver" name="Name Service">
  <command>
    <classpath><path ref="lib.path"/></classpath>
    <class>org.jacorb.naming.NameServer</class>
    <option>-Xbootclasspath/p:<classpath ref="boot.classpath"/></option>
    <arg><path ref=".">NS_Ref</path></arg>
  </command>
  <artifact path=".">nameserver.run</artifact>
  <condition>JacORB V 2.1</condition>
</case>
```

This case is equivilent to executing the following:

```
java -Xbootclasspath/p:<boot.classpath> -cp <lib.path> ↪
   org.jacorb.naming.NameServer <suite.path>/NS_Ref
```

where “<boot.classpath>” is a defined class path to libraries (JAR files) required by the JacORB name service, “<lib.path>” is a defined path to the execution suite ***lib*** directory (i.e., ***example/exec/lib***), and “<suite.path>” is a path to the suite output directory (i.e., ***example/exec*** defined by the ***path*** “.”).

domain service

This process case starts the domain specific services as an asynchronous process after condition satisfaction. These services include the domain factory services, the persistence service, and the notification/subscription services. The properties defined within this case are passed to the Java run-time as system properties. As with the name service the artifact ***exampleModelServer.run*** serves as a pre-condition to execution. If the base services have been previously started (by a case with the same fully qualified suite path and ***id***) then this case process will not be executed. **Note:** if for any reason these processes are interupted without allowing for normal shutdown (e.g., a power outage, etc.) then these “.run” files will remain even though the processes are no longer running. The existence of these files will prevent subsequent execution of the process cases that define these “.run” files as artifacts. Removing the files will allow execution of these process cases.

```
<case id="exampleModelServer" name="Start exampleModel Server">
  <command>
    <classpath ref="add.classpath">
      <path ref="*">lib</path>
      <path ref="*">lib/core_server.jar</path>
      <path ref=".">lib</path>
      <path ref="/">example/xmi</path>
      <path ref="lib.path"/>
      <path ref="lib.path">exampleModel_c.jar</path>
      <path ref="lib.path">exampleModel.jar</path>
    </classpath>
    <class>StartServer_exampleModel</class>
    <property id="org.omg.CORBA.ORBClass">
      org.jacorb.orb.ORB
    </property>
    <property id="org.omg.CORBA.ORBSingletonClass">
      org.jacorb.orb.ORBSingleton
    </property>
    <property id="core.properties">
      exampleModel_server.properties
    </property>
    <property id="core.persist.serial.location">
      <path ref=".">data/exampleModel</path>
    </property>
  </command>
  <artifact path=".">exampleModelServer.run</artifact>
  <condition>factories are started and ready</condition>
  <condition type="failure" pattern="EXCEPTION">
    exception occurred during service initialization
  </condition>
</case>
```

The classpath defined for this execution case includes paths required for both Java class archives (JAR files) as well as resource access. These paths are used specifically to obtain the following:

- ***<path ref="*">lib</path> => ./lib***
  Path to directory containing base properties.
- ***<path ref="*">lib/core_server.jar</path> => ./lib/core_server.jar***
  Path to JAR file containing base services and support classes.
- ***<path ref=".">lib</path> => ./example/exec/lib***
  Path to directory containing jacorb.properties required by JacORB (created by previously defined execution case in name service startup suite).
- ***<path ref="/">example/xmi</path> => ./example/xmi***
  Path to directory containing model XMI file required by model service.
- ***<path ref="lib.path"/> => ./example/build/lib***
  Path to directory containing ***exampleModel_server.properties*** specified by the ***core.properties*** system property.
- ***<path ref="lib.path">exampleModel_c.jar</path> => ./example/build/lib/exampleModel_c.jar***
  Path to JAR file (created by build suite) containing model domain specific client object and service interfaces.
- ***<path ref="lib.path">exampleModel.jar</path> => ./example/build/lib/exampleModel.jar***
  Path to JAR file (created by build suite) containing model domain specific object servants and services.

The properties defined for this execution case are used for the following:

- ***org.omg.CORBA.ORBClass***
  System property defining the name of the class implementing the CORBA ORB.
- ***org.omg.CORBA.ORBSingletonClass***
  System property defining the name of the class implementing the CORBA ORB singleton.
- ***core.properties***
  System property defining the name of the properties file used by various core services (must be located in class path).
- ***core.persist.serial.location***
  Property defining the location that will contain the root directory to hold serialized objects (persisted objects) - in this case the location is set to the path ***./example/exec/data/exampleModel.***

2.4.2 Example client script

The execution suite provided, illustrating simple client interaction, is described below. This suite will conditionally startup the base services (if the services were not previously started, as indicated by the existence of the ".run" artifacts, with the service execution suite described in section 2.4.1) then configures and executes the client script.

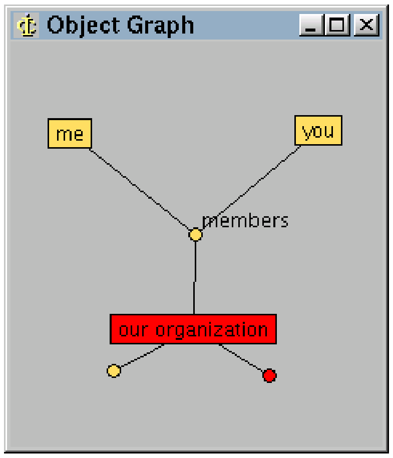
example client

This process case runs a javascript (see Appendix C.4) using the Mozilla JavaScript interpreter. This script invokes methods on various OML class methods illustrating simple object interactions. The objects created and modified by this script will be visually presented in the object graph client, as described below.

```
<case id="simpleexample" name="Simple Client">
  <command>
    <class>org.mozilla.javascript.tools.shell.Main</class>
    <classpath>
      ...
    </classpath>
    ...
    <property id="core.logLevel">
      Information
    </property>
    <property id="core.properties">
      exampleModel_om.properties
    </property>
    <arg><path>simpleExample.js</path></arg>
  </command>
</case>
```

2.4.3 Generic client applications

object graph

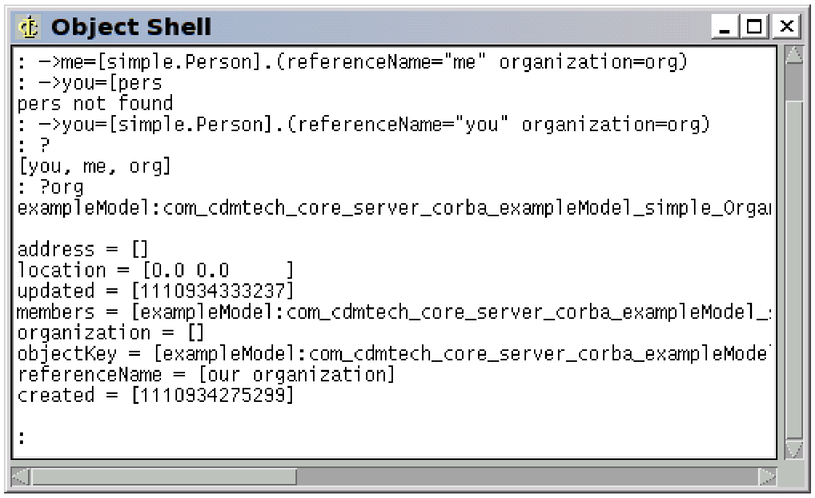


A simple visualization client is started by this process case. This suite will conditionally startup the base services (if the services were not previously started with the service execution suite described in section 2.4.1), then execute the object graph client.

```
<case id="objectgraph" name="Object Graph Client">
  <command>
    <class>com.cdmtech.core.client.gui.objectGraph.ObjectGraph</class>
    <classpath>
      ...
    </classpath>
    ...
    <property id="core.logLevel">Warning</property>
    <property id="core.properties">
      exampleModel_om.properties
    </property>
    <property id="simple.Entity.class.disAttrName">referenceName</property>
    <arg>simple.Organization</arg>
  </command>
  <condition>Initialized exampleModel_SubscriptionClient</condition>
</case>
```

The argument defined for the case process is used by the object graph client to establish an initial creation interest. In this example, the class “simple.Organization” is specified, which results in an interest in creation of organization objects. Upon notification of the creation of an organization the object graph will react by displaying an object node. The object graph presented is produced through registration of interests in object instance modification (initially on the root object). Directly associated objects are shown with links to the root object through link nodes (the small circles). With each object addition an object node is added and linked to the appropriate link node. If any links exist between the current root object (shown with a red background) and any other objects through their association roles, then those objects are also displayed with links to the relevant role (link node). Additionally, any objects associated with these immediate linked objects are shown with direct links (no link nodes) without regard to association roles. Objects outside this immediate neighborhood are not displayed. However, selecting an object node will result in the object being set as the root object node with the display updated based on the above display rules. Any link nodes shown in red may also be selected resulting in the opening or closing of the node links (display of linked objects).

object shell



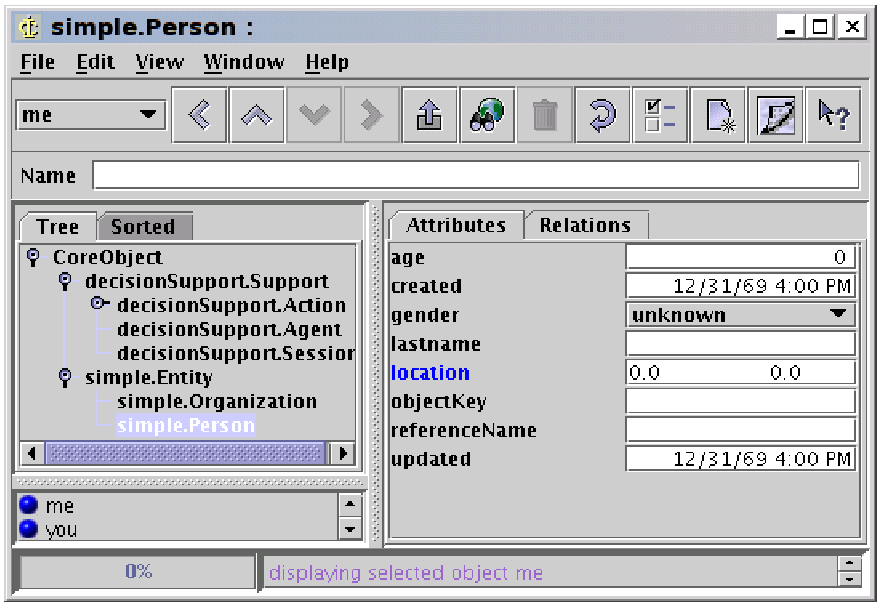
```
: ->me=[simple.Person].(referenceName="me" organization=org)
: ->you=[pers
pers not found
: ->you=[simple.Person].(referenceName="you" organization=org)
: ?
[you, me, org]
: ?org
exampleModel:com_cdmtech_core_server_corba_exampleModel_simple_Orga

address = []
location = [0.0 0.0      ]
updated = [1110934333237]
members = [exampleModel:com_cdmtech_core_server_corba_exampleModel_
organization = []
objectKey = [exampleModel:com_cdmtech_core_server_corba_exampleMode
referenceName = [our organization]
created = [1110934275299]

:
```

A generic client, providing a command-line interface to information objects served by the system, is started by this process case. This suite will conditionally startup the base services (if the services were not previously started with the service execution suite described in section 2.4.1), then execute the object shell client.

```
<case id="objectshell" name="Object Shell">
  <command>
    <class>com.cdmtech.core.client.shell.ObjectShell</class>
    <classpath>
      ...
    </classpath>
    ...
    <property id="core.logLevel">Warning</property>
    <property id="core.properties">
      models_om.properties
    </property>
  </command>
  <condition>Initialized exampleModel_SubscriptionClient</condition>
</case>
```

This client application uses the object management layer (OML) and therefore, discovers the structure of the information provided by the domain specific system at run-time. Usage of the object shell is described in Appendix F. The object shell provides commands for creating, deleting, modifying, and querying for objects.

instance viewer



age	0
created	12/31/69 4:00 PM
gender	unknown
lastname	
location	0.0 0.0
objectKey	
referenceName	
updated	12/31/69 4:00 PM

displaying selected object me

Another generic client, this time providing a graphical interface to information objects served by the system, is started by this process case. This suite will conditionally startup the base services (if the services were not previously started with the service execution suite described in section 2.4.1), then execute the instance viewer client.

```
<case id="instanceviewer" name="Instance Viewer">
  <command>
    <class>com.cdmtech.core.client.iv.InstanceViewer</class>
    <classpath>
      ...
    </classpath>
    ...
```

```
        <property id="core.properties">
          example_iv.properties
        </property>
      </command>
      <condition>Initialized exampleModel_SubscriptionClient</condition>
    </case>
```

This client application also uses the OML and therefore, is independent of any specific information domain (structure discovered at run-time). Usage of the instance viewer is described in Appendix E. The instance viewer provides functionality for creating, deleting, modifying, and querying for objects.

2.5 Multiple Information Domains

A principal aspect of the TIRAC architecture is centered around the distribution of information and the services that manage information. Client applications requiring access to information that is serviced across a distributed network of information services that need only to specify the model domains that define the information they require. Additionally, it is possible to design model domains that require access to other separately managed information domains through the use of associations to reference classes (classes defined in the external domain). Typically these associations will be one-way (uni-directional), as the reference class will usually have no requirement for accessing the referring class. Uni-directional associations provide a reference mechanism while retaining a high degree of decoupling, allowing access to established information domains where modification of the information model may not be desirable or even possible. An example of an information model that illustrates an association to an external reference class is described in section 2.6 and shown in figure 2.2.

2.5.1 Reference elements and uni-directional associations

In this model the class representing Entity is shown as a reference class (stereotype set to “reference”). Within the XMI model file this reference class is defined as follows.

```
    <UML:Model xmi.id="DecisionModel" name="decisionModel">
      <UML:Namespace.ownedElement>
        <UML:Package xmi.id="SimplePackage" name="simple"
                     stereotype="Reference">
          <UML:Namespace.ownedElement>
            <UML:Class xmi.id="EntityClass" name="Entity"
                       stereotype="Reference"/>
          </UML:Namespace.ownedElement>
        </UML:Package>
        ...
      </UML:Namespace.ownedElement>
    </UML:Model>
```

Notice that the ***Package*** (not the ***Model***) element containing the Entity ***Class*** element is also included and stereotyped as a reference. These referenced elements must be defined as owned elements of the model (i.e., the reference elements, not the actual elements, are owned by the referring model). Reference elements do not need to be fully replicated from the source model. However, the structure must be fully defined (element names and packaging must reflect the defined structure). Having defined these reference elements, they may then be referred to in associations within the referring model. For example, the following defines the ***Association*** element showing the “targets” end with the ***participant*** set to the entity class. Even though this association is uni-directional both ends of the association must be defined.

```
<UML:Association xmi.id="ActionTargetAssoc" name="ActionTarget">
  <UML:Association.connection>
    <UML:AssociationEnd name="targetAction" isNavigable="false"
                        aggregation="none"
                        participant="ActionClass">
      ...
    </UML:AssociationEnd>
    <UML:AssociationEnd name="targets" isNavigable="true"
                        aggregation="none"
                        participant="EntityClass">
      ...
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
```

2.5.2 Service startup

Similar to the startup for the example model services (see section 2.4.1), the following execution case specifies the startup class for the domain specific services and all required class paths and properties. Additionally, the class path includes the client library for the example model domain (exampleModel_c.jar). This library is required since the decision-support model requires access to features of the example model (notably the Entity class through the Action associations, see figure 2.2).

```
<case id="decisionModelServer" name="Start decisionModel Server">
  <command>
    <class>StartServer_decisionModel</class>
    <classpath>
      ...
      <path ref="buildpath">exampleModel_c.jar</path>
      <path ref="buildpath">decisionModel_c.jar</path>
      <path ref="buildpath">decisionModel.jar</path>
    </classpath>
    ...
    <property id="core.properties">
      decisionModel_server.properties
    </property>
    <property id="core.persist.serial.location">
      <path ref=".">data/decisionModel</path>
    </property>
    ...
  </command>
  <artifact path=".">decisionModelServer.run</artifact>
  <condition>factories are started and ready</condition>
  <condition type="failure" pattern="EXCEPTION">
    exception occurred during service initialization
  </condition>
</case>
```

2.5.3 Client configuration and startup

Client applications requiring access to multiple information domain services require some additional configuration (see chapter 4, section 4.3).

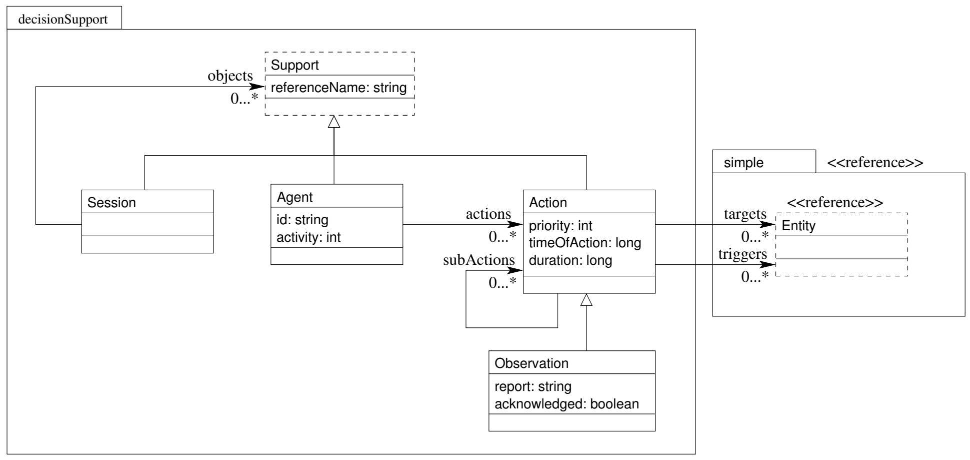


Figure 2.2: Decision Support Model.

Additionally, access to the client interfaces for both model domains must be provided to client applications. For example, startup of the object shell client may be defined using the following execution case.

```
<case id="objectshell" name="Object Shell">
  <command>
    <class>com.cdmtech.core.client.shell.ObjectShell</class>
    <classpath>
      ...
      <path ref="buildpath">exampleModel_c.jar</path>
      <path ref="buildpath">decisionModel_c.jar</path>
    </classpath>
    ...
  </command>
</case>
```

2.6 Decision Support Example

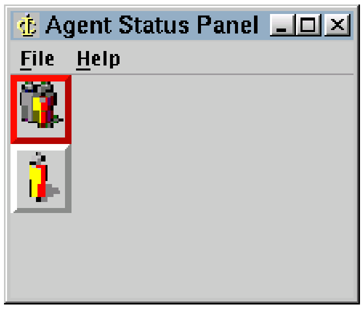
The object model (shown in figure 2.2) and execution suites are presented to illustrate a more concrete and complete example system that provides decision support.

This simple example model describes object classes that will be used to represent software agents (software modules incorporating decision-support logic applied to specialized information domains) and their possible actions. Software agents can be viewed simply as additional knowledge experts analogous to their human counterparts at least in the context of information assessment. Therefore, it makes sense to represent agents in the same way individual people collaborating in the decision-making process might be represented. Through this representation, agents (and for that matter human collaborators) can provide feedback as additional information entering the system (in this case in the form of actions and observations) allowing interested parties to receive this feedback using exactly the same mechanisms that would be applied to any other information interests.

This model defines a general action as an object with temporal characteristics (start time and duration) and priority. Actions can be applied to targets (represented through the ***targets*** association to ***Entity***) and be caused by triggers (also an association to ***Entity***). Actions may have subsequent actions (represented by the recursive association ***subActions***) which could be interpreted to mean actions performed as a consequence of a previous action. The sub-class ***Observation*** is defined as a kind of ***Action***, adding the notion of a reportable observation and acknowledgment. Common feedback provided by agents (defined within the context of this particular example model) is typically in the form of observations, to indicate possible conditions requiring attention.

Note: This model is simply an example of one possible representation for agents and their feedback. There is no constraint on how (or even if) agents are represented. However, there are benefits associated with explicit agent representation. It is often desirable to present agent feedback with respect to the agent that produced the feedback. Presenting this association further highlights the meaning of the feedback. One such presentation device is the Agent Status Panel client application provided as part of the TIRAC toolkit.

2.6.1 Agent Status Panel



This client is provided as an example to illustrate one possible method for displaying agent feedback and also serves to further illustrate the architecture provided by the TIRAC framework. The Agent Status Panel is in fact a very simple client that knows how to graphically display agents (as icons) and their associated feedback (alerts presented textually in a dialog) but otherwise has no knowledge of specific agents or their purpose. To effectively utilize the Agent Status Panel there should be a domain model defined that contains a class that would be suitable for agent representation. This class should also define an association to another class representing agent feedback. In this example, the decision support model, shown in figure 2.2, defines an ***Agent*** class that defines an association to ***Action*** which in turn defines an ***Observation*** subclass. These classes (***Agent*** and ***Observation***) satisfy the requirements for utilizing the Agent Status Panel.

2.6.2 Agent Session

Execution suites are provided to illustrate startup of agent sessions utilizing both the JESS and CLIPS inference engine implementations.

Note: use of the JESS based agent engine will require installation of JESS (see chapter 7).

Agents

Example agents are provided to illustrate use of both the JESS and CLIPS based agent engines. Dependent on which agent session is started the appropriate set of agent rules will be loaded. In either case, the example agent will react to the creation of organization objects by posting an observation providing an organization size classification (small, medium, or large). See Appendix C.5 for listings of both sets of agent rules.

2.7 System Development

Once a domain specific framework is created, specialized system development may begin. Typically, if definition of an information domain is part of the overall system design then specialized applications should also be designed in parallel to the domain model development. For highly specialized systems the end user applications will have a significant impact on the required underlying information structure. Likewise, required decision support will also have an impact and, therefore, agent logic design should begin early in the overall system design. However, even the best formulated system design will require some iteration after development has begun. There will be cases where information structure will have to change to accommodate system design changes. These structural changes can have minimal impact on system development if the environment for building the underlying infrastructure is, to high degree, automated. The tools provided in the TIRAC toolkit aid in the process of system development by automating the building of the system infrastructure driven solely by the domain model design. Turn around time for incorporating information structural changes is minimized allowing development to be focused on end user applications and decision support.

Distributed system frameworks, while providing tremendous flexibility, tend to be complex in their underlying infrastructure. While this complexity is typically hidden from the application user and developer, when problems surface, finding the cause and solution can be difficult. To minimize the impact of these problems it is highly recommended that a well defined test process and capability be developed and incorporated early in the development process. Additionally, there are other principle guidelines that can help prevent particularly difficult problems (e.g., performance and resource utilization, etc.) The following sections briefly outline some common guidelines that may help reduce problems with respect to system development based on a TIRAC framework.

2.7.1 Model Design and Processing Guidelines

These general guidelines are suggestions that, if utilized, may reduce model processing and maintenance problems. Refer to chapter 3 for specific information regarding model processing.

- Use packages (namespaces) to logically group model sub-domains. This helps to reduce the complexity of the overall model and provides some additional flexibility in system deployment (separate object factory generated for each model sub-domain).
- Use consistent naming conventions (e.g., lowercase “camel” style for attribute and association role names, uppercase “camel” style for classes, etc.)
- Define enumerations only if absolutely required. Hard-coded enumerations limit model use.
- Limit read-only accessible attributes. Accessibility is a hard constraint which requires a system rebuild to change.
- Avoid large enumerations. Large flat lists are generally best broken up into categories or taxonomies. They can also be an indication that the type is a catch-all.
- Avoid cross package associations, or if required then define as uni-directional. This avoids possible cyclic package dependencies and enables more effective domain reuse.
- Avoid multiple aggregate owners. May cause confusing results when an aggregation is destroyed where its aggregate parts are also owned by another aggregation.
- Avoid class generalization links between sibling packages. This avoids possible cyclic package dependencies and enables more effective domain reuse.
- Avoid defining any class generalization links that define parent in nested package with respect to child. This avoids possible cyclic package dependencies and enables more effective domain reuse.
- Avoid use of language (Java, CORBA IDL, CLIPS, etc.) specific keywords for any model elements or features. Errors in compilation (IDL, Java) or agent engine initialization (agent load) may indicate keyword name collisions.
- Run ***makereport.pl*** script on model XMI file. Make note of any errors reported and correct as required.
- Process model XMI to clean location (i.e., to a location that contains no other source code). Errors in Java source compilation may result from old source code from a previous failed build.

- Use sequence type "object" (i.e., use option setting ***-seq=object*** when executing model processing scripts). This results in a system built using the servant sequence type supporting association management. Servant association management is more robust than client association management especially in systems with multiple client applications interacting together.

2.7.2 Application Programming Guidelines

These general programming guidelines address issues specific to the Object Management Layer (OML) programmer's interface. For details on the OML refer to chapter 4.

- Minimize the number of calls made to methods that require remote method invocations (***POW.get***, ***POW.post***, ***Template.getObject***, etc.) Although the OML tries to limit remote calls internally, it is best to avoid calling these methods unnecessarily.
- Set all attribute values prior to posting an object. This is especially true for read-only accessible attributes (value can only be set at initial post). Setting attribute values after initial post will result in possibly unnecessary remote method calls.
- Avoid calls to ***POW.getAssoc*** method unless required. This method is especially expensive on first access since any objects referenced by the association end will be resolved to, resulting in possibly many remote method calls. This call would be exactly the same as getting object references, through a call to ***POW.get***, and then resolving to each object (i.e., calling ***Template.getObject***).
- Any ***POW*** instances passed to the ***POW.set*** method must have been previously posted (i.e., the object must be instantiated prior to reference in an association role).
- Object deletion listeners should not attempt to get the object referenced in the deletion event (i.e., do not call the ***Template.getObject*** method).
- Association role modification listeners should not attempt to get any objects referred to in the removed object list of the event. These objects may have been deleted and attempts to get (resolve to) them will result in exceptions.
- As a general rule it is best to avoid listening for association role modifications, instead, if possible, listen for associated object instance creation and deletion (may require use of constraint-based subscription service, see chapter 5).
- If a large number of instance based listeners are required consider using a class based listener instead.

2.7.3 System Deployment and Configuration Guidelines

These general guidelines are suggestions that, if utilized, may reduce problems encountered during system execution.

- Run the name service on a host machine that is also running a Web server. Specify a location for the name service Initial Object Reference (IOR) accessible through the Web server as a URL using the HTTP protocol. This gives greater flexibility for distributing services and clients across a network. For example, if the following IOR URL for the name service is specified in the JacORB properties file (see section 2.4.1), then the name service should be started on the machine "webhost.com" with the IOR file located at the root document directory for the Web server running on "webhost.com." All client applications requiring access to the system would need this JacORB properties file to obtain the initial reference to the name service which, in turn, is used to obtain references to system services.

```
In jacorb.properties: ORBInitRef.NameService=http://webhost.com/NS_Ref
On webhost.com: java ... org.jacorb.naming.NameServer <web root>/NS_REF
```

- Distribute CPU-intensive client applications. If possible, do not execute an agent engine on the same machine along with any other client applications, especially user interface clients. Ideally, an agent engine should be isolated on a separate host machine.
- Use a database back-end for the object instance store. Compared with simple object serialization, databases may perform better, utilize less resources, and scale better as information content increases.
- If using object serialization for persistence, isolate the domain services (persistence, subscription, factories) to a separate host machine (i.e., do not execute client applications on the same machine hosting the domain services).

Chapter 3 - Model Processing Tools

3.1 Introduction

A key capability provided by the TIRAC toolkit is the ability to process an object based information model to produce a complete information management infrastructure. The Unified Modeling Language (UML) standard [14] was chosen as the high-level language for describing object models within the context of the TIRAC framework. Additionally, the Extensible Markup Language (XML) and, in particular, the XML Metadata Interchange (XMI)[11] for the UML meta-model (version 1.4)[14] was selected as the storage format for object model descriptions. The XMI-UML meta-model format is a standard that is supported by a number of modeling tools and, therefore, allows for a significant amount of flexibility in the selection of supporting applications. Models developed for use with the TIRAC toolkit must adhere to some additional modeling requirements. These requirements are described in section 3.3.2. Model processing tools have been developed that produce various artifacts (code, reports, etc.) utilizing the XMI-UML output. These tools make extensive use of a UML meta-model based parser built using a parser generator provided by the NovoSoft Meta-Data Framework (NSMDF) [10].

3.2 Architecture Overview

3.2.1 UML Processor

The ***UMLProcessor*** abstract class provides basic functionality for processing UML models (XMI). It defines production methods (implemented empty) that are invoked in a prescribed order, based on natural hierarchal and associative relationships, during processing of a defined namespace. This class must be subclassed to satisfy specific production requirements (overriding the defined empty production methods).

The order of production is defined as follows:

1. Process package (or model) namespace
   (a) process package scoped enum datatypes (see process datatype)
   (b) process package scoped struct datatypes (see process datatype)
   (c) process package root classes (see process class)
   (d) optionally process child (nested) packages (recursive)
   (e) process associations
2. Process class
   (a) process class scoped enum datatypes (see process datatype)
   (b) process features (attributes, association roles)
   (c) process inherited features
   (d) process operations
   (e) optionally process child (specializing) classes (recursive)
3. Process datatype
   (a) process nested enum datatypes (structs only)

(b) process features (enum values, struct fields)

This prescribed order (based on the model hierarchy), dictated by the ***UMLProcessor***, enables specific producers to easily create a consistent output structure.

Specialized producers should be defined to produce specific parts of a required artifact with a top-level producer managing the complete production. The top-level producer typically instantiates sub-producers to handle processing of nested model elements (such as classes, datatypes etc.). The following sections illustrate this by describing some of the implementations of the artifact production tools provided as part of the UML processing tools.

3.2.2 IDL Producer

The ***IDLProducer*** extends ***UMLProcessor*** to implement production management for the creation of CORBA IDL [12, 13]. Figure 3.1 shows the class diagram comprising the complete IDL production. The ***IDLProducer*** class implements basic functionality required to produce IDL interface definitions. The ***CoreIDLProducer*** class extends this functionality by adding core specific production requirements, including the production of namespace factory definitions (through the FactoryProducer). Additionally, the ***IDLUtility*** class provides specific utility functions useful for generation of IDL.

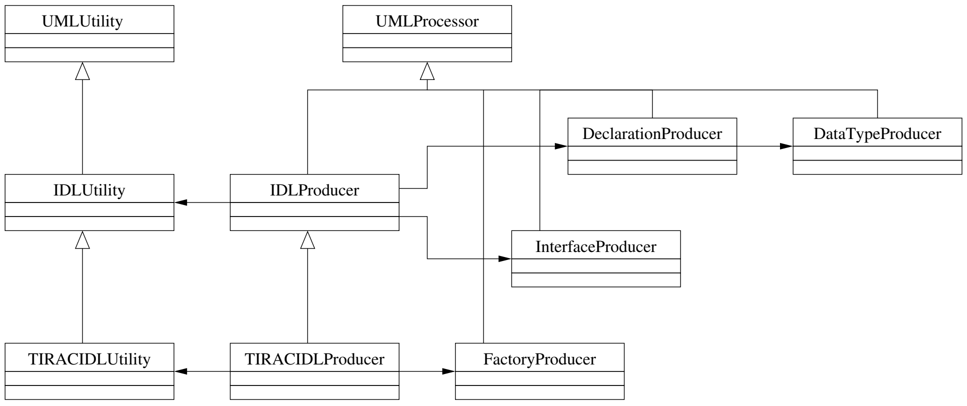


Figure 3.1: IDL Production Class Diagram.

3.2.3 Java Producer

The ***JavaProducer*** extends ***UMLProcessor*** to implement production management for the creation of Java classes. The ***JavaProducer*** class implements basic functionality required to produce Java classes and must be extended to implement specialized requirements. Additionally, the ***JavaUtility*** class provides general utility functions useful for output of Java class code.

Servant Producer

The ***ServantProducer*** is an example of a production that extends the ***JavaProducer***. The ***ServantProducer*** implements specific functionality for the production of Java servant implementations of CORBA skeleton classes produced by processing IDL. Figure 3.2 shows the class diagram comprising the complete servant production. The ***ServantProducer*** specifically manages the production of servant and factory classes. The ***ServantClassProducer***, in turn, manages production of class member declarations, constructor, and get/set methods.

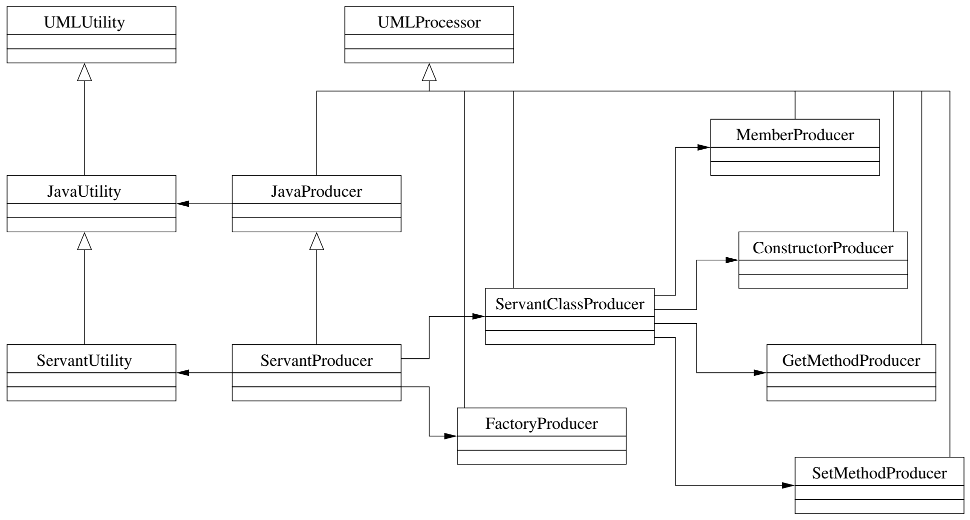


Figure 3.2: Java Servant Production Class Diagram.

3.3 Using the Model Processing Tools

A set of tools is provided to process UML models (XMI) into specific artifacts. The provided tools generate code for CORBA IDL, simple Java classes and types for localized implementations, and the servant and wrapper classes for distributed implementations. Additional tools include producers for OML (Object Management Layer - see chapter 4) class properties, as well as documentation (in LaTeX [15, 9]).

3.3.1 Modeling Tools

A number of graphical modeling tools [7, 6, 8] are available that support the UML methodology with export to XMI. The Poseidon for UML™ [7] modeler uses XMI as its native storage format. The XMI produced by Poseidon may be used directly, with the notable exception that Poseidon does not yet support enumeration datatype definition as defined by the UML 1.4 specification. Therefore, output from Poseidon must be transformed if any enumerations are defined in the model. A transform to correct this problem is provided and also serves as a simple example.

3.3.2 Modeling Requirements

This section outlines the additional modeling constraints imposed by the TIRAC toolkit on usage of the UML meta-model as defined in the OMG Unified Modeling Language (UML) Specification Version 1.4 [14]. The following is a checklist summarizing these additional modeling requirements.

- Packages
  - A ***Model*** element must be defined as the top-level container of all model elements.
  - The ***Model*** element must define a ***name*** that matches the name of the XMI file (without its ".xmi" suffix).
  - ***Package*** elements must be defined with "namespace" ***Stereotype*** (for factory generation required for packages with concrete classes/elements).
  - ***Package*** element names must be unique (i.e., no packages defined in the model with duplicate names).
  - ***Package*** elements must not define any cross-package class generalizations that would result in circular dependence (i.e., traversal of compete inheritence path should never return to the same package).
  - Referenced ***Package*** elements (defined in another model) must be defined with "reference" ***Stereotype.***
- Classes and Datatypes
  - Struct datatypes must be defined using ***DataType*** with "struct" ***Stereotype***.
  - ***Datatype*** elements may only be referenced within the scope (containing and/or inheriting) in which they are defined.
  - ***Class*** and ***DataType*** element names must be unique within the containing scope.
  - ***Enumeration*** literal names must not duplicate any ***Class*** or ***DataType*** name defined within the containing scope.
  - Referenced ***DataType*** and ***Class*** elements (defined in another model) must be defined with "reference" ***Stereotype.***
  - ***DataType*** elements defining primitive types are restricted to types supported by the IDL to Java mapping specification [13].
  - ***DataType*** elements may not participate in any generalization (i.e., datatypes cannot extend or be extended).
- Attributes
  - All ***Attribute*** elements must be defined with ***ownerScope*** set to "instance."
  - All ***Attribute*** elements must be defined with ***changeability*** set to "changeable" or "frozen."
  - All ***Attribute*** element names are unique within the scope (self and inheriting) of a ***Class***.
  - ***Attribute*** elements defined using an ***Enumeration*** datatype must have an ***initialValue.***
  - Optionally defined "unitsOfMeasure" ***TaggedValue*** (with ***dataValue*** set to unit of measure - see section 3.3.2) for numeric typed attributes.
- Associations
  - All ***Association*** elements must have at least one navigable end (***isNavigable*** set to "true").
  - All ***Association*** elements must have a unique association ***name*** (i.e., no ***Association*** elements defined in the model with duplicate names).
  - All ***Association*** element role (***AssociationEnd***) names are unique within the scope (self and inheriting) of a ***Class***.
  - For cross-domain uni-directional associations:
    - The navigability of the ***AssociationEnd*** with ***participant*** set to the referenced ***Class*** must be set to navigable (i.e., ***isNavigable*** set to "true").
    - The navigability of the ***AssociationEnd*** with ***participant*** set to the referring ***Class*** must be set to non-navigable (i.e., ***isNavigable*** set to "false").

    * The stereotype of the referenced ***Class*** (the ***participant*** at the navigable end) must be set to "reference."
    * The package containing the referenced ***Class***, as well as all containing packages, must also be included and stereotyped as "reference."

- Operations
  - Only ***Class*** elements may contain ***Operation*** definitions.
  - All ***Operation*** elements must be defined with ***ownerScope*** set to "instance."
  - ***Operation*** implementation must be defined with "implementation" ***TaggedValue*** with ***dataValue*** set to implementation definition.

Packages

Multiple packages may be defined in either model or package scope (packages may be nested). Any package that contains concrete (non-abstract) classes must be stereotyped as a "namespace." For each namespace a separate factory will be generated by the code generation tools. It is through these factories that object instances are constructed. Any classes defined outside of package scope will be managed by a generated "domain" namespace (i.e., a factory will be generated for the domain object model to manage all non-packaged classes).

Classes

Classes may be defined as concrete or abstract. Classes may be stereotyped as "reference," which will result in no code generated for the class. However, references to the class will be generated (e.g., inheritance, associations, etc.)

Attributes

All attributes must be instance scoped (i.e., no static or class scoped members are supported). All accessible attributes must be visible externally (i.e., public). Attributes that are not visible externally will result in no generated code exposing the attribute in the client interface (code will be generated supporting the attribute in the object servant class only providing internal access with persistence). Attribute changeability may be set to either "frozen" or "changeable." If set to "frozen" then no public set accessor method will be generated (i.e., the attribute is read-only).

The following primitive types are supported:

- string (unbounded character array)
- bool or boolean
- char (8 bit signed character)
- short, unsigned short (16 bit integer number)
- int, unsigned int (32 bit integer number)
- long, unsigned long (64 bit integer number)
- float (32 bit floating point number)
- double (64 bit floating point number)

Numerical attributes (e.g., type short, int, long, float, or double) may be given a unit of measure by including a tagged value with name “unitsOfMeasure”. Two units of measure may be specified using a colon to delimit the pair. The first measure defined is assumed to be the internal storage unit of measure, where the second is the default display unit of measure. Note that no code will be generated to support unit of measure conversion, instead client-side support is provided by the OML (see chapter 4), which in-turn utilizes the generated properties to identify and perform requisite conversions. The supported units of measure (supported tag value shown in parentheses) include the following:

- displacement - millimeter (mm), centimeter (cm), meter (m), inch (in), foot (ft), yard (yd), kilometer (km), mile (mi), nautical mile (nm)
- area - square millimeter (mm2), square centimeter (cm2), square meter (m2), square inch (in2), square foot (ft2), square yard (yd2), square kilometer (km2), square mile (mi2), square nautical mile (nm2)
- volume - cubic millimeter (mm3), cubic centimeter (cm3), cubic meter (m3), cubic inch (in3), cubic foot (ft3), cubic yard (yd3), liter (ltr), gallon (gal)
- speed - meters per second (m/s), feet per second (ft/s), kilometers per hour (km/hr), miles per hour (mi/hr), knots
- weight - gram (gm), kilogram (kg), metric ton (mt), ounce (oz), pound (lb), ton
- time - date, millisecond (ms), second (s), minute (min), hour (hr)
- temperature - celsius (C), farenheit (F), kelvin (K), rankine (R)
- angle - degree (deg), latitude (lat), longitude (lon)
- rate - liter per hour (ltr/hr), gallon per hour (gal/hr), pint per hour (pt/hr), ounce per hour (oz/hr)
- usage - hours per day (hrs/day), minutes per day (mins/day), seconds per day (secs/day)

Associations

All associations between classes must be either a simple association or an aggregation. General composition is not currently supported, however, the ability to define structured data types (structs) may be considered a kind of composite relationship (i.e., an instance of a structured data type is a physical part of its owner and cannot exist outside of the context of its owner). All associations must have at least one navigable end. Note that even for uni-directional associations (i.e., associations with only one navigable end) roles will be defined for both ends. Uni-directional associations that span model domains must be defined with the referenced end (participant class stereotyped as “reference”) navigable. All associations and aggregations must have a unique association name (within the scope of the complete domain) and all roles defined in the scope of a class must be unique.

Dependant on the sequence type selection specified in model processing, association (and aggregation) ends may be managed by servant sequences. For the simple sequence type support. Basic array type management is provided for the simple sequence type. Additionally, association management functionality is provided for the object sequence type.

Operations

All operations must be instance scoped (i.e., no static or class scoped methods are supported). The operation implementation is defined using a tagged value with the tag name set to “implementation.” The value specified will be placed verbatim into the body of the implemented operation method. For complex operations it is highly recommended that use of a “delegate” class be employed to contain the actual implementation as opposed to including the code directly into the model.

3.3.3 Processing Tools

The following sections describe use of the model processing tools and some specific issues to be aware of. All processing tools may be executed directly from their respective Java main class or through use of a Perl convenience script. In either case, a summary of command line usage may be obtained through use of the ***-help*** option. Use of the Perl script simplifies command line usage (e.g., no Java class path need be specified) and is therefore recommended. For details on tool script command line usage and syntax please refer to the release notes in Appendix A.9.

Model report generation

This tool produces a simple model report (in XML) that provides basic model structure and highlights model problems. An XSL transform is provided to extract model problem reports and output to simple text. This provides for a quick summary of modeling issues that require attention before subsequent model processing.

The Java main class for execution of the model report producer is ***ReportProducer*** (***com.cdmtech.core.tool.-uml.model*** package). A Perl convenience script called ***makereport.pl*** is provided in ***<install dir>/scr/uml***. The following is an example use of this script:

```
-> perl <install dir>/scr/uml/makereport.pl exampleModel.xmi
```

This results in the creation of a file called ***exampleModel.xml*** (in the current directory), which contains the XML report. Additionally, the ***makereport.pl*** script produces and displays a model summary (using an XSL transform) if any model problems are detected.

The XSL transform, ***ModelReportSummary.xsl*** may be used to obtain a quick summary of model problems. The makereport.pl script uses this transform to produce a model summary if any model problems are detected. For example, using the ***transform.pl*** script provided in ***<install dir>/scr*** the following will produce the summary output for the above example model report[1]:

```
-> perl <install dir>/scr/transform.pl ↪
     -xsl <install dir>/doc/xsd/ModelReportSummary.xsl exampleModel.xml
```

Additionally, the XSL transform, ***ModelReportToHTML.xsl*** is provided to produce an HTML report suitable for display in a Web browser. For example:

```
-> perl <install dir>/scr/transform.pl ↪
     -xsl <install dir>/doc/xsd/ModelReportToHTML.xsl ↪
     -out exampleModel.html exampleModel.xml
```

Client interface library generation

The Java code used to build the client object interface library is produced by the client wrapper producer. The code produced consists of classes that reflect the system domain model structure and wrap calls to the underlying CORBA object interface methods. The CORBA client interfaces are produced by an IDL (Interface Definition Language) compiler. The IDL in turn is produced by the IDL producer.

The Java main class used for executing the IDL producer is ***CoreIDLProducer*** (***com.cdmtech.core.tool.uml.-idl*** package). The Java main class for execution of the client wrapper producer is ***WrapperProducer*** (***com.-cdmtech.core.tool.uml.java.wrapper*** package). A Perl convenience script called ***makeclient.pl*** is provided in ***<install dir>/scr/uml***. This script calls the client wrapper producer, the IDL producer, and the IDL compiler to generate Java source code. The script then compiles all the code, and packages the resulting Java bytecode into a JAR file. The following is an example use of this script:

[1]The ↪symbol at the end of a line indicates that the following line is a continuation (i.e., no carriage return).

```
-> perl <install dir>/scr/uml/makeclient.pl exampleModel.xmi ↪
     -gen temp -out lib -seq=object
```

This results in the creation of the JAR file ***exampleModel_c.jar***, which is placed in the ***lib*** directory. All generated code is placed temporarily in the ***temp*** directory. The generated code is deleted after code compilation (unless the ***-keep*** option is used).

Note: Notice the use of the ***-seq*** option. This option is used to indicate to the code producer which sequence type management support, for association ends, is required in the generated code. In this case, the "object" value indicates that the sequence type supporting object association management is required. It is important that this option value be specified for the servant class generation (described in the next section) as well. Additionally, selection of the sequence type will also affect the configuration of client applications that use the Object Management Layer (see chapter 4, section 4.3).

Servant class library generation

The Java code used to build the servant object class library is produced by the servant producer. The code produced consists of classes that reflect the system domain model structure and implement object life-cycle management behavior (i.e., creation, access, and modification). The CORBA servant skeleton classes (classes providing structure but no implementation) are produced by an IDL (Interface Definition Language) compiler. The IDL in turn is produced by the IDL producer.

As above, the Java main class used for executing the IDL producer is ***CoreIDLProducer*** (***com.cdmtech.core.-tool.uml.idl*** package). The Java main class for execution of the servant producer is ***ServantProducer*** (***com.-cdmtech.core.tool.uml.java.servant*** package). A Perl convenience script called ***makeserver.pl*** is provided in ***<install dir>/scr/uml***. This script calls the servant producer, the IDL producer, and the IDL compiler to generate Java source code. The script then compiles all the code, and packages the resulting Java bytecode into a JAR file. The following is an example use of this script:

```
-> perl <install dir>/scr/uml/makeserver.pl exampleModel.xmi ↪
     -gen temp -out lib -seq=object
```

This results in the creation of the JAR file ***exampleModel.jar*** which is placed in the ***lib*** directory. All generated code is placed temporarily in the ***temp*** directory. The generated code is deleted after code compilation (unless the ***-keep*** option is used).

Note: The value used for the ***-seq*** option is the same value specified for the client wrapper producer, as described in the previous section.

Object management class property generation

The Object Management Layer (OML) requires additional information describing model structure and features. This information is provided as Java properties in a structure reflecting the class hierarchy of the object model. These class properties are produced by the OML class property producer.

The Java main class for execution of the class property producer is ***CoreOMLProducer*** (***com.cdmtech.core.tool.-uml.oml*** package). A Perl convenience script called ***makeproperties.pl*** is provided in ***<install dir>/scr/uml***. The following is an example use of this script:

```
-> perl <install dir>/scr/uml/makeproperties.pl exampleModel.xmi ↪
     -out lib
```

This results in the creation of a file called ***exampleModel.properties*** (in the ***lib*** directory) which contains the generated class properties.

COOL defclass generation

The CLIPS Object-Oriented Language (COOL) extensions requires definition of object classes. These object classes are provided in the CLIPS COOL language syntax in the form of defclass constructs. These class definitions are produced by the COOL defclass producer.

The Java main class for execution of the COOL defclass producer is ***CoreCoolProducer*** (***com.cdmtech.core.-tool.uml.cool*** package). A Perl convenience script called ***makecool.pl*** is provided in ***<install dir>/scr/uml***. The following is an example use of this script:

```
    -> perl <install dir>/scr/uml/makecool.pl exampleModel.xmi -out kb
```

This results in the creation of a file called ***exampleModel.kbc*** (in the ***kb*** directory), which contains the COOL defclass definitions.

File generation from templates

Application specific startup scripts and property files typically contain a significant amount of content that does not require modification from general or default settings. Therefore, to simplify the creation of these files, a set of template files - files that contain required content with “place-holder” tokens for content needing specific values - are provided along with a Perl convenience script to process these template files into application specific files. This Perl convenience script is called ***makefile.pl*** and is provided in the ***<install dir>/scr*** directory. The script takes, as required input, the name of a template file and the name of an output file. If the template file is not found at the specified path then it will be looked for in the ***<install dir>/scr/tpl*** directory. Subsequent arguments define token/value (in the form of ***<token name>=<value>***) pairs that will be used to replace “place-holders” in the template file with the specified values. The following is an example use of this script:

```
    -> perl <install dir>/scr/makefile.pl jacorb.properties.tpl ↪
         jacorb.properties nsref=file:/home/user/NS_Ref
```

This results in the creation of a file called ***jacorb.properties*** in the current directory with the token “nsref” replaced with the value “file:/home/user/NS_Ref”.

Complete system generation

In addition to the scripts used for generation of specific system artifacts, a script is provided that will produce all artifacts required for a complete system. This script is called ***makeall.pl*** and is located in ***<install dir>/scr/uml***. This script requires setting the environment variable ***PROJECT_HOME*** to point to a location containing project sub-directories. Each project sub-directory must contain a sub-directory called “xmi.” The project domain model XMI file must be placed within this sub-directory. If, for example, a project called “example” is defined with a domain model XMI file called “exampleModel.xmi” then a sub-directory called “example” would be created in the directory pointed to by ***PROJECT_HOME***. The “xmi” directory within this “example” sub-directory would contain the “exampleModel.xmi” file. The following then would be an example use of the ***makeall.pl*** script:

```
    -> perl <install dir>/scr/uml/makeall.pl example
```

Artifacts produced by this script will be placed into the project sub-directory called “lib.”

Chapter 4 - Object Management Layer

4.1 Introduction

The Object Management Layer (OML) class library provides a decoupling presentation layer for accessing functionality to manage the life-cycle of objects. The layer provides functionality that wraps interaction with underlying information objects providing run-time discovery of object class structure and value type constraint. Internally, this class structure is utilized to find and invoke the specific methods required to access and modify object instances. Additionally, type information is obtained and utilized internally to constrain value input. From the client application perspective, interaction with object instances is reduced to interaction with an interface that uses simple strings with attribute value constraint and translation (to the specific, required data type) handled internally. The benefits of utilizing the OML interface (as opposed to direct object interaction) are most apparent when run-time input and output requirements dictate flexibility. Examples of applications requiring this flexibility are user interfaces, as well as interfaces to external information sources where, in both cases, information input is not strictly controlled. Additionally, the OML provides a presentation layer where information output can be more easily tailored. Again, this is helpful when applied to user interfaces and interaction with external information sources.

In addition to information object access and presentation, the OML provides functionality for the management of interests, which is implemented internally and exposed to using applications through the standard Java event model (listener registration with event method callbacks). In addition, support is provided for accessing multiple object information sources (domains) simultaneously and transparently.

An object-oriented representation of information incurs a requirement for managing objects and their associations. The OML is designed to simplify client application object management functionality through the use of run-time information structure (meta-data) discovery and value presentation.

The specific functionality provided by the OML library is outlined as follows:

- Run-time discovery of class meta-data
  - introspection through Java reflection
  - supplemental meta-data through class properties
- Value presentation and constrained input
  - string formatting and conversion based on value type
- Information discovery through ad-hoc query and interest notification
- Plugins for specialized management and information access requirements
  - array type management
  - association management
  - struct type management
  - numeric unit of measure conversion
- Server interface for accessing specialized information sources
  - multiple interface implementations for simultaneous access to various sources

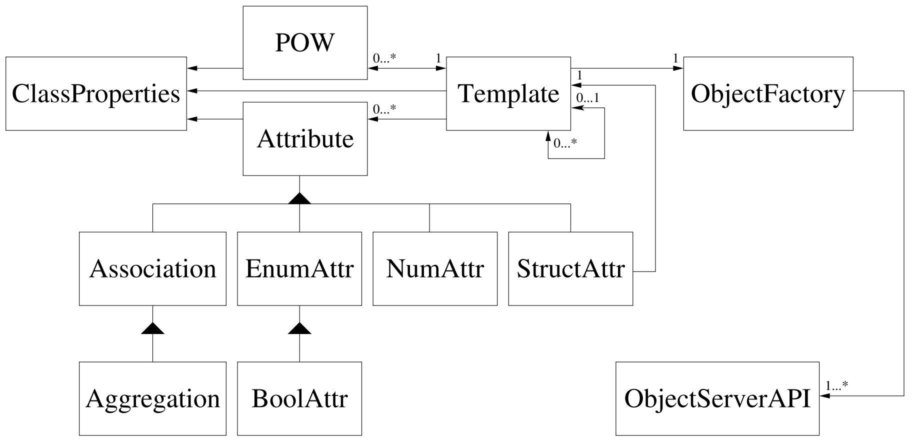


Figure 4.1: The Object Management Layer Classes.

4.2 Architecture Overview

The design of the OML centers around the ***Template***, Proxy Object Wrapper (***POW***), and ***Attribute*** classes. Underlying functionality is provided by the ***ObjectFactory*** and various plugin classes supporting specialized information source implementations. Figure 4.1 presents a class diagram showing these classes and their relationships.

4.2.1 Template

The ***Template*** class supports domain object class management by providing the following features:

- Class introspection through Java reflection and supplemental class properties
- Object construction and destruction
- Class based interest management

Object class introspection, provides the functionality required to expose object access and constrain value input and presentation through the use of Java reflection and supplementary information provided through ***ClassProperties*** (utility class that manages additional class information through properties). Instances of the ***Template*** class provide object class structural information and are utilized internally for the management of object instances. ***Template*** instances may also be viewed as producers of Proxy Object Wrappers (***POW***).

4.2.2 POW

The Proxy Object Wrapper (***POW***) class supports object instance management by providing the following features:

- Object creation and deletion
- Queued object interactions
- Constrained object instance interaction
- Attribute values passed as strings with type constraint enforced

- Instance based interest management

Utilizing ***Template*** instances, ***POW*** instances may be created to manage instances of specific classes defined by the ***Template***. Interaction with values, through ***POW*** instances will be constrained by ***Attribute*** instances managed by the ***Template***. Additionally, object value modifications are queued with proxy object method invocation initiated by a post operation. Queued values may be cleared (reset to previous state) without proxy object interaction.

4.2.3 Attribute

The ***Attribute*** class (and subclasses) provides attribute value management through use of the following features:

- Type introspection
- Set/get accessor discovery and invocation
- Data type to string and string to data type conversion
- Specialized plugin classes may be used to override provided classes

Attribute instances utilize ***ClassProperties*** (provided by ***Template***) to obtain type information for subsequent access and input validation. Specialized classes may be provided (as plugins) to handle specific access/constraint requirements. Some classes are provided for common requirements, and are outlined below:

- ***NumAttr*** - support for numeric attributes with optional unit of measure (provides unit of measure conversion functionality).
- ***EnumAttr*** - support for attributes constrained to a limited set of enumerated values.
- ***BoolAttr*** - extends ***EnumAttr*** for specialized boolean value constraint (constrained to two values).
- ***StructAttr*** - complex struct type support utilizing ***Template*** to manage struct fields.
- ***Association*** - simple association management (assumes management handled by object proxy/servant).
- ***Aggregation*** - extends ***Association*** to provide support for more constrained relationship (implied ownership).
- ***ManagedAssociation*** - extends ***Association*** by adding association management (required for simple object servant implementations that do not provide association management).
- ***ManagedAggregation*** - extends ***ManagedAssociation*** to provide support for more constrained relationship (implied ownership).

4.2.4 Object Factory

The ***ObjectFactory*** class provides an Application Programmer Interface (API) to specialized object server APIs (plugins). This class is implemented as a singleton (only one instance is created) to maintain object server APIs (associated to specialized object domains/models) and to provide methods for interaction with specific, appropriate object servers. Based on an information class' containment within a domain (or model), the specific object server API is selected by the ***ObjectFactory***.

4.2.5 Object Server API

The ***ObjectServerAPI*** is an interface that defines standard functionality that must be provided by implementing server API classes. The functions described by this interface include the following:

- Information object discovery (query)
- Information object retrieval (resolve)
- Information object destruction
- Interest management (addition/removal)

4.2.6 Class Properties

In addition to class reflection, information required to manage instances, their attributes and associations, is provided through the use of properties. The additional information provided includes:

- Class/Type meta-information
  - constructor parameter list (ordered attribute names as expected by full constructor)
  - name of attribute defined as object key (class only)
  - name of parent class (class only)
- Attribute meta-information
  - attribute type (must correspond to type with defined attribute manager plugin - see Appendix B.5)
  - default (or initial) value
  - allowed values
  - display values (corresponding to allowed values)
  - derived attribute? (attribute not included in constructor, but read accessible)
  - hidden attribute? (attribute not normally accessible)
  - unit(s) of measure (numeric only)
  - association name (association only)
  - associated class name (association only)
  - attribute mapping defined using subset of OCL (Object Constraint Language) (mapped attribute only)

4.3 Configuration

Client applications that utilize the OML must supply a properties file to tailor the OML for use with specific domains. This properties file is loaded as part of the OML initialization process and is specified through the ***com.cdmtech.core.properties*** system property (defined on the Java runtime command line using the ***-D*** option). The property file is defined by name only and will be loaded as a system resource, which is assumed to be located somewhere in the Java class search path (specified using the Java runtime command line option ***-cp***). The properties used by the OML are described in detail in Appendix B.5.

4.4 Using the Object Management Layer

The following sections provide an introduction to programmatic use of the OML. Details on available methods and their function can be found in the API documentation (JavaDoc).

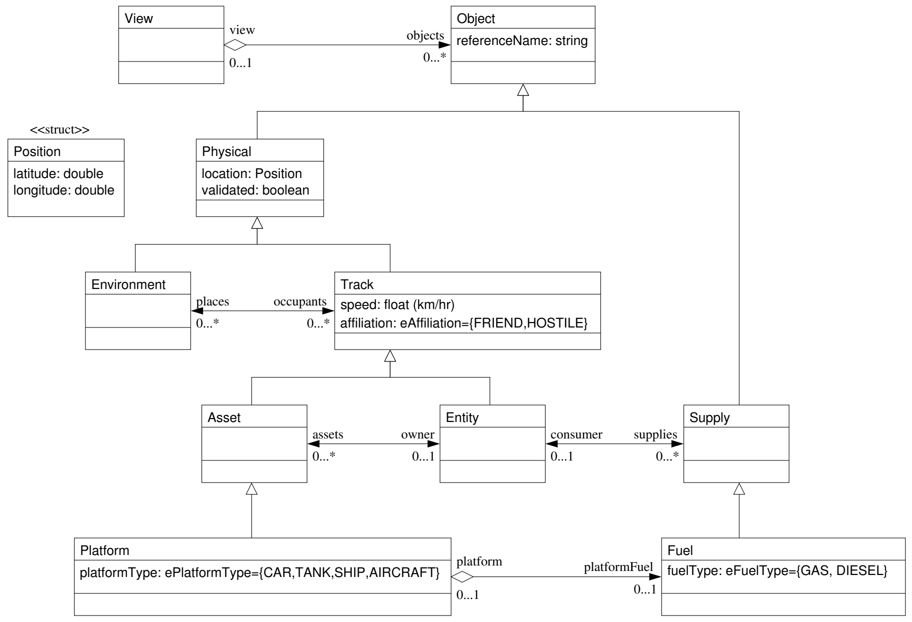


Figure 4.2: Example Object Model.

4.4.1 Object Interaction

The ***POW*** class adds generic functionality to the object model classes to aid in object manipulation. To illustrate, consider the example model in figure 4.2 and the following code statements:

```
POW myTank = Template.getTemplate("Platform").createObject();
myTank.set("referenceName", "my tank");
myTank.set("platformType", "TANK");
POW myFuel = Template.getTemplate("Fuel").createObject();
myFuel.set("referenceName", "my diesel fuel");
myFuel.set("fuelType", "DIESEL");
```

It should be noted that object creation and attribute modification transactions are queued locally and will not be reflected in the object server instance store until a call is made to the ***POW post*** method. However, calls to object ***delete*** are not queued and will result in immediate object deletion. For example, the method call ***myTank.post()*** results in the creation of the ***myTank*** object with all attribute values passed in as arguments to the ***Platform*** object constructor. Any subsequent calls to the ***POW*** instance set methods will result in calls to the proxy object set methods (with the next call to the ***post*** method).

The ***Template*** class implements functionality to support attribute constraints and validation. Additionally, it contains support for class constructor and access (set and get) method determination through runtime class reflection and properties. A ***Template*** instance is created for each class, as required, with each class represented through the defined hierarchy. The associated ***Attribute*** class and its subclasses provide constraints on attribute values. One of the benefits incurred through the use of the ***POW*** is the fact that all attribute values are entered and obtained as strings. The constraint on attribute values is handled internal to the ***Attribute*** classes. The benefit, from a user interface point of view, is that specialized attribute value management becomes unnecessary or at least highly simplified since only strings need be dealt with. As an example, consider the following code statements:

```
myFuel.set("fuelType", "DIESEL");
myTank.set("speed", speed.toStore("40 mi/hr"));
```

```
    myFuel.set("fuelType", "WATER");
    myTank.set("speed", "incredibly slow");
```

The first two statements result in successfully setting the indicated attribute values. The first sets the enumeration attribute ***fuelType*** to "DIESEL" which is a valid value contained in the enumerated value set (defined in the object model). The second sets the numerical attribute ***speed*** to 40 miles per hour and is internally converted to the store unit of measure (kilometers per hour) by the ***Attribute*** subclass ***NumAttr***. The third and fourth statements result in exceptions thrown, because neither are valid values for those particular attributes.

Association Interaction

The management of object associations is a particularly important aspect of any system requiring interaction with complex information representation. The OML framework allows for the inclusion of plugins to support specialized value management requirements. As implemented, associations are exposed through their object class roles, and, as such, are simply treated as specialized attributes. Several plugins are provided that implement, to varying degrees, the additional functionality required to manage associations (see section 4.2.3). To illustrate typical interaction with object association roles consider the following code statements:

```
    myTank.add("platformFuel", myFuel);
    myTank.remove("platformFuel", myFuel);
```

The first statement adds an object (***myFuel***) to the association whose role is ***platformFuel***. In this case, the object model defines this attribute as an aggregation. The ***myFuel*** object's role for this association (in this case ***platform***) is updated to now include a reference to the platform object (in this case myTank). Finally, the last statement removes this newly added association (object references are removed from both ends of the association).

It should be noted that object creation and attribute modification transactions are queued locally and will not be reflected in the object server instance store until a call is made to the ***POW post*** method (note, however, that calls to the object ***delete***, as well as any object modifications resulting from the deletion, are not queued). For example, the method call ***myTank.post()*** results in the creation of the ***myTank*** object with all attribute values passed in as arguments to the ***Tank*** object constructor. Any subsequent calls to the ***POW*** instance accessor methods will result in calls to the proxy object accessor methods (with the next call to the ***post*** method). The using class does not need to be concerned about these details since this object management behavior is provided by the OML classes.

Finally, the method call ***myTank.delete()*** results in the deletion of the ***myTank*** object. If the ***myTank*** object is associated to a fuel object (e.g., ***myFuel***) then the fuel object will also be deleted. This behavior is dictated by the stronger link implied by the aggregate relationship between the ***Platform*** and ***Fuel*** objects.

4.4.2 Object Query

The ***POW*** class provides methods for querying for references to objects that satisfy a particular object state pattern. This capability is accessed through temporary ***POW*** instances whose attribute and association role values are set to reflect the desired object pattern match criteria. Constraints may be set as values passed in a form expected by the attribute or association role datatype in which case the value constraint will be treated as an equality condition. Additionally, dependent on the attribute type, condition test operators may also be used. Condition operators are passed as symbols prepended to the input value. Note that value condition operators are only supported on single valued attributes and association ends whose datatype is either a primitive, string, or enumeration type. The supported test condition operators are the following:

= Equals

!= Not equals

The following operators are only supported for attribute value criteria where string values are tested lexicographically (see Java ***String.compareTo*** method) and enumeration values are tested positionally.

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

For example, to find all “hostile,” moving ***Platform*** objects that are owned by an ***Entity*** object called “enemy” the following query operations could be used:

```
Template entity = Template.getTemplate("Entity");
POW tmp = entity.getObject(null);
tmp.set("referenceName", "enemy");
POW enemy = entity.getObject(tmp.query()[0]);
Template platform = Template.getTemplate("Platform");
Attribute speed = platform.getAttr("speed");
tmp = platform.getObject(null);
tmp.set("owner", enemy);
tmp.set("affiliation", "HOSTILE");
tmp.set("speed", speed.toStore(">0"));
Object[] platforms = tmp.query();
```

The ***POW query*** method returns an array of object keys that may, in turn, be used to obtain ***POW*** instances through use of the ***Template getObject*** method as shown.

4.4.3 Object Interests

The ***POW*** and ***Template*** classes also include methods for managing both instance and class based interests. The implementation of these methods follows the design pattern specified by the Java event model. Specifically, instances of the ***POW*** and ***Template*** classes are event producers and contain methods defined for registering listeners (instances of classes that implement an appropriate listener interface). When the ***POW*** and/or ***Template*** instance fires an event, methods defined by the listener interface are invoked passing in the event as an argument. Subscriptions (interest criteria linked to client application) registered with the object server are managed internally through calls to these listener registration methods. For example, take a component that is interested in the creation and deletion of ***Track*** objects. This component would implement the ***ObjectListener*** interface and the methods ***objectCreated*** and ***objectDeleted***. The component would then register itself as a listener with the following code statements:

```
Template.getTemplate("Track").addObjectCreateListener(this);
Template.getTemplate("Track").addObjectDeleteListener(this);
```

Internally, these method calls will add subscriptions through the object server API, linking an interest in ***Track*** object creations and deletions to the client application. Subsequent creations and deletions of ***Track*** objects in the object server will result in a notification to the client with the firing of the appropriate event to the registered listener (i.e., the method ***objectCreated*** or ***objectDeleted*** will be called on the listener) passing in an event object. In this case, the event object defines methods that can be used to obtain the object identifier and class name for the object that has either been created or deleted.

4.4.4 Object Server Interfaces

A generic interface is provided in the OML framework to support client interaction with object servers. Each implementation of an object server interface may provide access to servers based on different architectures. The only requirements are that object interaction take place through client-side instance methods, client-side classes adhere to a prescribed pattern, and interest notification be event based. Object server interfaces are tied to unique domains (class namespaces).

Objects that are remotely serviced by an object server provide for a distributed, collaborative framework, however, the use of purely local objects (i.e., objects that are not maintained outside of the local client application environment) provides additional flexibility. Examples include objects whose characteristics are all derived (e.g., facades/views), objects that implement behavior alone (e.g., private agents), or client-side user-interface objects (i.e., objects that interact directly with client-side functionality). By providing an object server interface to local objects, interaction with these objects may take place through the same client interface (i.e., the OML). Additionally, the classes that model these objects may be defined and implemented utilizing the same tools provided in the TIRAC toolkit - supporting code and property generation.

Both the ***POW*** and ***Template*** classes make use of the ***ObjectFactory*** class, which provides the client access entry-point to the object server interfaces. With each domain associated with a single server interface, the ***ObjectFactory*** can determine which server interface to use through class identification within a domain. Therefore, interaction with objects and classes (through ***POW*** and ***Template*** instances) is handled transparently without any direct domain specification by the client application. The association of domains to object server interfaces are specified as properties (see Appendix B.5).

4.4.5 Attribute Value Management

The ***Attribute*** class and its subclasses (***Association***, ***Aggregation***, etc.) provide specialized management functionality for various attribute types. Additional management classes may be added by extending the appropriate ***Attribute*** subclass. These additional classes may be used to replace or add to existing management classes.

4.4.6 XML Import and Export

The OML also provides the capability to import and export instances to XML. The schema for the XML file can be generated from the domain model XMI file. The two classes that provide the import and export capability are ***XMLToPOWImport*** and ***POWToXMLExport***. To export instances to XML the following steps must be followed:

- XMLExportInterface exporter = AbstractXMLExport.getInstance(schemaFile);
  where schemaFile is the generated schema from the domain XMI.

- exporter.exportObjects(objects);
  where objects can be an array of POW's or objectKey values.

The current export capability supports exporting only the given list of objects. Associated objects are not exported automatically.

Optionally, only a subset of attributes can be exported as follows:

- exporter.exportObjects(objects, attrList)
  where attrList is a string array containing the attributes for the given object class to export and objects is an array of objects to export (either keys or pows).

Likewise, to import instances from a previously exported XML document:

- XMLImportInterface importer = AbstractXMLImport.getInstance(schemaFile);
  where schemaFile is the generated schema from the domain XMI.
- importer.importObjects(xmlDocument);
  where xmlDocument is the xml Document object.

The specific import or export classes to use can be specified in the following properties if using other than the default classes (see Appendix B.5).

```
<domain>.importClassName=<import class name>
<domain>.exportClassName=<export class name>
```

4.4.7 Example Code

Basic object operations

- Create objects.

  ```
  POW track = Template.getTemplate("Platform").createObject();
  POW fuel = Template.getTemplate("Fuel").createObject();
  POW timbuktu = Template.getTemplate("Environment").createObject();
  ```

- Set the ***referenceName*** attribute and post object (results in an instantiation). Note that in this example ***referenceName*** is a non-unique displayable name, the unique id was generated internally.

  ```
  timbuktu.set("referenceName", "timbuktu");
  timbuktu.post();
  ```

- Set the ***referenceName*** for the track object.

  ```
  track.set("referenceName", "track");
  ```

- Get the attribute manager for the ***speed*** attribute defined in the object model ***Track*** class. Note that there is no requirement that classes must have been defined in the same package.

  ```
  Attribute speed = Template.getTemplate("Track").getAttr("speed");
  ```

- Set more attribute values for the track object. Note that the ***location*** attribute is defined in the object model as using a complex datatype (i.e., struct), and may be set by either supplying a string of tab delimited values in the order defined by the model, setting individual fields, or using a ***Position*** instance.

  ```
  track.set("affiliation", "HOSTILE");
  track.set("validated", "TRUE");
  track.set("location", "35\t-121");
  track.set("location.latitude", "40");
  POW location = Template.getTemplate("Position").createObject();
  location.set("longitude", "-100")
  track.set("location", location);
  ```

- Set the ***speed*** attribute using the attribute manager to convert from a unit of measure to the expected internal storage format.

  ```
  track.set("speed", speed.toStore("100 mi/hr"));
  ```

- Link a place to the track object.

```
track.add("places", timbuktu);
```

- Add fuel (this time an aggregation as defined in the object model).

```
track.add("platformFuel", fuel);
```

- Post the object (results in an instantiation), print it, and then delete it. The deletion of the track object will also result in its fuel being deleted and its link to the place being removed.

```
track.post();
track.print();
track.delete();
```

- Query for some objects with constraints. In this example, find ***Platform*** objects contained in "testview" that are "HOSTILE" and moving faster than 50 miles per hour.

```
POW view = Template.getTemplate("View").getObject("testview");
POW tmp = Template.getTemplate("Platform").getObject(null);
tmp.set("view", view);
tmp.set("affiliation", "HOSTILE");
tmp.set("speed", speed.toStore(">50 mi/hr"));
Objects[] objs = tmp.query();
```

Listener registration for simple interests

- Register creation interest on ***Track*** class (notification will be sent if any object of this class is created).

```
ObjectListener listener = new MyObjectListener();
Template.getTemplate("Track").addObjectCreateListener(listener);
```

- Register modification interest on track object ***speed*** attribute (notification will be sent only if the attribute value is modified on this specific object).

```
ObjectModificationListener listener = new MyModificationListener();
track.addObjectModificationListener("speed", listener);
```

Listener registration for complex interests[1]

- Register a complex subscription.
  Note: This is supported only with the constraint-based subscription service (see chapter 5).

- Create interest on track object ***speed*** attribute with value greater than 50. Create the criteria using classes defined in the event constraint model (see figure 5.1). For object-based interests, use the actual ***objectKey*** instead of null as the second argument to the ***EventCriteria*** class.

```
EventCriteria criteria =
    new EventCriteria("Track", null, true, eEventType.MODIFY);
criteria.addConstraint(
   new FieldConstraint("speed", eOpType.GREATER_THAN,
                       new Float(50.0)));
```

- Register the interest. For object-based interests, register directly with the proxy object wrapper.

```
ObjectModificationListener listener = new MyModificationListener();
Template.getTemplate("Track").addListener(criteria, listener);
```

[1]Requires constraint-based subscription service and installation of the Java Expert System Shell (JESS), see Appendix A.5.

Chapter 5 - Subscription Service

5.1 Introduction

There are currently two Subscription Service implementations: the simple Subscription Service and the constraint-based Subscription Service[1]. Both implementations provide a publish and subscribe capability for domain object events. Subscriptions can be class-based (creation, modification, and deletion), object-based (modification and deletion) or attribute-based (modification). The constraint-based Subscription Service extends the basic capabilities of the simple Subscription Service by also allowing subscriptions that are constrained to very specific conditions, such as an attribute modification which is within a particular value range of interest.

As domain objects are created, modified, and/or deleted, corresponding object events are published with the Subscription Service by the domain object factories. In turn, the Subscription Service determines which subscribers, if any, need to be notified of the object event(s) and then notifies each subscriber.

5.2 Implementation

The Subscription Service is implemented as one of the core base services, with one Subscription Service per object model domain. Each object model domain can be configured to use a different Subscription Service implementation. However, all factories of a given object model domain will use the same Subscription Service. Likewise, clients interact with a specific Subscription Service when registering interests in objects of a particular object model domain.

Each Subscription Service has a single dispatch thread and multiple notifier threads, one per subscriber. As domain object events occur, the events are published by the object factories and queued by the Subscription Service. The Subscription Service dispatch thread then determines which subscriptions are satisfied by each published event and dispatches satisfied events to the event queue of the notifier threads for the relevant subscribers. The simple Subscription Service uses a set of subscription maps to efficiently determine which subscribers to notify. In the case of the constraint-based Subscription Service, object events are filtered through an inference engine to make this determination.

The subscriber's notifier thread then processes all queued events by sending notifications to the subscriber. On the client side, these notifications are queued as they are received until the client processes them.

If any event notification fails, the service will retry sending the notification based on the property ***EventRetry***. If a given event notification fails after the configured number of retires, the service will drop the event notification for the particular subscriber. If communication with a subscriber fails, the service will attempt to re-establish communication for a maximum number of retries as configured by property ***SubscriberRetry***. If the service determines that a subscriber no longer exists (e.g., a client application terminates without cleaning up its registered subscriptions), then service will clear all of the subscriber's subscriptions and free up any associated resources. See Appendix B.2 for more information.

5.2.1 Event and Constraint Models

The event model (shown in figure 5.1) illustrates the set of classes used to subscribe to and publish object events. The ***EventCriteria*** class represents a single subscription (or interest) by a single listener (or client). Instances of ***EventCriteria*** are created by a client and used to register an interest with the service in a particular type of object event.

[1] Requires installation of the Java Expert System Shell (JESS), see Appendix A.5.

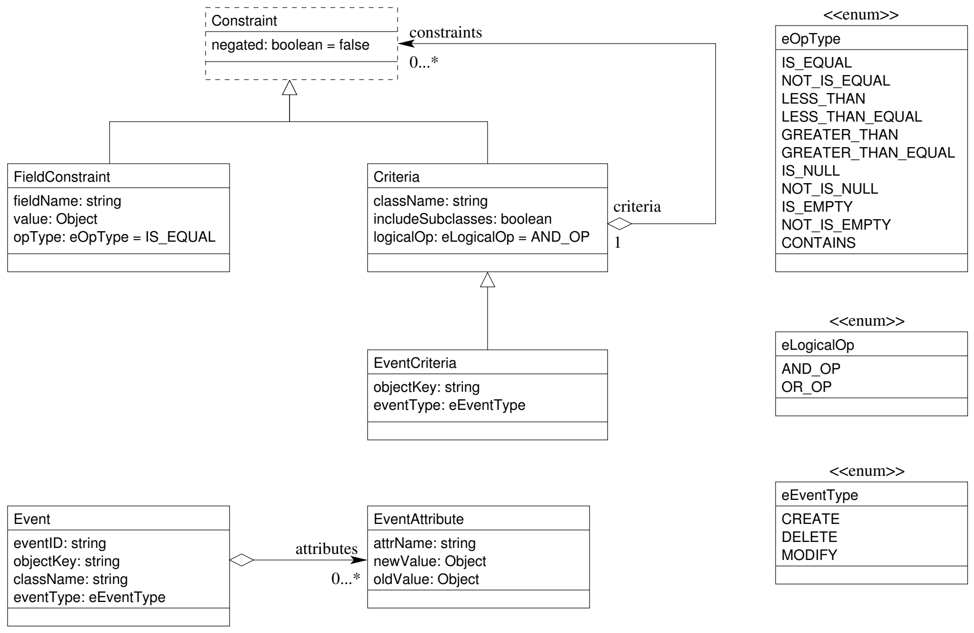


Figure 5.1: Event and Constraint Model.

Object model domain factories create instances of the ***Event*** class to represent a single object event. In the case of object modifications, the ***Event*** instance will also contain the ***EventAttribute*** instance(s), which contain the old and new values for the modified attribute(s). The ***Event*** instances are then published by a factory with its appropriate Subscription Service.

The Subscription Service determines which clients to notify of which published events by checking for satisfied ***EventCriteria***. Notifications sent by the service and received by the client include the ***EventCriteria*** and the ***Event***(s), which satisfied the criteria. If the received ***Event*** is a modification (***eventType*** = MODIFY), then the client can extract the old and new attribute values directly from the ***EventAttribute*** without having to interrogate the object itself (see section 5.2.3).

A common constraint model (shown in figure 5.1) was developed for registering subscriptions with the constraint-based Subscription Service as well as for performing queries with the Persistence Service. This constraint model, together with the event model, define all the classes necessary for registering subscriptions and publishing object events with the Subscription Service.

The simple Subscription Service makes uses of the ***FieldConstraint*** class to indicate modification interests in specific attributes. The constraint-based Subscription Service extends the simple Subscription Service, providing additional support for constraint-based subscriptions.

5.2.2 Specifying Attribute Constraints

The constraint-based Subscription Service supports specification of a single attribute constraint. This is done by adding one or more ***FieldConstraint*** instances to an ***EventCriteria*** instance.

For example, consider a ***Track*** class with an attribute ***speed***. An interest can be specified to receive all modification events for a ***Track*** object whose speed is greater than 50. The user would first create an ***EventCriteria*** instance with ***className*** = "Track" and ***eventType*** = MODIFY. Then a ***FieldConstraint*** instance with ***fieldName*** ="speed", ***opType*** = GREATER_THAN, and ***value*** = 50 would be added to the constraints of the ***EventCriteria*** instance. The user would then use this ***EventCriteria*** to register an interest in ***Track.speed*** modifications with a value over 50. All modifications to the ***speed*** attribute will be received as long as the speed is greater than 50. If the speed goes below 50, an event whose ***valid*** flag is set to false will be received. Further changes to the attribute value, as long as it is less than 50, will not be received.

The supported constraints for the various attribute types are as follows:

1. For numeric attributes the supported constraint ***opType***s are LESS_THAN, LESS_THAN_EQUAL, IS_-EQUAL, NOT_IS_EQUAL, GREATER_THAN, and GREATER_THAN_EQUAL.
2. For enumeration and boolean type attributes the supported constraint ***opType***s are IS_EQUAL, and NOT_-IS_EQUAL.
3. For associations and aggregations the supported constraint ***opType*** is CONTAINS.
4. For array type attributes the supported constraint ***opType***s are CONTAINS, IS_EMPTY, and NOT_IS_-EMPTY.
5. There is no support for specifying constraints on struct type attributes except general modifications.
6. For all other attributes such as strings etc., the supported constraint ***opType***s are IS_EQUAL, NOT_IS_-EQUAL, IS_NULL, NOT_IS_NULL, IS_EMPTY and NOT_IS_EMPTY.
7. General attribute modification interests can be registered by setting the ***opType*** to IS_UNDEFINED for a given attribute.

5.2.3 Old and New Values

As indicated previously, both implementations of the Subscription Service provide accurate old and new attribute value information. These values are made available via the ***EventAttributes*** associated with a particular object modification event. In the case of array type attributes (including associations), the old value will contain any values removed from the array while the new value will contain any values added to the array. Again, these values can be used directly as a performance enhancement, rather than having to interrogate the object itself.

5.2.4 Event Ordering

Events are normally received and dispatched in the same order that events are produced in the object model domain. This order can be modified by configuring the queue type for the Subscription Client (see property ***orderPolicy***, section B.2).

5.3 Configuration

5.3.1 Server Properties

Both the simple and constraint-based Subscription Servers can be configured to customize runtime behavior. Configuration is done via properties provided to the server during initialization (refer to Appendix B.3 for a complete list of properties).

5.3.2 Client Properties

Each client can define a set of properties that affect the runtime behavior of the Subscription Server for the particular client.

Note: These properties are specified in the client properties file and passed along to the Subscription Server (refer to Appendix B.2 for a complete list of properties).

Chapter 6 - Persistence Layer

6.1 Introduction

The Persistence Layer provides the capabilities for saving, restoring, updating, deleting, and querying for Java objects. The Persistence Layer is a general purpose utility and is completely independent of the core components. The Persistence Service, on the other hand, is dependent upon the Persistence Layer, and uses the Persistence Layer capabilities to save, restore, update and delete servant objects, and perform queries.

Additionally, there are archiving tools included with the Persistence Layer component. These tools allow for the creation and restoration of archive files with respect to a set of persistence domains. The archiving capability is described in the *Archiving Capability* section (*6.5*).

6.2 Architecture Overview

The central feature of the Persistence Layer object model (figure 6.1) is the Persistence interface. This interface declares the public API for all of the major capabilities of the Persistence Layer. There are currently two implementations of this interface: SerialPersistence and JDBCPersistence.

6.2.1 SerialPersistence

SerialPersistence is the default implementation of the Persistence interface provided by the Persistence Layer. This implementation uses serialized object files as its back-end object instance storage. Java objects are stored in a format independent of any particular object class version. They are instead stored as a set of DataTransfer objects and their associated Attribute objects (see figure 6.1). Thus, classes can be recompiled without losing the ability to restore previously stored Java object instances (assuming the “persistence signature” of the object class has not changed).

This implementation is low maintenance and requires minimal configuration (typically, no configuration is required.) The drawbacks with the SerialPersistence implementation include high memory utilization and slow query performance. These drawbacks become more pronounced as the number of object instances increases. This implementation is a good choice for average use patterns with a relatively small set of objects (approximately a few thousand). However, it is not recommended for use in production level systems where large instance sets will be used.

The primary limitation with SerialPersistence is that all persistence operations require objects to reside in memory. By default, SerialPersistence maintains an object cache of unlimited size and all objects are kept in memory. The more objects that are saved to persistence storage, the larger the cache grows and the more memory that is required by SerialPersistence. However, SerialPersistence can be configured with a maximum object cache size, which helps to reduce memory utilization at the expense of having to read objects back into memory from disk as needed. See Appendix B.4 for more information.

6.2.2 JDBCPersistence

JDBCPersistence is another implementation of the Persistence interface provided with the Persistence Layer. This implementation, as the name implies, uses a database via JDBC for back-end object instance storage. It can, in principle, use any relational database that has a valid JDBC 1.0 driver (JDBCPersistence only uses features of the JDBC 1.0 API.) It can also use ODBC data sources via a JDBC-ODBC bridge driver. JDBCPersistence has been

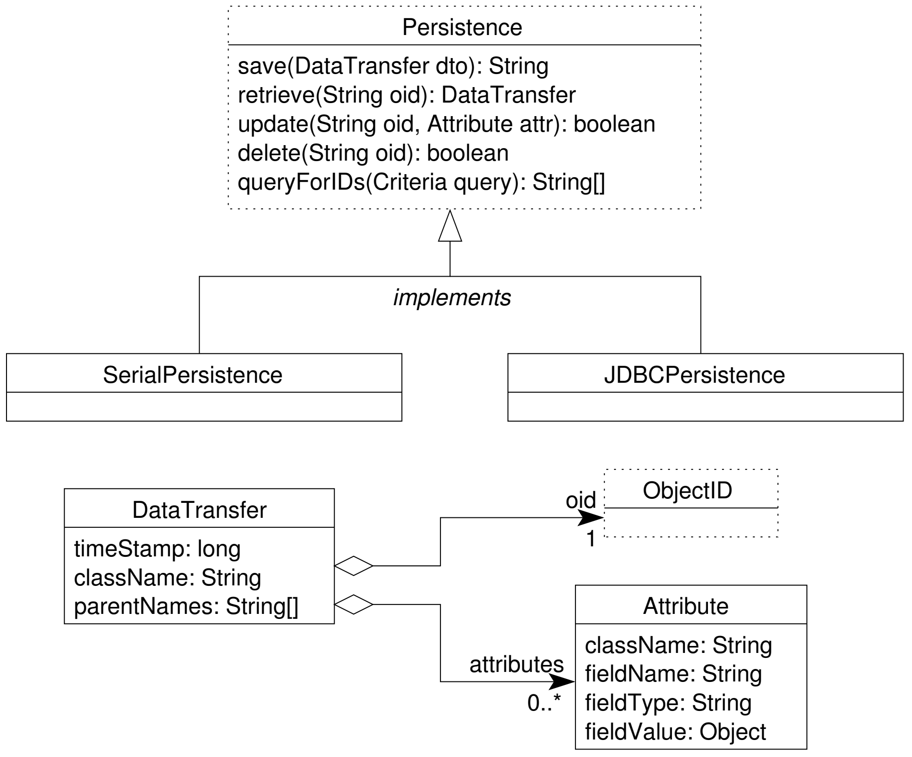


Figure 6.1: The Persistence Layer Classes.

tested with the MySQL database using the MySQL Connector/J JDBC driver, the Microsoft SQL Server database using the jTDS JDBC driver, and also with the Microsoft Access database using the JDBC-ODBC bridge driver distributed with the Sun Java Development Kit (JDK). Other databases should work, but have not been specifically tested for compatibility.

Unlike SerialPersistence, this implementation requires some configuration and database maintenance (see section 6.3). The advantage of using JDBCPersistence over SerialPersistence is the ability to work with very large instance sets (limited only by disk space where the database is installed) and a more efficient query capability. Both of these advantages are realized by JDBCPersistence primarily because objects are stored only in the database and are not kept in memory (as is the case with SerialPersistence.) The disadvantage with the current JDBCPersistence implementation is that, depending on the object model, database and configuration options used, it can be slower than SerialPersistence for certain operations (such as object deletions.)

6.3 Configuration

As mentioned above, there may be some configuration required in order to use the Persistence Layer in your operating environment. Configuration is done primarily through properties (see Appendix B.4). However, when using the JDBCPersistence implementation, some minimal database administration is required as well.

6.3.1 Database Administration

When using the JDBCPersistence implementation of the Persistence interface, there may be some database administration required. In particular, the Persistence Layer will **not** create a new database or ODBC data source. As a result, these must be established prior to using the Persistence Layer via JDBC. Since the mechanisms for database creation vary, database specific documentation should be consulted. It may also be necessary to configure the database and JDBC driver to implement particular security requirements. Refer to the appropriate database and JDBC driver documentation for security configuration details.

6.4 Using the Persistence Layer

The following are example use-case scenarios for using the Persistence Layer with the core base services. The first scenario is the simplest and illustrates the use of SerialPersistence. The second and third scenarios demonstrate the use of JDBCPersistence. All scenarios assume that the core_persist and core_server components have already been installed, and that the ***<domain>_server.properties*** have been generated for each project domain. Also, for the JDBC based use-cases, it is assumed that the desired database server has already been installed and configured.

6.4.1 Using SerialPersistence

You can optionally modify ***<domain>_server.properties*** (or just ***core_persist.properties*** in the case of a single domain) to change the storage location or behavior. However, the defaults should work in most situations. Start the Name Server and then the core base services and domain factories for each project domain using the generated ***StartServer_<domain>*** script(s). Typically, no additional configuration is required unless the persistence location or the object cache management behavior needs to change. Creation of objects can be verified by performing queries on the object set or by looking in the subdirectories of the top-level storage location directory.

6.4.2 Using JDBCPersistence

Using the JDBC based persistence implementation is slightly more involved than using the serial based persistence.

Using the MySQL Database and the MySQL Connector/J JDBC Driver

1. Copy the MySQL Connector/J jar file to the ***lib*** directory of the core installation folder or other desired location. The MySQL database and Connector/J JDBC driver are available for download from www.mysql.com.
2. Extend the classpath for each set of domain services. This can be accomplished by modifying the ***<domain>_server.properties*** file(s) or ***StartServer_<domain>*** script(s) for each project domain to include the MySQL Connector/J jar file in the classpath. See Appendix B.4 for details on which properties to modify.
3. Create a new MySQL database for each project domain. This is most easily accomplished as follows:
   - Start the ***mysql*** client application.
   - Enter the command ***CREATE DATABASE <database>;*** where ***<database>*** is the name of the database.
   - Configure the security permissions for this database if necessary (refer to the MySQL user manual).
   - Exit the ***mysql*** client application.
4. Modify ***<domain>_server.properties*** file(s) for each project domain as follows:
   - Set the persistence type equal to ***jdbc*** or set ***com.cdmtech.core.util.persist.jdbc.JDBCPersistence*** as the Persistence implementation class.
   - Set ***com.mysql.jdbc.Driver*** as the JDBC driver implementation class.

- Set ***jdbc:mysql://<hostname>/<database>*** as the JDBC URL where ***<hostname>*** is the host where the MySQL server is running (this can be set to ***localhost*** for the local machine) and ***<database>*** is the name of the database created in step 3. Depending on security requirements, a user and password may need to be added to the URL definition. Refer to the MySQL and MySQL Connector/J user manuals for more on security permissions configuration.
- Make sure that the domain services classpath is extended as described above (if done via properties)

Start the core base services and domain factories for all project domains using the ***StartServer_<domain>*** script(s), and the Persistence Layer should successfully initialize using the specified MySQL database(s). If any exceptions are thrown during initialization, it is probably due to an incorrect value for the URL property, or forgetting to include the JDBC driver jar file as a part of the extended classpath. Creation of objects can be verified by performing queries on the object set or by using the ***mysql*** client application and entering the commands ***USE <database>;*** followed by ***SELECT COUNT(*) FROM OBJECT_CLASS;***.

Using the Microsoft Access Database and the JDBC-ODBC Bridge Driver

The Microsoft Access database is nearly ubiquitous on the Microsoft Windows platform. If Microsoft Office is installed on your Windows machine, then you probably also have the Microsoft Access database. Configuration of the Persistence Layer to make use of a Microsoft Access database is actually easier than configuration for use with a MySQL database. However, establishing an ODBC data source is a bit more involved than simply creating a new database. The steps that follow assume that Access and ODBC has been installed on your Windows machine.

1. Create a new ODBC data source. This is typically done using the the ODBC Data Source Administrator as follows:
   - Open the ODBC Data Sources control applet found in the Control Panel.
   - Select the ***User DSN*** tab.
   - Select ***MS Access Database*** from the list of data sources.
   - Click the ***Add*** button.
   - Select ***Microsoft Access Driver (*.mdb)*** from the driver list.
   - Click the ***Finish*** button.
   - Enter a name and optionally a description for the new data source.
   - Click the ***Create*** button.
   - Provide a database name, select where to put the database and then click ***OK***. A new empty Access database should have been created.
   - Exit the ODBC Data Source Administrator.
2. Modify ***<domain>_server.properties*** for each project domain as follows:
   - Set the persistence type equal to ***jdbc*** or set ***com.cdmtech.core.util.persist.jdbc.JDBCPersistence*** as the persistence implementation class.
   - Set ***sun.jdbc.odbc.JdbcOdbcDriver*** as the JDBC driver implementation class.
   - Set ***jdbc:odbc:<ODBC data source>*** as the JDBC URL where ***<ODBC data source>*** is the name of the ODBC data source created in step 1.

Start the core base services and domain factories for all project domains using the ***StartServer_<domain>*** script(s), and the Persistence Layer should successfully initialize using the specified ODBC data source (which in turn is configured to use the new Access database.) If any exceptions are thrown during initialization, it is probably due to specifying an incorrect ODBC data source name in the URL property. Creation of objects can be verified by performing queries on the object set or by using the tools that come with the Access database.

6.5 Archiving Capability

There are two primary functions provided by the archiving capability (archiver): creation and restoration of archives. An archive is created by retrieving one or more sets of domain objects from persistence and saving them to an archive file. An archive is restored by retrieving one or more sets of domain objects from an archive file and saving them to persistence. The archiving tools available with the Persistence Layer component take the form of two stand-alone applications (one command line based and one GUI based) and support classes and interfaces. The main archiver API is located in the ***com.cdmtech.core.util.persist.archive*** package. The GUI specific archiver API is located in the ***com.cdmtech.core.util.persist.archive.gui*** package.

6.5.1 The Archive Abstraction

Both the create and restore archive capabilities operate over a set of persistence instances (one per domain) and require a physical archive file. Managing this relationship can become quite complex. The ***ArchiveAdapter*** class helps to simplify this complex archive abstraction. It does so by creating and maintaining a logical archive abstraction as an association between a physical archive file and a set of persistence instances. Each persistence instance is configured as required for a specific object model domain (as previously described). Persistence instances are added to or removed from the set managed by the ***ArchiveAdapter***, referenced by the domain name.

6.5.2 Configuration

The API of the archiving capability allows for flexible control over the use of the archiver tools. The archiving capability can be configured and reconfigured programatically during runtime to accommodate a variety of use cases. However, the most common use case may be to simply provide a set of non-changing configuration properties as part of the standard ***core.properties*** file. Ultimately, the purpose of configuring the archiver tool is to create and maintain the archive abstraction that is managed by the ArchiveAdapter.

Properties

The set of properties used by the archiver tools primarily define the set of domains and the persistence configuration for each domain (as described previously.) When using the ***PropertiesFactory***, ***PersistenceFactory*** or ***ArchiveFactory*** utility classes, the only difference is that each set of persistence properties is prefixed by the domain name. For example, for the domain "MyDomain," the persistence property ***core.persist.serial.location*** would become ***MyDomain.core.persist.serial.location***. Again, this change to the persistence property names is only required when using the ***Factory*** utility classes. This is the situation for the use case where the properties are included as part of the ***core.properties*** file.

6.5.3 Archive File Structure

The current archive file structure takes the form of a Java Archive (JAR) file. This representation makes it easy to include pertinent archive information as part of the archive file within the JAR's manifest. Use of JAR files provides a standard format which is widely accepted and will enable future enhancements such as secure signing.

The JAR file manifest contains attributes and entries specific to the archive. These attributes include the original archive file name, the creation timestamp, and the archive tag value (if provided at the time of archive creation.) There is also one entry per domain, which includes domain specific attributes. These attributes and entries are used during archive restoration to verify that a selected archive file is a valid archive.

6.5.4 Archive Creation

The first of the primary archiving tool capabilities is the creation of archives. This involves first creating and configuring an ***ArchiveAdapter*** for a desired set of domains, and then initiating archive creation. Archive creation is initiated on the ***ArchiveAdapter*** by calling ***createArchive***. When archive creation is initiated, if a file already exists with the name provided, then the existing file is backed up (renamed) first. This behavior is configurable via the ***ArchiveAdapter***. If a tag value was specified for the ***ArchiveAdapter***, then the new archive will be "tagged" with that value. If a tag value is not provided, then the new archive will be "un-tagged." Finally, the set of objects for each configured persistence domain is retrieved from persistence and saved to the specified archive (JAR) file. See section 6.5.3 for more details.

6.5.5 Archive Restoration

The second of the primary archiving tool capabilities is the restoration of archives. This also involves first creating and configuring an ***ArchiveAdapter*** for a desired set of domains, and then initiating archive restoration. Archive restoration is initiated on the ***ArchiveAdapter*** by calling ***restoreArchive***. The specified archive file is first verified as a valid archive. If a tag value was specified for the ***ArchiveAdapter***, then the archive must be tagged with the same tag value (case insensitive). If a tag value was not specified for the ***ArchiveAdapter***, then tag comparison is not performed.

After the archive is successfully verified and tags match (if specified), the archive will be restored. The set of objects for each configured persistence domain will be read from the archive (JAR) file and saved to persistence. If the archiver tool is configured for domains not contained in the archive, then those domains are ignored. Likewise, if the archive contains domains that are not currently configured for the archiver tool, then those domains will also be ignored. Only domains which are both configured for the archiver tool and are contained in the archive will be restored.

6.5.6 Archiver Application

The command line (non-GUI) stand-alone application is implemented by the ***Archiver*** class. This application accepts a number of options that affect the configuration of the archiving capability. With no command line options, the ***core.properties*** file must contain all of the configuration parameters and the default archive file is ***archive.jar***. In order to cause an archive to be created or restored, the ***-c*** or ***-r*** option must be provided, respectively. The command line options for the ***Archiver*** application are:

- **-f** *file name* - specify the archive file name to use for create or restore (for restore, the file must exist). If not specified, the default archive file name will be ***archive.jar***.
- **-p** *file name* - specify a properties file to load for archiver configuration in addition to any existing ***core.-properties***. If not set, only the current ***core.properties*** will be used for configuration.
- **-t** *tag value* - specify the archive tag value (optional). If not specified, the archive will be created "un-tagged". For restore, the archive's tag (if it exists) will not be used for verification.
- **-c** - indicate to create a new archive.
- **-r** - indicate to restore an existing archive.
- **-v** - indicates more detailed output during create or restore, and to output the archive report afterward (default is false).

6.5.7 ArchiverGUI Application

The GUI stand-alone application is implemented by the ***ArchiverGUI*** class. This application accepts a number of options which affect the configuration of the archiving capability. With no command line options, the ***core.properties*** file must contain all of the configuration parameters. The command line options for the ***ArchiverGUI*** application are:

- **-p** *file name* - specify a properties file to load for archiver configuration in addition to any existing ***core.-properties***. If not set, only the current ***core.properties*** will be used for configuration.
- **-f** *file name* - specify the initial archive file name. If not set, there will be no initial archive file name. It will have to be provided via the file chooser dialog.
- **-t** *tag value* - specify the initial archive tag value. If not set, there will be no initial archive tag value. It will have to be provided via the tag text field.
- **-w** - specify to use the windows look and feel. If not specified, then the default look and feel is used.

The ***ArchiverGUI*** application simply provides an application frame GUI component for the ***ArchiverPanel***. From the ***ArchiverPanel***, the user can set the archive tag value, choose to ***Open an archive...*** or ***Save an archive...*** and view archiver log messages.

Open Archive

When the user chooses to ***Open an archive...*** the ***Open Archive ArchiveFileChooser*** dialog is opened, allowing the user to select an existing archive file. If the user provided a tag value, then only those archive files with matching tags (case insensitive) will be viewable. The user can also choose to view all files in the current directory by changing the ***Files of type*** combo box setting to ***All Files***. After selecting an archive file, the user can choose to open (restore) the archive by clicking on the ***Open*** button. Alternatively, the user may cancel the ***Open*** operation by clicking on the ***Cancel*** button.

Save Archive

When the user chooses to ***Save an archive...*** the ***Save Archive ArchiveFileChooser*** dialog is opened, allowing the user to select an existing file or enter a new file name. After providing an archive file name, the user can choose to save (create) the archive by clicking on the ***Save*** button. Alternatively, the user may cancel the ***Save*** operation by clicking on the ***Cancel*** button.

6.5.8 Configuration Utility

A configuration utility is included with the archiving tools to assist in the creation of the archiver properties and start-up scripts for the archiver applications. This utility is in the form of a Perl script named ***archiver_config.pl***. It is located in the ***scr/archiver*** directory of the core installation folder.

The parameters to this script are a project name followed by one or more domain names. For example, if configuring the archiver on a Windows machine for a project named “MyProject” with domains “MyDomain1”, “MyDomain2,” and “MyDomain3” the configuration utility would be invoked as:

<installation directory>\scr\archiver\archiver_config.pl MyProject MyDomain1 MyDomain2 MyDomain3

This will produce three files in the current working directory: ***MyProject_archiver.properties***, ***MyProject_StartArchiver.bat***, and ***MyProject_StartArchiverGUI.bat***. The default properties generated will configure the archiver tools to use serial persistence for each project domain. The properties file may need to be modified if the default properties are not appropriate for your project.

Chapter 7 - JESS Agent Engine

7.1 Introduction

The JESS[1, 2] based agent engine[1] provides a complete environment to start and manage a JESS based agent session. It provides an automated mechanism for representation and management of information within the JESS environment reflective of the underlying information object state. This chapter describes, in detail, how to install, configure, and run a JESS based agent engine.

7.2 Architecture Overview

The JESS based agent engine is comprised of several key pieces that provide the required functionality:

- an **Agent Session Manager** to manage the overall agent session. It manages the tasks relating to starting, pausing, resuming, stopping and ending the agent session.
- an **Agent Manager** to initialize agents and process any agent activations.
- a **SemanticNet Manager** to manage object based events, mainly involved with mapping events to/from the inference engine.

The Agent Session Manager manages the overall working of the agent session. During each cycle of operation it allows both the Agent Manager and SemanticNet Manager to process any agent based and object based events, respectively. The SemanticNet Manager establishes and manages subscriptions for a given agent session. It queues and processes any object events and maps them to the JESS inference engine. The Agent Manager cycles through all the agent modules and processes any agent activations. It queues and processes any JESS events and maps them to equivalent object events.

7.3 Configuration

The following sections outline the steps that should be followed to setup and run the JESS based agent engine. Refer to Appendix B.6 for specific definitions of the properties used to configure an agent session.

7.3.1 Generate Batch and Property Files

Template files can be used along with a provided Perl script (***makefile.pl***) to generate the batch and properties files needed to run a JESS agent engine for a given project. Two batch files - one to run the agent engine in INITIALIZE mode and the other in NORMAL mode and one properties file must be generated.

- Generate a startup batch file to start the agent engine in INITIALIZE and NORMAL mode.

[1]Requires installation of the Java Expert System Shell (JESS), see Appendix A.11.

```
<install dir>/scr/makefile.pl↪
   JessAgentSessionInit.bat.tpl JessAgentSessionInit.bat↪
   project=projectName domain=<comma delimited list of domains>
<install dir>/scr/makefile.pl↪
   JessAgentSession.bat.tpl JessAgentSession.bat↪
   project=projectName domain=<comma-delimited list of domains>
```

- Generate a properties file using the template properties file that is used by the agent engine.

```
<install dir>/scr/makefile.pl↪
   JessAgentSession.properties.tpl ./lib/JessAgentSession.properties↪
   project=<projectName> outputDirectory=<directory>↪
   agentsLoadFile=<your agent load file>↪
   agentInterestsFileName=<interests file to generate interests>
```

- The specified command line properties are key properties that are required to start an agent engine. Other optional properties, such as Agent class name, Container class name, and so on can also be specified in the properties file. View the generated properties file to set some of these optional properties.

7.3.2 Agent Session Modes

The JESS based agent engine can be started in three possible modes, namely INITIALIZE, NORMAL, and DEBUG.

- INITIALIZE Mode: In this mode of operation the agent engine pre-processes all the agent rules to generate an ***interests*** and an ***attributes*** file. The ***interests*** file is generated by processing all the LHS patterns for the agent rules defined in the load file. The ***attributes*** file is generated by processing all ***defqueries*** and ***defrules***.
- NORMAL Mode: This is the mode that should commonly be used to start the JESS based agent engine. If the ***interests*** and ***attributes*** files previously generated are specified in the agent engine properties file, they are used during the initialization process.
- DEBUG Mode: In this mode, the JESS agent engine generates a ***JessAEDebug.xls*** file containing information, such as all the facts asserted into the system, the rules that were activated, and/or fired and the facts that caused the activation/firing.

For better performance and memory utilization, the JESS based agent engine should first be run in the INITIALIZE mode to generate the ***interests*** and ***attributes*** profile. Generating the ***attributes*** profile limits the JESS ***deftemplates*** to only contain the minimum class definitions that are necessary for the agent rules. This not only lowers the memory footprint but also reduces the number of network calls that are necessary during initialization and whenever a new object is created.

INITIALIZE Mode

The JESS based Agent Engine must first run in the INITIALIZE mode in order to generate agent engine specific data to improve performance. Each time changes are made to the agent rules, this step must be executed to ensure that the data files generated in this step are current. To generate the JESS based agent engine specific data files the following steps must be followed:

1. Create a JESS load file (e.g., ***agents.load***) and load all the “kb” files defining agent logic in this file.

2. Ensure the following properties are set in the agent engine properties file generated in a previous step:

   a) ***core.client.agentLoadFile***=<agent load file name created in step 1>

   b) ***core.client.interestsFile***=<interests file name (gets generated in this step)>

   c) ***AgentSession.includeAttributeFileName***=<attributes file name (gets generated in this step)>

   d) ***AgentSession.outputDirectory***=<directory name (where the two output files above are saved)>

3. Use the generated startup batch file for running the agent engine in the INITIALIZE mode.

4. This should result in the creation of two files - the ***interests*** file and ***attributes*** file specified in steps (b) and (c) in the directory specified in step (d).

NORMAL Mode

The same property file used in the previous section can be used to run the agent engine in NORMAL mode. This mode should be set when running an agent session (agents loaded and active). Verify that the following property is set:

- ***AgentSession.currentMode****=NORMAL*

7.3.3 Subscriptions

The JESS based agent engine relies on event notification to map information to JESS instances. There are two possible ways in which interests can be registered: dynamically by processing agent rules at runtime or statically through the use of an ***interests*** file. For dynamic subscriptions, as new constructs are added and removed, the Subscription Manager retrieves class and attribute information by parsing the left hand side patterns and registers (or unregisters) interests on behalf of the agent engine. For static interests, an ***interests*** file for a given set of agent rules, can be generated by running the JESS agent engine in INITIALIZE mode. In this mode, all agent rules are pre-processed to generate an ***interests*** file. During normal operation of the agent Engine, the ***interests*** file is used to initialize subscriptions. The following properties are used to affect the subscription behavior of the agent engine:

- ***core.client.interestsFile***=<agent interests file>
- ***AgentSession.outputDirectory***=<location of interests file>
- ***AgentSession.useDynamicSubscription***=true | false

7.4 Writing JESS Agents

7.4.1 Information Representation

The JESS based agent engine adheres to certain conventions to represent the information within the inference engine:

- For each domain class (including subclasses) used by the agent rules an equivalent JESS ***defclass*** definition is generated.
- Each domain class definition further defines all fields utilized by the agent rules using the following convention:

  - All simple attributes are represented as a SLOT, with the data type set to the equivalent JESS type such as RU.INTEGER, RU.STRING, RU.FLOAT and RU.LONG. All numeric types of short and byte are assumed to be of type RU.INTEGER. All numeric types of double are assumed to be of type RU.FLOAT.
  - All single-valued enumeration type attributes are represented as SLOT, with data type set to JESS type RU.ATOM.
  - All single-valued struct type attributes are represented as SLOT, with data type set to JESS type RU.-STRING.
  - All associations with a multiplicity of one are represented as SLOT, with data type set to JESS type RU.STRING whereas all associations with multiplicity greater than one are represented as MULTI-SLOT, with data type set to JESS type RU.LIST.
  - All array type attributes are represented as MULTISLOT, with data type set to JESS type RU.LIST. In addition, the specific list value defines the data type of individual elements in that list.

- Each domain class further defines an additional slot called "OBJECT," which contains a reference to the POW (Proxy Wrapper Object, see chapter 4) object, a slot called "class," which contains the actual (truncated - without the prefix ***com.cdmtech.core.client.corba***) class name for the java object, a slot called "name," which is a unique name (normally the ***objectKey*** for all domain classes or a unique identifier for all struct classes)

- Each struct class further defines an additional slot called "_owner," which holds the unique identifier of the owning instance.

Attributes file

The JESS agent engine simplifies the generation of ***defclass*** definitions further by making use of an ***attributes*** file (if one is available), which contains only a subset of domain classes (and subclasses) and attributes that are of importance to the agent logic. This not only reduces the memory footprint but also reduces time spent in initializing and updating ***definstances*** (and/or facts). An ***attributes*** file can be generated automatically by first running the agent engine in the INITIALIZE mode. In this mode of operation, agent logic is pre-processed to generate an ***attributes*** file containing all classes and attributes being used within the agent rules. This file is later used in NORMAL mode, to generate ***defclass*** definitions. The format for the generated ***attributes*** file is similar to the ***interests*** file above. Every attribute in a given class that is being used, is written on a single line as follows:

className:attributeName

where ***className*** is the domain class name and ***attributeName*** is the attribute that is being used.

The following properties must be specified to generate and use an attributes file:

AgentSession.usePartialTemplates=*true*

AgentSession.includeAttributeFileName=*<attributes file name>*

AgentSession.outputDirectory=*<directory to save (or read) the attributes file (optional property)>*

7.4.2 Information Management

The JESS agent engine manages the filtering of the pertinent information that is processed by the inference engine by establishing clearly defined subscriptions based on the agent logic. The specific subscriptions for a given agent session can be provided via an ***interests*** file. This file like the ***attributes*** file described in the previous section, can be automatically generated by running the JESS agent engine in INITIALIZE mode. The next subsection provides a brief description of the format for this file.

Description of interests file

The agent engine uses an ***interests*** file (if one is available) to initialize subscriptions for a given agent session. This limits the information that is processed by the underlying inference engine to only that which is of importance to the agent rules. An ***interests*** file can be automatically generated by first running the agent engine in INITIALIZE mode. The following example ***interests*** file using the sample object model (see chapter 4, figure 4.2) is only provided as a sample. A single subscription must be specified on a line. For each class specified creation and deletion interests are registered. Attribute modification subscriptions are specified by the classname followed by the attribute name delimited by the ":".

CLASS:Physical

CLASS:Environment

CLASS:View

Physical:location

Track:affiliation

Object:referenceName

View:objects

Platform:platformType

Platform:platformFuel

7.4.3 JESS User functions

This section provides an overview of the JESS functions that are included with the JESS agent engine.

Interaction with definstances

The JESS based agent engine provides convenience user functions to assist in the assertion, deletion, and modification of JESS ***definstances***. The use of these functions is mandatory, owing to the fact that the TIRAC toolkit does not follow the standard "java bean" component model. All arguments enclosed in square braces are optional and $^+$ is used for one or more argument values.

- *make-instance*

  To create a JESS ***definstance*** for a given class, the following user function must be used. It servers the dual purpose of not only persisting an instance of the given Java class, setting all the given attribute values via the proxy object wrapper (POW), but also asserting the equivalent JESS ***definstance*** to the knowledge base
  Usage:
  (*make-instance* <*className*> [(<*attributeName*> <*attributeValue*>)]$^+$)
  Examples:
  (bind ?tank (*make-instance* Track (speed 50.0) (referenceName "tank")))
  (*make-instance* Entity (referenceName "blue unit") (asset ?tank))
  The function supports the usage of most slot value types and bound variables at this time. Once a fact is bound, as shown in the first example, it can be modified at a later time on the JESS command line or on the right hand side of a rule by using the "modify" userfunction included in the original JESS distribution. When a new ***definstance*** is asserted, all associated struct values are also asserted as associated ***definstances*** in JESS.

- *delete-instance*

  Retracts the ***definstance*** and clears the associated java object from the object instance store and cleans up all composite facts (i.e. structs) :
  Usage:
  (*delete-instance <FACT>*)

- *get*

  Retrieves the value for an attribute <*attrName*> directly from the java instance <*POW*>
  Usage:
  (get <*POW*> <attrName>)

- *set*

  Sets the value for an attribute <*attrName*> of a java instance <*POW*> to <*attrValue*>
  Usage:
  (set <*POW*> <*attrName*> <*attrValue*>)

- *send*

  Usage:
  (send <*FACT*> <*functionName*>)
  where <*functionName*> can be one of *print*, *delete*, *update-slot*, *get-*, *add-assoc*, *remove-assoc*
  Examples:
  (*send <FACT> print*)
  Pretty print the <*FACT*> using print-fact
  (*send <FACT> delete*)
  Deletes the <*FACT*> using *delete-instance* user function.
  (*send <FACT> update-slot <slotName> <slotValue>*)
  Updates the slot using modify ***deffunction***
  (*send <FACT> get-slotName*)
  Returns the slot value for the given <*slotName*> for the given <*FACT*>.
  (*send <FACT> add-assoc* <slotName> <*FACT*>)
  Adds the second <*FACT*> to the first <*FACT*> association role name suing *add-assoc* ***deffunction***.
  (*send <FACT> remove-assoc <assocName> <FACT>*)
  Removes the second <*FACT*> from the first <*FACT*> association role name given by <*assocName*> using *remove-assoc* ***deffunction***.

Utility functions

The following utility functions are automatically included and available for use:

- *class-instances*

  Prints a list of all the java objects (i.e., <*POW*> proxy object wrapper objects) existing locally.

  Usage:
  (*class-instances <className>*)

  where *<className>* is an existing class name in the object model.

- *instances*

  Returns a list of all java objects (i.e., <*POW*> proxy object wrapper objects). Optionally, if a list of class names is provided as argument, only objects belonging to those classes is returned.

  Usage:
  (*instances* [<*class name*>$^+$])

  (*instances*)

(*instances* Track)

(*instances* (*create*$ Track Asset))

- *save-snor*

  Writes the given <*FACT*> to <*fileName*> in SNOR format. Use *restore-snor* to restore the <*FACT*> in future.

  Usage:
  (*save-snor* <*FACT*> <*fileName*>)

- *restore-snor*

  Restores any objects stored in the given SNOR <*fileName*> and asserts ***definstances*** for each object that is restored.

  Usage:
  (*restore-snor* <*fileName*>)

- *print*

  Pretty prints the fact whose <*fact-id*> is provided as a two column list of slot name and value pairs. If a list of <*slotName*> (s) is provided then only those slot name/value pairs are printed

  Usage:
  (*print* <*fact-id*> [<*slot name*>$^+$])
  (*print* 2)

  (*print* 2 ?slotName)

  (*print* 2 $?slotNames)

- *print-fact*

  Pretty prints the given <*FACT*> as a two column list comprising of slot name and value pairs. If a list of slot names is provided then only those slot name/value pairs are printed

  Usage:
  (*print-fact* <*FACT*> [<*slot name*>$^+$])
  (*print-fact* <*FACT*>)

  (*print-fact* <*FACT*> ?slotName)

  (*print-fact* <*FACT*> $?slotNames)

- *add-assoc*

  Associates then second <*FACT*> to the first <*FACT*> association role <*slotName*>.

  Usage:
  (add-assoc <*FACT*> <*slotName*> <*FACT*>)

- *remove-assoc*

  Dis-associates the second <*FACT*> from the first <*FACT*> association role <*slotName*>

  Usage:
  (*remove-assoc* <*FACT*> <*slotName*> <*FACT*>)

- *add-value*

  Usage:
  (*add-value* <*FACT*> <*slotName*> <*newValue*>)

  Adds the <*newValue*> to the given <*FACT*> attribute array list represented by the <*slotName*> if it is not already present in the array list.

- *remove-value*

  Usage:
  (*remove-value* <*FACT*> <*slotName*> <*oldValue*>)

  Removes the given <*oldValue*> from the slot attribute <*slotName*> array list, if it exists, from the give <*FACT*>.

- *add-struct*

  Usage:
  (*add-struct* <*FACT*> <*slotName*> <*FACT*>)

  Adds the second struct <*FACT*> to the first owner <FACT> struct role represented by <*slotName*> whose multiplicity is greater than 1.

- *remove-struct*

  Usage:
  (*remove-struct* <*FACT*> <*slotName*> <*FACT*>)

  Removes the second struct <*FACT*> from the first owner <FACT> struct role represented by <*slotName*>, whose multiplicity is greater than 1.

- *update-struct*

  Usage:
  (*add-struct* <*FACT*> <*slotName*> <*structFieldName*> <*structFieldValue*>)

  Updates the given <*structFieldName*> slot for the corresponding struct fact whose struct role is specified by <*slotName*> (multiplicity 1), for the owner fact <*FACT*> to the new value, <*structFieldValue*>.

- *get-slot-value*

  Returns the current slot value for the attribute <*slotName*> from the java object <*POW*>. This function can be used to retrieve the current slot value, especially if the fact slot value has not been updated, which is quite possible if no interests have been registered for the particular class and/or attribute modifications.
  (*get-slot-value* <*POW*> <*slotName*>)

- *fact-for-key*

  Returns the <*FACT*> for the given <*objectKey*> or nil, if it cannot be found. Now uses the ***defquery*** defined in CoreUtil.kbf file, which needs to be included in the load file. Instead use the ***deffunction*** *get-fact-for-key* defined in the file CoreUtil.kbf.

  Usage:
  (*fact-for-key* <*objectKey*>)

- *isKind*

  Returns true if the <*childClass*> is a sub-class of the <*parentClass*>.

  Usage:
  (*isKind* <*parentClass*> <*childClass*>)

- *get-facts*

  Returns the list of facts in the knowledge base. Optionally a list of class names can be provided as arguments in which case, only returns facts for those classes.

  Usage:
  (*get-facts* [<*className*>$^+$])

  (*get-facts* Track)

  (*get-facts* (*create*$ Track Asset))

- *is-assoc*

  Returns true if the given <*slotName*> is an association role name in the given <*className*>.

  Usage:
  (*is-assoc* <*className*> <*slotName*>)

- *is-writable*

  Returns true if the given <*slotName*> has write access in the given <*className*>.

  Usage:
  (*is-writable* <*className*> <*slotName*>)

- *get-struct-for-name*

  Returns a struct fact for the given <*uniqueName*> it if it can be found or nil otherwise.
  (*get-struct-for-name* <*uniqueName*>)

- *get-structs-for-owner*

  Returns a list of struct facts for the given owner unique <*objectKey*> if any can be found or an empty list otherwise.
  (*get-structs-for-owner* <*objectKey*>)

Optional User Functions

Optional math and geo-spatial user functions available are defined in two different packages:

- Math functions such as sin, cos etc. are defined in com.cdmtech.core.client.aml.jess.javaFunctions.JessMathFunctions.
- GeoNavigational functions (e.g., distance between two points etc.) are defined in com.cdmtech.core.client.aml.jess.javaFunctions.GeoNavigationFunctions.

To load these functions into JESS, add the following lines to your load file.

(load-package com.cdmtech.core.client.aml.Jess.javaFunctions.JessMathFunctions)

(load-package com.cdmtech.core.client.aml.Jess.javaFunctions.GeoNavigationFunctions)

7.5 Running an Agent Session

Follow the steps enumerated in section 7.3 to generate batch and properties files. Then follow the steps below to start a JESS based agent session:

- Create the following directory ***PROJECT_HOME/kb/jess*** and put your kb files in this directory where all project specific files reside in ***PROJECT_HOME***.
- Create an ***agents.load*** file in this directory and include your “kb” files and other JESS commands for initializing the JESS environment in this file. To load files without providing an absolute path, place files in a directory called “jess” and include the path up to but not including the “jess” directory in the classpath being used to start the JESS based agent engine.
- Edit your agent engine properties file to set the following property (if not already done).

  core.client.agentLoadFile=*<agent load file name>*

- Generate an ***interests*** and ***attributes*** file by running the agent engine in the INITIALIZE mode. Make sure the following properties are specified in your agent engine properties file.

  core.client.interestsFile=*<interests file name>*

  AgentSession.includeAttributeFileName=*<attributes file name>*

  AgentSession.outputDirectory=*<directory where these files are to be generated>*

  AgentSession.useDynamicSubscription=*false*

  AgentSession.usePartialTemplates=*true*

- Start the naming server, base services, and domain factories before starting the JESS agent engine.

  – Use the generated batch file ***JessAgentSession.bat*** or ***InteractiveJessAgentSession.bat*** to start the agent engine in NORMAL mode. The ***InteractiveJessAgentSession.bat*** file starts the agent engine in interactive mode allowing runtime interaction with the inference engine.

- If necessary an optional container can be used to attach the agent session to a particular container. If using a container object, edit the following properties in the agent session properties file:

  core.client.containerClassName=*<container class name>*

  core.client.containerObjectName=*<specific container object name>*

  core.client.collectionRoleName=*< collectionRoleName >*

  The container object name need not be the ***objectKey*** but can be a display value. In which case, the following class property must be defined for the container class defined in the above property:

  <containerClassName>.class.disAttrName=*<valid display attribute name>*

  For example:

  Container.class.disAttrName=*containerName*
  The agent engine provides additional management if the above properties are specified. Only those external objects associated to the specified ***Container*** object are loaded into the inference engine. All objects created by the agent engine are automatically associated with the specified ***Container***. Further, if either ***core.client.removeCollectablesAtStartup*** or ***core.client.removeCollectablesAtShutdown*** are set to “true” then the agent engine performs cleanup operations at startup and shutdown.

7.6 Debug Utility

In order to debug agent rules, a simple debug utility is available. All events such as object creations, modifications, and deletions that were propagated to the inference engine, all fact assertions, retractions, modifications, and what objects triggered what rules can be recorded to an output file. To enable recording of these events, the following property must be set to true:

AgentSession.debugMode=*true*

Data is recorded in a tab-delimited format and can be viewed easily using any spreadsheet software such as Excel.

The default file name to which data is recorded is ***JessAEDebug.xls***. This can be changed by setting yet another property:

AgentSession.debugLogFile=*<filename>*

The values that are tabulated are: LogType, Index, ModuleName, RuleName, ClassName, ObjectKey, Attribute Name, New Value, Old Value and Remarks.

- LogType - the logType is a descriptive string that describes the event. For JESS events it can be: FACT ASSERTED, FACT RETRACTED, FACT MODIFIED, DEFINSTANCE ASSERTED, DEFINSTANCE RETRACTED, DEFINSTANCE MODFIED, ACTIVATION, DEACTIVATION and FIRED. For Java based events it can: CREATION, DELETION and MODIFICATION.

- Index – denotes the number of logs for the given LogType. Usually this is 0. But for “ACTIVATION”, “DEACTIVATION” etc multiple logs are recorded. For example for “ACTIVATION” log, all matches for the LHS patterns are also logged.
- ModuleName – the module name.
- RuleName – the Defrule name if any.
- ClassName – the Class name for the Fact or Java Object being recorded.
- ObjectKey – the unique object key if any for the Fact or Java Object.
- AttrName – the attribute name for modification type events.
- NewValue – the new value for the attribute for modification type events.
- OldValue – the old value for the attribute if any for modification type events.
- Remarks – Not currently used.

For each *Activation*, *Deactivation*, and *Rule Firing*, all the facts that match the corresponding action are logged to the file, with their current values.

7.7 Examples

7.7.1 Attribute Types

The following examples provide some insight on handling the various simple and complex attributes such as structs, enumerations, associations (and aggregations), and arrays. All examples are written using the example domain model in Figure 4.2.

1. Simple Attributes - comprise of string and numeric attribute types. All string and character types are defined as JESS type RU.STRING, boolean as RU.ATOM and numeric types are defined as the JESS equivalent type - all short, byte, and integer as RU.INTEGER, all float and double as RU.FLOAT, and long as RU.LONG.

   (a) Numeric - to test a Track’s speed is greater than 50
   (Track (speed ?speed &: (> ?speed 50.))

   (b) Boolean - to test for validated attribute being set to TRUE
   (Physical (validated TRUE))

   (c) String - to test that referenceName is not equal to “blue”
   (Object (referenceName ?name ~”blue”))

2. Enumeration - single valued enumeration types are defined as JESS RU.ATOM type. To pattern match (or define a condition) on particular affiliation type on the Track class:

   (a) Single-valued enumeration
   (Track (affiliation FRIEND | NEUTRAL))

   (b) Multi-valued enumeration
   (TestClass (enums $?enums &: (member$ FRIEND $?enums))

3. Struct - single valued struct types are handled similar to association (and aggregation) types. All composite classes are represented as ***defclasses*** and an “owner” slot defines the owning instance. In addition the owning instance contains a reference (a unique identifier) for each composite relation. In the case of multi-valued structs, the owning instance contains a list of string values representing the unique keys of the composite objects (i.e. structs)

   (a) Single-valued struct
   (Track (location ?locKey))
   (Position (name ?locKey) (latitude ?lat) (longitude ?lon))

   (b) Multi-valued structs
   (Track (locationHistory $?locHist) (name ?trackKey))
   (Position (name ?posKey&:(member$?posKey $?locHist))

4. Associations and aggregations - are defined similar to structs. The associate or aggregate contains a reference to one or more unique object keys based on multiplicity.

   (a) Single-valued association (and aggregation)
   (View (name ?viewKey))
   (Object (view ?viewKey))

   (b) Multi-valued associations (and aggregations)
   (Asset (name ?assetKey))
   (Entity (assets $?assets &: (member$?assetKey $assets)))

Chapter 8 - CLIPS Agent Engine

8.1 Introduction

The CLIPS Agent Engine is an application for managing and running a group of CLIPS based agents. An instance of the CLIPS Agent Engine configured and initiated for a specific purpose is called an agent session. Each agent session can manage zero or more agents and can be configured independently from other agent sessions. Agents of a given agent session may collaborate among themselves and in a distributed fashion, using a distributed object application framework developed using the TIRAC toolkit.

An agent session can optionally be associated with a specific container object. If configured as such, only collectible objects contained by this container will be processed by the agent session. However, the agent session can be configured to allow all non-collectible objects to be processed. See section 8.4 for more information.

8.2 Architecture Overview

The CLIPS Agent Engine component is comprised of several pieces that fit together to provide the required functionality. These pieces are:

- Agent Session - This is the main entry point for the application.
- Session Manager - Manages an agent session runtime context switch between the CLIPS based agents and other management pieces.
- Semantic Network Manager - Synchronizes the state of the CLIPS COOL instances with their corresponding distributed objects in the application framework.
- Agent Manager - Manages the CLIPS run cycle for agents of an agent session.
- System Time Manager - If configured to do so, creates and periodically updates a SYSTEM-TIME COOL instance to be used for time sensitive pattern matching.

After initialization, an agent session calls on the Session Manager to begin the context switch cycle (loop). Each time through this loop the agents are given an opportunity to execute and events are processed. When the agents have nothing to do and there are no events to process, the Session Manager blocks waiting for external object events.

The Session Manager allows the agents to execute by passing control to the Agent Manager. In turn, the Agent Manager gives each agent with work to do (agents which have rule activations) a chance to run. Each agent is allowed to run, at most, its current number of activations (to prevent any one agent from monopolizing the process).

After one cycle through the runnable agents, the Agent Manager returns control to the Session Manager. The Session Manager then passes control to the Semantic Network Manager to process any events (both external and internal) that may have been queued while the agents were executing.

The Semantic Network Manager receives, queues, and processes external object events originating from the distributed object server and internal CLIPS events originating from the CLIPS agents. External object events result in COOL instances being created, deleted, or modified. Internal CLIPS events result in distributed objects being created, deleted, or modified.

The Semantic Network Manager uses the Subscription Service to create interests only in the class of objects relevant to the agents of an agent session. By narrowing the focus of interest in this way, the number of external events that the Semantic Network Manager must process can be greatly reduced.

8.3 Installation

The CLIPS Agent Engine is a component of the TIRAC toolkit. The TIRAC toolkit software distribution currently provides Windows and Linux versions of the CLIPS Agent Engine. Versions of the Agent Engine for other target platforms are available, but will require an additional, platform specific installation.

The TIRAC toolkit installation directory contains the ***kb*** subdirectory, which in turn contains all of the CLIPS constructs files that come with the CLIPS Agent Engine. These files are:

- ***Core.kbm*** - Redefines the MAIN CLIPS defmodule so that all constructs are exported to other modules that wish to import them.
- ***Core.kbc*** - Provides the definition of the top-level COOL defclasses: STRUCT (from which all object model structs directly inherit) and CoreObject (from which all object model classes inherit, perhaps indirectly.)
- ***Core.kbf*** - Contains the deffunctions and defmessage-handlers of the Agent Engine.
- ***Core.load*** - A CLIPS batch file which loads the basic set of Agent Engine CLIPS constructs for use in collaborative mode.
- ***Core_sh.kbf*** - Provides user function definitions to allow the use of the Agent Engine in a stand-alone CLIPS shell. These functions emulate collaborative mode user functions (methods that are normally only defined in collaborative mode) and the collaborative mode agent management control cycle. These functions only need to be loaded if you are running in stand-alone mode. Also see the description of user functions that follows.
- ***Core_sh_ext.kbf*** - Provides an extended set of user function definitions to allow the use of the Agent Engine in a stand-alone CLIPS shell, other than the one provided (e.g., CLIPSWin). These are merely “stubbed-out” functions of those user functions built into the provided CLIPS shell. These functions need to be loaded if you are running in stand-alone mode with an alternative CLIPS shell. Also see the description of user functions that follows.
- ***Core_sh.load*** - A CLIPS batch file which loads the basic set of Agent Engine CLIPS constructs plus the function definitions required for use in a stand-alone CLIPS shell.
- ***generic.kbm*** - Defines the GENERIC CLIPS defmodule used for the example GENERIC agent.
- ***generic.kb*** - Defines the CLIPS rules for the example GENERIC agent.
- ***setUtil.kbf*** - Defines functions used by the subscription utility.
- ***SubscritionUtil.kbf*** - Defines functions of the subscription utility. See ***makesubs*** batch file below.

In addition to the CLIPS files, a number of template files can be found in the ***scr/tpl*** directory along with a configuration utility Perl script ***ae_config.pl*** located in the ***scr/agentEngine*** directory. The configuration utility script makes use of the template files to provide the initial configuration of a project specific agent engine. The usage of this configuration utility is described in section 8.4. The template files are:

- ***AgentSessionSuite.xml.tpl*** - A template XML file for running an agent session via the execution framework (see Appendix D).
- ***StartAgentSession.bat.tpl*** - A template batch file for starting a Windows agent session.
- ***StartAgentSession.tpl*** - A template shell script for starting a Linux agent session.
- ***makesubs.bat.tpl*** - A template batch file for generating an interest file for the agent engine from Windows.
- ***makesubs.tpl*** - A template shell script for generating an interest file for the agent engine from Linux.

- ***agents.properties.tpl*** - A template properties file for defining the runtime behavior of an agent session.
- ***agents.load.tpl*** - A template CLIPS batch file for loading agent constructs in collaborative mode.
- ***agents_sh.load.tpl*** - A template CLIPS batch file for loading agent constructs in stand-alone mode.

8.4 Configuration

The CLIPS Agent Engine, as distributed, will need to be configured prior to use. There is a configuration utility and a number of template files, which are meant to help with this configuration. As described above, a Perl script is provided in the ***scr/agentEngine*** subdirectory of the TIRAC toolkit installation directory which can be used to setup your project specific agent engine. This script requires three parameters: the destination directory, the project name, and the domain name(s). The most straightforward approach to use this configuration utility under Windows is as follows:

1. Verify that a file type has been associated with Perl scripts.
2. Create a shortcut to the ***ae_config.pl*** Perl script.
3. Right-click on the shortcut and select ***Properties***.
4. Add to the end of the ***Target*** text field your destination directory, project library directory, project name, and domain name(s) separated by spaces. For example, if your project name is “FOO” with domains “FOO1_0” and “BAR1_0,” then your additions will resemble ***<project home>\kb <project home>\lib FOO FOO1_0 BAR1_0***.
5. Execute the shortcut by double-clicking on it. If prompted for property information, enter values for the requested properties.
6. After running this Perl script, there should be six new files in the specified destination directory (where ***<project>*** indicates the given project name):

- ***<project>*****_makesubs.*****bat*** - A batch file for invoking the subscription utility that generates an interests file for a set of agents. See section 8.4.2 below.
- ***<project>*****_agents.*****load*** - The default CLIPS batch file used to load agent engine constructs for collaborative mode. This generated file will need to be modified to include the CLIPS constructs specific to a project. See comments in this file for more information.
- ***<project>*****_agents_sh.*****load*** - The default CLIPS batch file used to load agent engine constructs for stand-alone mode. This generated file will need to be modified to include the CLIPS constructs specific to your project. If using an alternative CLIPS shell for testing the agents in stand-alone mode, it is necessary to uncomment the appropriate line in this file. See comments in this file for more information.
- ***<project>*****_AgentSessionSuite.*****xml*** - An execution framework XML document for starting an agent session in collaborative mode. A container name for which to start an agent session may be added to this document as a command line argument, or defined in the generated agent properties file. This generated XML document might need to be modified to add project specific classes or jars to the classpath. By default, the only project specific jars included in the generated file are the ***<domain>_c.jar*** files for each specified domain.
- ***<project>*****_StartAgentSession.*****bat*** - A batch file for starting an agent session in collaborative mode. A container name for which to start an agent session may be provided as a command line argument, or defined in the agent properties file. This generated batch file might need to be modified to add project specific classes or jars to the classpath. By default, the only project specific jars included in the generated batch file are the ***<domain>_c.jar*** files for each specified domain.
- **<project>*****_agents.properties*** - The Java properties required for running an agent session in collaborative mode. This generated properties file might need to be modified for specific project requirements. See Appendix B.7 *CLIPS Agent Session Properties* for more information on property settings.

8.4.1 Properties

There are a number of properties (see Appendix B.7) used to configure the Agent Engine to be run as a project specific agent session. Some are required, whereas others are optional. These properties are typically defined in the Java properties file indicated by the core.properties system property (e.g., ***-Dcore.properties=<project>_-agents.properties*** as specified in the ***<project>_AgentSessionSuite.xml*** *and* ***<project>_StartAgentSession.bat*** files.)

As mentioned in the *Introduction* (8.1), it is necessary to determine a collectible class during initialization. The collectible class can be specified explicitly by setting the collectible class property. Alternatively, it can be determined implicitly by setting both the container class and the collection role name properties. If neither of these methods are used to define a collectible class, then the root class (typically CoreObject) will be used by default. Once a collectible class has been determined, it is possible to optionally specify a container class and an association between the collectible class and the container class, and additional properties related to the collectible class. Refer to the optional session management properties section of Appendix B.7.

8.4.2 Subscriptions

The set of subscriptions (interests) used by a particular agent session is defined in the subscriptions configuration file (e.g., **<project>*_agents.interests***). As described in Appendix B.7, the interests file property indicates the location and name of the interests file and must be located in your Java classpath.

Each line of the interests file defines either a class based subscription or an attribute based subscription. Class based subscriptions have the form ***CLASS*:*<qualified class name>*** and result in a subscription to the creation and deletion of objects of the specified class. Attribute based subscriptions have the form ***<qualified class name>*:*<attribute name>*** and result in a subscription to the modification of the specified attribute for all objects of the specified class. All lines in the interests file beginning with “//” (double forward slash) are considered to be a comment and are ignored. Class names defined in the interests file can be fully qualified or partially qualified, but must be contained in one of the domains specified by the core domains property.

As an example, to have an agent session notified about all FOO.DomainObject creations and deletions, and also whenever an FOO.DomainObject’s owner attribute is modified, the interests file might be defined as:

```
CLASS:FOO.DomainObject
FOO.DomainObject:owner
```

You may either create and edit the interest file by hand, or use the generated ’makesubs’ batch file to help create the interests file for your project specific set of agents.

The configuration utility described above assumes your interest file will be named ***<project>_agents.interests***. This is the default value assigned to the interests file property. It is also the default output file used by the generated ***makesubs*** batch file ***<project>_makesubs.bat***.” The default defclass file names assumed by the configuration utility and used by the generated ***makesubs*** batch file take the form of ***<domain>.kbc*** for each domain. Optionally, you may provide the name of the interests file as a command line argument. See comments in the generated ’makesubs’ batch file for more information.

The ***makesubs*** batch file will search the current working directory (the directory in which the batch file is executed) and its subdirectories for all files with the “.kb” extension which contain the “defrule” CLIPS keyword. Each of these files is then parsed to identify object patterns on the LHS of each rule. Finally, an interests file is generated for all of the identified object patterns. See notes under section *8.5.3 Other Conventions* for additional information about object patterns.

8.4.3 CLIPS Batch Files

The purpose of a CLIPS batch file is to load CLIPS constructs into and initialize the CLIPS environment. The next part of the CLIPS Agent Engine configuration deals with the CLIPS batch files used to load CLIPS constructs and COOL instances into the CLIPS environment during the initialization of an agent session.

CLIPS Constructs

As previously mentioned, the CLIPS batch file for loading constructs into a collaborative mode agent session is specified by the agent load file property. The agent load file is 'batched' during agent session initialization **prior** to resetting the CLIPS environment. It must contain valid CLIPS batch file commands (typically CLIPS environment settings and load statements). Refer to the CLIPS Basic Programmer's Guide for more information on CLIPS batch file commands.

The CLIPS batch file ***<project>_agents.load*** generated by the configuration utility should be used as a starting point for defining a project specific agent session load file. The configuration utility will have added commands in the generated agent load file to load the Agent Engine CLIPS constructs from the "kb" subdirectory of the core installation directory. These load commands may need to be modified if the core installation directory used for runtime is different than what was used during configuration. Also, the configuration utility will have added to the agent load file commands to load the project specific ***<domain>.kbc*** defclass file(s). Again, this is based on the domain name(s) provided during configuration, and may need modification.

Typically, however, the only hand modifications required to the generated agent load file are to append project specific agent construct load commands. Each agent should have at least a module definition (defmodule) file and a knowledge base (defrules, etc.) file that need to be loaded. As an example, see the GENERIC agent included in the CLIPS Agent Engine software distribution and the comments contained in the generated agent load file. You will probably want to similarly modify the generated ***<project>_agents_sh.load*** file to load the desired agent constructs for running a stand-alone mode agent session.

COOL Instances

COOL instances can be instantiated during the initialization of a collaborative mode agent session by specifying an instances load file using the instances load file property. If specified, the instances load file is 'batched' during agent session initialization **following** a reset of the CLIPS environment. Any valid CLIPS batch file commands may be included in the instances load file, but typically only the CLIPS commands ***restore-instances*** and ***load-instances*** will be used. Note that instances loaded using the ***load-instances*** command will be instantiated both in CLIPS and in the distributed object environment. However, COOL instances loaded using the ***restore-instances*** command will be instantiated only in CLIPS (i.e., not in the distributed object environment).

8.4.4 Startup Files

Depending on your project specific use of the CLIPS Agent Engine, it may be necessary to modify the collaborative mode execution framework XML document ***<project>_AgentSessionSuite.xml*** or startup script ***<project>_-StartAgentSession.bat*** that was generated by the configuration utility. These files expect the Agent Engine runtime libraries (Windows DLLs) to be installed somewhere on the execution path or, in the case of the startup script, to be in the ***lib*** subdirectory of the core installation directory. These startup files further assume that the ***lib*** subdirectory of your project directory contains the project specific jar files and other required Java resources. If these assumptions are not valid for your project, you'll need to modify these files to reflect the correct project specific locations. See section 8.6 *Running an Agent Session* below for more information.

8.4.5 Object Model Requirements

As described in section 8.4.1 and in Appendix B.7, the CLIPS Agent Engine can make use of certain classes that are part of your application domain object model. Specifically, you should have the notion of a high-level domain object class (from which all of your domain objects inherit). Optionally, a container class can be defined which is associated with the domain object class. With the specification of a container class (and identification of a container object), the agent session will interact only with collectible objects which are part of the container's collection. Finally, an agent class can also be defined, which directly or indirectly inherits from the high-level domain object class. Specifying an agent class, and its agentId and activity attributes, allows the Agent Manager to update agent objects as they become active or inactive in CLIPS.

8.5 Writing CLIPS Agents

The primary purpose of the CLIPS Agent Engine is provide an environment in which CLIPS based agents may execute, reason, and ultimately support decision making. The Agent Engine itself does not provide any decision support. Hence, to provide this, agents will need to be developed in the CLIPS language, which can then be added to the base-level Agent Engine using the agent load file (as described above).

This guide does not attempt to explain the details of CLIPS based agent programming, and the reader is assumed to already have a good knowledge of the CLIPS language and execution environment. The CLIPS user's guide and programmer's guide should be referred to for specific information on usage of the CLIPS shell and programming language. To utilize the Agent Engine successfully certain conventions are required and others suggested. These conventions are described below.

8.5.1 Defining a Module

It is strongly suggested that for each domain agent in a given agent session, that a corresponding CLIPS module be defined. In general, each agent instance (object) defined should have some kind of identifier (e.g., agentId) that corresponds to a module definition. This agent identifier should be unique among all agents of a particular agent session. This allows the Agent Manager to update the activity level of each agent instance as it prepares to run that agent's module. It is possible to use the same module for multiple agent instances, or to not use a module at all. However, this does deviate from the normal use pattern of the Agent Engine, and great care should be taken to avoid unexpected behavior. It is recommended that the module definition for the GENERIC agent (generic.kbm) be copied and modified for each domain agent of your agent session.

8.5.2 Defining a Knowledge Base

The knowledge base of an agent will include rules, and possibly other CLIPS constructs. It is strongly suggested that a rule be included, which creates an appropriate agent instance. See the knowledge base definition for the GENERIC agent (generic.kb) for an example of how to create this 'bootstrapping' rule.

The defrules provided for an agent define its logical behavior. Patterns are established on the left-hand side (LHS) of a rule, which, when satisfied, cause some action or actions to take place as defined on the right-hand side (RHS) of the rule. In general, all standard CLIPS patterns and actions can be used when writing rules. However, there are certain conventions to adhere to when creating your rules.

Rule Patterns

The LHS of rules define the patterns on which a rule matches. Some of these patterns can be object pattern-matching and joins. Briefly, a join pattern literally "joins" two (or more) patterns using some kind of constraint. The constraint is typically a bound variable. For non-association joins (join patterns that do not use any association slots) no special consideration is required. However, when creating joins on object associations, always use the objectKey or name slot value of a matched object. Both of these slots are of type INSTANCE-NAME, which is also the type used for association slots.

For example, a join pattern which matches on all Alert instances of a particular Agent instance might be formed as:

```
?agent <- (object (is-a Agent)
   (objectKey ?agentKey))
?alert <- (object (is-a Alert)
   (alertAgent ?agentKey))
```

Here you see that the bound variable *?agentKey* is used to join the objectKey attribute of the Agent object to the association role alertAgent of the Alert object. This convention applies to more complex joins as well (such as multislot associations).

Additionally, take note that when generating a set of interests the subscription utility only considers the object patterns defined by the LHS of your rules. It does not generate interests for objects or slot values obtained using methods such as 'find-instance' or 'get-<slot>,' and does not consider the RHS of your rules at all. Therefore, it may be necessary to define patterns that would normally be considered 'bad form' for CLIPS programming. For example, normally it is considered 'good form' to only bind variables on the LHS of a rule that are to be used on the LHS of that rule (e.g., to join two patterns, or to pass to a test condition function call.) However, unless you plan to manually modify your interests file, you must include a pattern (and possibly a variable binding) for all values used on the LHS and the RHS of your rule. In this way, the subscription utility will automatically provide an interest for the pattern-matched slot or object, and the value will be kept 'in sync' with the distributed object's value.

Rule Actions

The RHS of a rule is where the action is. The typical kinds of actions performed by an agent are to create, modify or delete objects. It is also possible to have rules with actions that do not perform these kinds of actions, but in order to have an agent do anything meaningful (collaborate), one or more of these kinds of actions must be defined on the RHS of one or more rules. In order to perform these actions in a collaborative mode, there are a few special message-handlers defined on the CoreObject COOL defclass. These message-handlers are:

- init after - this message is implicitly sent to the object by using *make-instance* to create a new COOL instance. This results in a CLIPS CREATE event being sent to the Semantic Network Manager, which in turn creates a distributed object from the new COOL instance.
- update-slot <SLOT NAME> <SLOT VALUE> - this message must be explicitly sent to a COOL instance in order to set the slot value and cause a CLIPS UPDATE-SLOT event to be sent to the Semantic Network Manager. This message-handler will only work with single-slot values (non-multifield).
- update-struct <SLOT NAME> <STRUCT VALUES> - this message must be explicitly sent to a COOL instance in order to update a STRUCT object associated with the COOL instance. It results in a CLIPS UPDATE-SLOT event being sent to the Semantic Network Manager. This message-handler will only work with single-slot associations with STRUCT objects.
- update-struct-at <SLOT NAME> <INDEX> <STRUCT VALUES> - this message must be explicitly sent to a COOL instance in order to update the STRUCT object at the given index associated with the COOL instance. It results in a CLIPS UPDATE-SLOT event being sent to the Semantic Network Manager. This message-handler will only work with multi-slot associations with STRUCT objects.
- add-struct <SLOT NAME> <STRUCT INSTANCE NAME> - this message must be explicitly sent to a COOL instance in order to associate a new STRUCT object to a COOL instance. It results in a CLIPS UPDATE-SLOT event being sent to the Semantic Network Manager. This message-handler will work with both single-slot and multi-slot associations with STRUCT objects.
- remove-struct <SLOT NAME> <STRUCT INSTANCE NAME> - this message must be explicitly sent to a COOL instance in order to disassociate a STRUCT object from a COOL instance. It results in a CLIPS UPDATE-SLOT event being sent to the Semantic Network Manager. This message-handler will work with both single-slot and multi-slot associations with STRUCT objects.
- delete before - this message-handler is activated when a delete message is explicitly sent to the object. It results in a CLIPS DELETE event being sent to the Semantic Network Manager.
- add-assoc <SLOT NAME> < OBJECTKEY> - this message must be explicitly sent to a COOL instance in order to associate another object with this object. It results in a CLIPS ADD-ASSOC event being sent

to the Semantic Network Manager. This message-handler will work with both single-slot and multi-slot associations.
Note: The other end of the association will also be set in the associated distributed object, but only after the Semantic Network Manager processes this event. The other end of the association will **not** be set in the associated COOL instance unless there is an object slot pattern defined (and consequently an interest defined) for the associated object's role in the association.

- remove-assoc <SLOT NAME> <OBJECTKEY> - this message must be explicitly sent to a COOL instance in order to disassociate another object from this object. It results in a CLIPS REMOVE-ASSOC event being sent to the Semantic Network Manager. This message-handler will work with both single-slot and multi-slot associations.
  Note: The other end of the association will also be removed from the associated distributed object, but only after the Semantic Network Manager processes this event. The other end of the association will **not** be set in the associated COOL instance unless there is an object slot pattern defined (and consequently an interest defined) for the associated object's role in the association.

You may use other message-handlers on COOL instances such as get-<slot> and put-<slot> or some you've defined yourself. However, be aware that the put-<slot> message-handlers are 'non-collaborative' and only affect the COOL instance (the change is not propagated to the distributed object). Also, if you decide to provide your own specialized message-handlers be careful to not overwrite any of the specific message-handlers described above.

When sending a message to a instance, you'll typically want to only send to the INSTANCE-ADDRESS of the instance. If you try to send a message to the INSTANCE-NAME of an instance, the COOL message dispatcher might not be able to find the instance. This can happen if the defclass for the instance you're sending a message to is defined in a module other than the current module. This is a minor annoyance, perhaps even a bug, in CLIPS. Fortunately, the INSTANCE-ADDRESS of an instance is easily obtained by binding to it on the LHS of a rule or by using the InstanceAddress function provided with the CLIPS Agent Engine and defined below.

In addition to the set of message-handlers just described, there are several functions which can be used with the Agent Engine. These functions are:

- InstanceAddress <INSTANCE-NAME> - Attempts to get the INSTANCE-ADDRESS for the given instance name. It does this by looking for the given instance in each module, starting with the current module, until the named instance is found. If the instance is not found, the given INSTANCE-NAME is returned.

- GetUniqueName <CLASS NAME> - Generates a unique name for the given class (expected to be a symbol). Return type is INSTANCE-NAME.

- GetPropertyValue <PROPERTY NAME> - Attempts to get the Java property value for the given property if in collaborative mode. If in stand-alone mode, treats the given property as an environment variable and attempts to get the environment variable value. Return type is STRING.

- GetObjectId <INSTANCE-NAME> - Attempts to get the identifier of the object mapped to the given COOL instance. This provides a mechanism to access the internal object mapping information of the Semantic Net Manager. Return type is STRING.

- GetInstanceName <SYMBOL> | <STRING> - Attempts to get the name of COOL instance mapped to the given object identifier. This provides a mechanism to access the internal object mapping information of the Semantic Net Manager. Return type is INSTANCE-NAME.

- get-env <SYMBOL> | <STRING> - Attempts to get the value of the given environment variable. This is most useful in stand-alone mode, but can be used in collaborative mode also. Return type is STRING.

- agenda-length [* | <MODULE NAME>] - Determines the number of rule activations for one or more modules. If no arguments are provided, only the current module is used. If a specific module is named, then only that module's agenda-length is returned. If '*' is provided as the argument, then the sum of all rule activations for all modules is returned. Return type is INTEGER.

- manage-agent <MODULE NAME> - Attempt to focus and run the given module (agent). If the given module name is not a defined module, an error is printed an no action takes place. Otherwise, if the given module has rule activations, then that module is focused and allowed to run a maximum of its activation count number of rules. This is only available in stand-alone mode.

- manage-agents [<maximum cycles>] - Cycle through all defined modules for at most the given number of cycles (if provided). If a maximum number of cycles is not provided, a default of 100 is used. Each cycle 'manage-agent' is called for each defined module. This is only available in stand-alone mode.

- system-time - Returns the current system time as an integer which is the number of seconds since the Epoch (January 1, 1970 00:00:00 GMT). For more information, lookup the 'time()' function in a C library reference.

- format-date <FORMAT> <DATE> [UTC] - Returns a formatted date (time) as specified by the given FORMAT. A valid FORMAT string must conform to the requirements of the 'format' parameter of the C library function "strftime." The DATE argument is an integer representing the number of seconds since the Epoch (such as is returned from "system-time"). By default, the formatted date assumes "local" time and the given date is converted to the current timezone. The optional argument UTC indicates the date should not be converted and UTC (GMT) time is used instead of local time. For more information, lookup "time()," "localtime()," "gmtime()," and "strftime()" in a C library reference.

8.5.3 Other Conventions

There are a few other conventions that should be considered when working with CLIPS and the Agent Engine (and other parts of the TIRAC toolkit.) Most are common sense, but sometimes it helps to enumerate the known 'gotchas.'

- Do not use language specific reserved keywords in your object model. For example, a problem was found where an object model had a class attribute called *name*. During code generation, this results in a COOL defclass with a slot called *name* which is a reserved slot name in COOL.

- Limit the use of logical blocks in your rule patterns especially for object slots which are modified frequently. The behavior of a logical block on the LHS of a rule is to perform 'truth maintenance' by automatically retracting facts or deleting instances that were created on the RHS of a rule whenever patterns within the logical block change. Creation and deletion of instances, especially in a distributed object environment, is time intensive. Using logical blocks can therefore incur a lot of extra overhead and can seriously degrade the performance of the Agent Engine when running in collaborative mode. Normally, truth maintenance can be achieved without using logical blocks by using a set of three rules: one for the initial existence of a condition (a 'create' rule), one for changes to the condition (a 'modify' rule), and one for non-existence of a condition (a 'delete' rule).

- Be as specific as possible with your subscriptions (interests). Generally, the more focused your interests, the better performance you'll experience. Also, some patterns are required in order to cause interests to be properly generated. See section 8.5.2 *Rule Patterns* above for more information.

- Avoid the use of heavyweight function calls on the LHS of rules. This can't always be avoided, but limiting the use of such functions can greatly improve performance of the CLIPS pattern-match network. Also, take note that function calls on the LHS will **not** be re-evaluated unless some other fact or object pattern of the rule changes.

- Code your agents defensively. Anticipate invalid or incomplete information and decide how to handle this situation. Make sure it is handled on both the LHS and RHS of your rules and in your functions. In certain situations, it may be desirable to use 'object-pattern-match-delay' when manipulating instances to allow for the complete definition of instance slot values prior to performing pattern matching. See the CLIPS Reference Manual for more information.

- Test each agent individually: first in stand-alone mode and then in collaborative mode. Once successful tests have been carried out on each agent individually, then you can integrate your agents. Even then it is recommended that the agent integration proceed one at a time, with testing performed after each agent addition. Avoid the temptation to begin the testing of your agents in a fully integrated, collaborative mode environment. Simply put, the collaborative mode is too complex and can be quite time consuming for this level of testing.

8.6 Running an Agent Session

With the CLIPS Agent Engine properly installed and configured, and after writing your agents, it is possible to start an agent session in either stand-alone or collaborative mode.

8.6.1 Stand-alone Mode Agent Sessions

Running an agent session in stand-alone mode is very useful for testing and verifying your agent code before attempting to run in collaborative mode. It is highly recommended that agent developers get in the habit of using the stand-alone mode to do individual agent testing as well as agent integration testing. In practice, it has been found that this process provides very good results and much more reliable agent code.

To run a stand-alone agent session you will need a CLIPS interpreter, such as the CLIPS shell provided with the core distribution, or a 3rd party CLIPS interpreter, such as CLIPSWin. Either will work, although certain functions that are built-in to the provided CLIPS shell will not be available if using CLIPSWin. However, as described in section 8.3, there is a set of deffunctions provided in the ***Core_sh_ext.kbf*** file that provide 'stubbed-out' definitions of the built-in functions to allow the use of a 3rd party (external) CLIPS interpreter.

Begin by verifying the commands contained in the stand-alone CLIPS batch file (e.g., ***<project>_agents_sh.load***) as described in section 8.4.3 *CLIPS Batch Files*. Then use this batch file with the provided or 3rd party CLIPS shell to run the stand-alone agent session. For instance, when using the provided CLIPS shell, the command ***clips -f <project>_agents_sh.load*** can be used to start the CLIPS shell using the generated stand-alone CLIPS batch file. If instead, you are planning to use CLIPSWin (or some other external CLIPS interpreter), make sure to modify the generated stand-alone CLIPS batch file by uncommenting the line that loads the ***Core_sh_ext.kbf*** file. Then follow the instructions for running a batch file using the 3rd party CLIPS interpreter.

8.6.2 Collaborative Mode Agent Sessions

Note: *Before starting an agent session in collaborative mode, make sure the Object Server is running.*

Once you've tested your agent code using a stand-alone mode agent session, you'll probably want to try it out "for real" in collaborative mode. Using either the generated execution framework suite ***<project>_AgentSessionSuite.xml*** or startup script ***<project>_StartAgentSession.bat*** is typically the simplest way to run a collaborative mode agent session. There are a number of default assumptions made by these startup files which are identified in section *8.4*.

You may provide zero or one command line arguments when starting an agent session. The first argument, if provided, indicates the objectKey for the container object to use with this agent session. If this argument is not provided on the command line, it can be defined in the properties file. If a container objectKey is not specified by either mechanism, and a container class has been defined, then a new container object (with an arbitrary objectKey) will be created for you. See *CLIPS Agent Session Properties* (Appendix B.7) for more information.

Once started, an agent session will load the CLIPS constructs you've placed in your "agents load file" (e.g., ***<project>_agents.load***) and reset the CLIPS environment. Next, the interests file you've configured (e.g., ***<project>_agents.interests***) will be processed and appropriate subscriptions made. Once the subscriptions are defined, the agent session queries the object server for all objects of the subscribed to classes (optionally, which are also associated to the identified container.) Each of these objects is then instantiated in CLIPS as a COOL instance. At this point, if an ’instances load file’ was defined in your agents properties, these constructs will now be loaded. The final part of agent session initialization is to process any initial external events before dropping into the control cycle loop.

Once an agent session has entered the control cycle loop, it will continue to run until explicitly shutdown. A collaborative mode agent session can be shutdown using the CTRL-c key sequence, a kill signal, or by sending a remote shutdown command. If shutdown gracefully, the agent session will attempt to ’clean-up’ after itself (assuming the properties are configured to do so). Note that the agent session will only catch the CTRL-c key sequence or kill signal and shutdown gracefully if using Java 1.3 or higher. If using an older version of Java, the agent session simply exits.

Chapter 9 - Interoperability Bridge Framework

9.1 Introduction

The *Interoperability Bridge Framework* [17] provides a generic framework for seamless interaction and/or integration of multiple heterogeneous systems. A system can transparently connect to the Interoperability Bridge (either stand-alone or web-based) via a generic Connection API, request remote services and/or provide services to other remote systems. Information sent across the bridge is always in the native format of the originating system albeit in a valid XML format. As information representation can be quite dissimilar between different systems, a *Translation Service* plays the key role in transforming information from one system format to another.

The presence of a generic *Connection API* enables systems to register services with the *Interoperability Bridge*, publish local system requests and responses, and receive remote requests and responses without knowing intimate details about the remote system. All interaction for a system is directly with the Interoperability bridge. Remote requests are brokered by the bridge to a remote system that can service that request. Information published to the bridge is in XML format of the native system. Translators can be configured to translate the XML messages from a given remote system to native system format.

9.2 Architecture Overview

Figure 9.1 provides an overview of the *Interoperability Bridge* framework architecture. The *Interoperability Bridge* serves as a central publication and messaging service enabling systems to register services, lookup available services and publish remote requests and responses.

A remote request undergoes transformation in various stages as it passes through the different layers of the framework (see figure 9.2) before it is finally serviced. A remote request in the native format of the originating system, is first formated into the remote system XML format by the Remote *Connector* and published to the *Interoperability Bridge*. The *Interoperability Bridge* forwards this request without any transformation to the *Translation Service* where the remote XML is translated to local XML format and forwarded to the Local *Connector*. The Local *Connector* reformats the XML to the local request format. The transformed request is then processed via a Connection Delegate, which connects directly to the external system to service the request. Results from the remote request follow a similar path back to the requesting system.

9.3 Implementation

The *Interoperability Bridge* framework implementation provides the flexibility to transparently use either the local or the web-based remote bridge implementation. All communication between systems happens via the bridge. A *Connector* implementation connects a, possibly external, system to the bridge. Each system can configure the *Translation Service* to perform translation of documents received across the bridge from remote (or external) systems. Translation can be achieved using XSL-based transformation, inference-based transformation using an inference language such as JESS, or a combination of both. XSL mappings can be generated using tools such as MapForce™ [5].

The *Interoperability Bridge Framework* provides flexibility to easily develop extensions to support scenarios requiring specialized handling. Figure 9.3 provides the logical layout of the core framework classes.

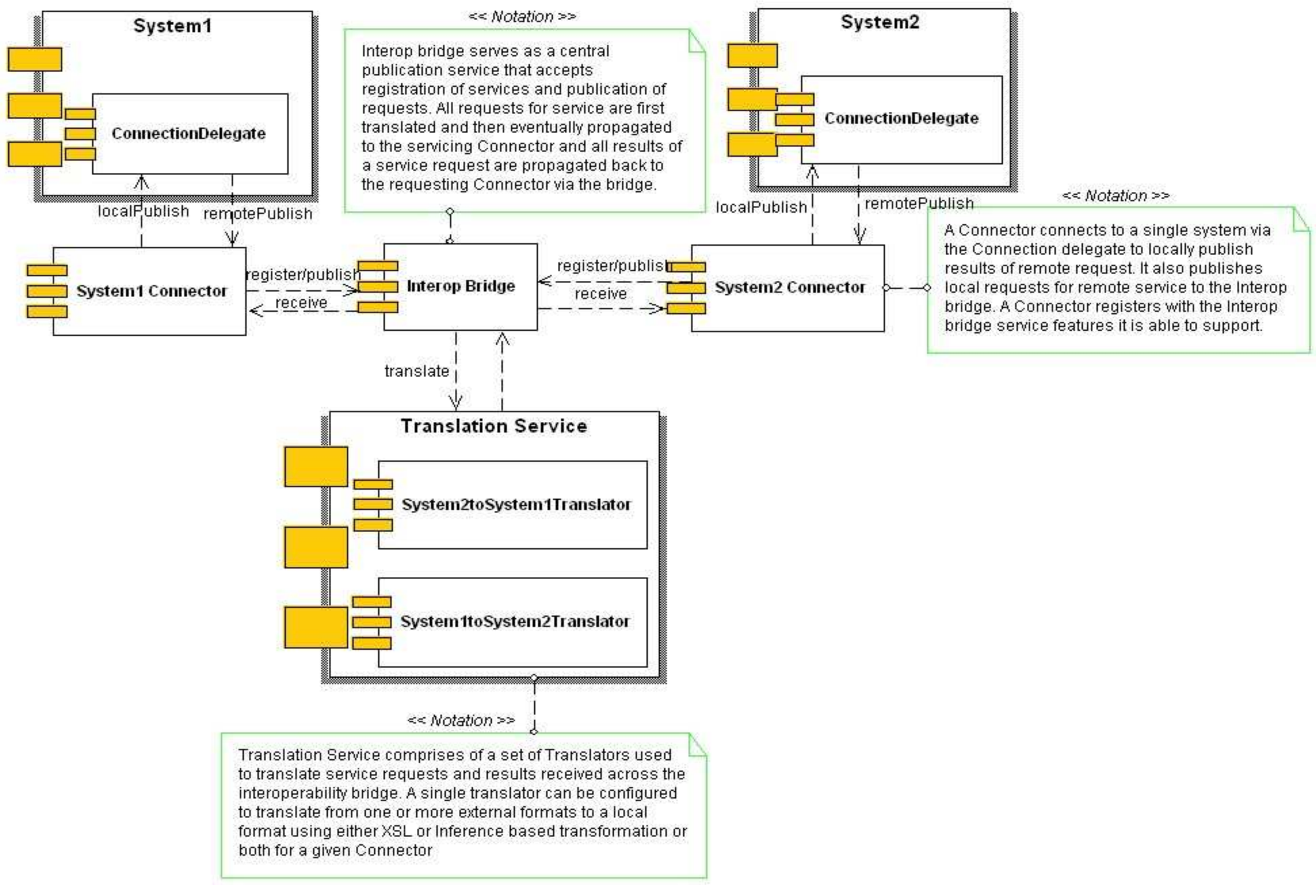

System1
ConnectionDelegate
localPublish
remotePublish
System1 Connector
register/publish
receive
<< Notation >>
Interop bridge serves as a central publication service that accepts registration of services and publication of requests. All requests for service are first translated and then eventually propagated to the servicing Connector and all results of a service request are propagated back to the requesting Connector via the bridge.
Interop Bridge
System2
ConnectionDelegate
localPublish
remotePublish
register/publish
receive
System2 Connector
<< Notation >>
A Connector connects to a single system via the Connection delegate to locally publish results of remote request. It also publishes local requests for remote service to the Interop bridge. A Connector registers with the Interop bridge service features it is able to support.
translate
Translation Service
System2toSystem1Translator
System1toSystem2Translator
<< Notation >>
Translation Service comprises of a set of Translators used to translate service requests and results received across the interoperability bridge. A single translator can be configured to translate from one or more external formats to a local format using either XSL or Inference based transformation or both for a given Connector

Figure 9.1: Interoperability Bridge Design.

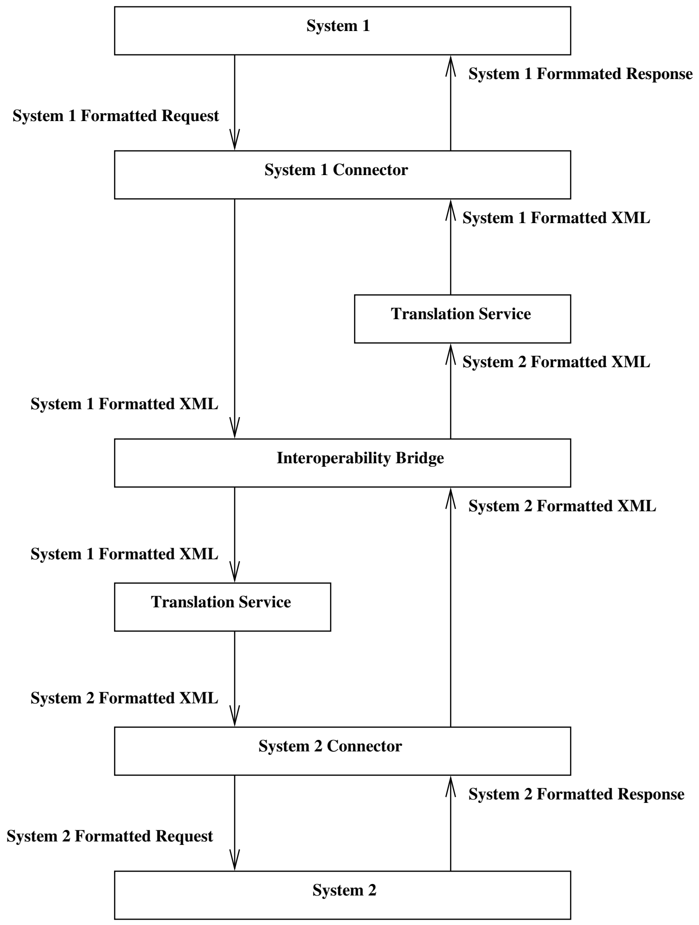


Figure 9.2: Request/Response path.

9.3.1 Description of key interfaces

- TranslationService – services the translation needs of a system using a series of translators to perform translation.
- TranslationInterface – interface definition for a Translator that performs translation of an XML document from one format to another.
- ConnectorInterface – interface definition for a Connector that connects a single system with the Interoperability Bridge.
- ConnectionDelegate – interface definition for a Connection delegate that connects to a single system and provides support for remote service requests such as query and subscription.
- XMLExportInterface – defines generic API to export instances to XML. For TIRAC-based systems the POWToXMLExport provides an implementation to export POW (refer to chapter 4) to XML.
- XMLImportInterface – defines API to import instances from XML. For TIRAC-based systems the XMLToPOWImport provides an implementation to import XML to POW instances.
- AbstractTransformer – defines API to transform data from one system to another. XMLBasedTransformer and InferenceBasedTransformer are two concrete implementations of this interface.

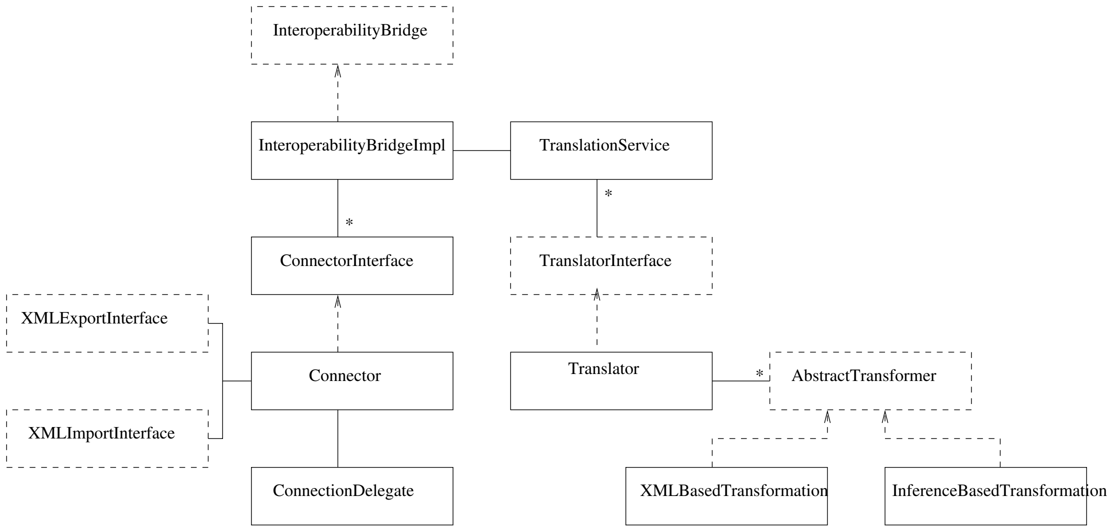


Figure 9.3: Class Diagram.

- InteroperabilityBridge – defines API for the central registration and publication service. InteroperabilityBridgeImpl (local implementation that requires all Connectors to co-exist) and RemoteInteroperabilityBridge (connects with the InteroperabilityWebService to publish service requests and retrieve response from external systems) are two implementations.

9.3.2 Interoperability Bridge

The *Interoperability Bridge* framework employs a Web service architecture to enable multiple systems to communicate with each other using a standard XML documents to describe service requests and responses. A system can register, with the bridge, services (e.g., query, subscription, etc.) it is able to provide. Remote service request results can be published back to the bridge. All documents received by the bridge are first translated to the target format before being forwarded to the target *Connector* for processing. Registration of services is accomplished by providing a description of available services in the form of an XML document conforming to the Service XML Schema (see *Service.xsd*). Each service request and response document must contain the following attributes specified on the root element:

sourceID – a unique identifier identifying the document originating system.

targetID – a unique identifier identifying the destination system if known. If targetID is not provided, an attempt is made to send the document to all registered systems.

requestID – an identifier uniquely identifying the particular service request. This id can be used at a later time to determine the state of a request (optional, but essential, if feedback is required or to determine status of a particular request).

serviceType – a string describing the nature of the request. Can be one of “data,” “query,” “subscribe,” “unsubscribe” or “status.” The default value is “data.”

Two particular implementations are provided for the InteroperabilityBridge interface within the framework: a local InteroperabilityBridge and a RemoteInteroperabilityBridge. The InteroperabilityBridgeImpl is a local implementation that requires all Connectors to co-exist in the same virtual machine. The other implementation is the RemoteInteroperabilityBridge that connects to the InteroperabilityWebService and does not require all Connectors to exist in the same virtual machine. Using the RemoteInteroperabilityBridge requires a Web server such as Jakarta Tomcat to be running. A Connector in this instance would connect to one known system and the RemoteInteroperabilityBridge. Before service requests from a given system can be fulfilled by another system, both systems must be connected. Requests are queued indefinitely until retrieved, by the Interoperability Bridge, so the order of connection is not a matter of concern. The choice of implementation to use can be specified in the following property (see Appendix B.8):

InteroperabilityBridge.serviceClassName=com.cdmtech.core.translation.RemoteInteroperabilityBridge

The default value for this property is the InteroperabilityBridgeImpl (i.e., local implementation that does not require a web server). If using the RemoteInteroperabilityBridge the following property must also be specified:

InteroperabilityWebService.url=http://localhost:8080/axis/services/InteroperabilityWebService

The above value is the default value. If a different setting is required then this property must be defined.

9.3.3 Translation Service

Translation plays a key role in the Interoperability Bridge framework. The ability to dynamically translate documents at runtime enables systems to collaborate seamlessly without requiring drastic changes in information representation in either system. Translation of data from one format to another can be achieved either by XSL transformation or via inferencing capability. Both types of transformation require XML schema files defining the data structure for the source and target data sources. When a system registers with the Interoperability Bridge, the *Translation Service* is configured to provide translation capability for the system. The *Translation Service* maintains a set of Translators to perform translation of documents received from external systems to the local system format.

A *Translator* employs all available transformations to transform an XML document from the external format to the local format. Two types of Transformer implementations are provided by the framework, namely an XSL-based transformation and an inference-based transformation (using the JESS inference engine).

XSL-based transformation

Simple translation of data from one format to another can be achieved using a XSL transformation file. Given XML schema definition files for the source and target data format, the XSL transformations can be easily generated off-line using tools such as MapForce. For TIRAC-based systems, a schema file can be generated directly from the XMI file by utilizing the makeSchema.pl perl script included with the distribution. For sample schema and XSL transformation files, see the example files included with the distribution.

JESS-based transformation

Complex translation can be accomplished by embedding an inference engine such as JESS or CLIPS. Employing an inference engine, complex logic can be specified based on the system state in addition to the already complex transformations between the two systems thus adding an extra level of filtering capability. For non TIRAC-based systems, the inference-based translator requires an implementation of XML import and export functionality to import the external XML to JESS facts and export JESS facts to XML or directly to java objects (or instances) recognized by the external system.

For a typical JESS-based transformation, it is necessary to provide a definition of information within the inference engine (i.e., deftemplates). An XML import to import the XML to JESS facts or a JESS user function named *xml-import* is required. Likewise an XML export to export the JESS facts to XML (or objects) or a JESS user function named *xml-export* is required.

For TIRAC-based systems, the ***LoadClasses*** userfunction can be used to automatically generate the deftemplate definitions by providing the schema file as input argument. To generate partial deftemplates from a subset of the domain, an attributes file containing class and attribute information can be provided. The format for this file is as follows:

<className>:<attrName>

where *className* is a valid class in the domain and *attrName* is a valid attribute in that class. Each entry must be on a new line.

To import from XML to facts the import class ***CoreXMLToFactImport*** can be used and to export facts to XML the ***FactToCoreXMLExport*** or ***FactToCoreExport*** (that transforms facts directly to TIRAC recognized instances) can be used.

LoadClasses userfunction provides the ability to define only a subset of classes and attributes as deftemplates by employing an attributes file containing a list of class and attribute names. The format for this file is as follows:

<class name>.<attribute name>

where ***<class name >*** is the root class where the ***<attribute name >*** should be defined

and ***<attribute name>*** is a valid attribute in the given class.

Each entry must be on a single line.

In addition to the attributes defined in a given schema file, the ***LoadClasses*** user function also defines the following additional slots:

- ***_class*** – contains the exact class name.
- ***_uniqueKey*** – a unique key for all classes except struct classes and usually containing the value of objectKey if it exists or a generated value. For query and subscription this is a generated value.
- ***_owner*** – for all struct classes in order to tie a given struct with the parent instance.
- ***_name*** – for all struct classes containing a unique generated value.

Mapping Requirements

Certain assumptions are made by the TIRAC-based import classes when importing a translated XML document. The schema generated from the XMI for a TIRAC-based system defines some additional attributes to support handling of not only query and subscription but also mapping of instances. When defining mappings, care must be taken to provide a suitable mapping of the required attributes, for the proper functioning of the translation and import capability for TIRAC-based systems.

1. Creation, deletion and modification of instances.
   A boolean attribute named ***clear*** is added to the CoreObject class whose default value is false and indicates a creation event. The core XML import maintains a mapping of external instance keys and the corresponding object keys. Future updates to an instance created in the current session is treated as modification and results in updating the instance. If the ***clear*** attribute value is set to true, then it is assumed to be a deletion and the mapped instance is deleted from the object store. **(Required for deletions)**

2. Query and subscription requests.
   Two new attributes are further added to the CoreObject class to support query and subscription:
   constraintType – which is an enumeration type defining the type of constraint. For possible values see eOpType class in the constraint/event model. (**Required**)
   eventType – which is an enumeration type defining the type of event with possible values “CREATE,” “DELETE,” and “MODIFY.” **(Required)**

3. Document root element.
   The following attributes are added to the root element:
   serviceType – which is an enumeration type with possible values “data”, “query,” “subscribe,” “unsubscribe,” and “status.” (**Required**)
   sourceID – a unique identifier for the source where the request or response originated. (**Required**)
   targetID – a unique identifier to send the request or response. (**Optional**)
   requestID – a unique identifier to identify a request or the response to a request. **(Optional)**

4. The core import class maintains a mapping between an external system instance unique identifier and the corresponding local system instance identifier if a unique key is mapped to the ***objectKey*** attribute in an instance of type CoreObject. Thus to maintain a link between an external system and a TIRAC-based system instance, a unique ***objectKey*** value must be defined. This enables future updates to be handled as modifications instead of as a creation. See figures 9.4 and 9.5 for possible ways to define a unique identifier.

9.3.4 Connection

Once the remote XML document has been translated to the local format, it is forwarded to the *Connector* for processing. If the document contains data (as specified in the "serviceType" field), it gets published to the local system by importing the XML to the local system instances if an XML import class is specified (see com.cdmtech.core.-client.xml.XMLToPOWImport for TIRAC-based systems). If the document contains a remote service request (such as query, subscribe or unsubscribe as specified in the "serviceType" field), the *Connector* delegates the request via a suitable *Connection Delegate*. The TIRAC specific implementation provides both a CoreConnector class that initializes both XML import and export capability and a CoreConnectionDelegate class that connects directly to a TIRAC-based system and provides support for both remote query and subscription requests. The following properties (see Appendix B.8) can be specified to indicate the ConnectionDelegate interface implementation class, the XMLExportInterface implementation class and the XMLImportInterface implementation class:

<sourceID>.connectionDelegateClassName=<ConnectionDelegate implementation class name>

<sourceID>.exportClassName=<XMLExportInterface implementation class name>

<sourceID>.importClassName=<XMLImportInterface implementation class name>

9.4 Using the Interoperability Bridge Framework

This section provides information on how to configure and connect different systems to the Interoperability Bridge. See Appendix B.8 for a description of all properties used for configuration.

9.4.1 Connecting two TIRAC-based systems

The framework provides an implementation of the *ConnectorInterface* to connect a TIRAC-based system to the *Interoperability Bridge*. Multiple TIRAC-based systems can be connected without any further implementation. The steps to follow to connect two TIRAC-based systems is as follows:

1. Using the makeSchema.pl script generate schemas for the two systems using the domain model XMI as input. For example, to generate a schema file for the basicModel domain with XMI file basicModel.xmi: *<install_dir>/scr/translation/makeSchema.pl -xmi <path to schema file/basicModel.xmi -out <path to output directory>*

2. Likewise, follow step 1 to generate the schema for the second TIRAC-based system.

3. Generate mappings to/from system 1 to system 2 using say MapForce and save the resulting XSL file. Both to/from mappings are necessary, even if only uni-directional flow of information is desired. This is because remote query/subscription requests from system 1 need to be translated before system 2 can interpret the request and once results are received back at the system 1 they need to be translated back for system 1 to consume. Instead of, or in addition to an XSL file, additional logic to perform complex mapping can be provided as JESS rules.

4. Add the following properties to your properties file - (assume for example that the domains for system 1 and system 2 are basicModel and basicTranslationModel, respectively).

- Specify schema files for the two system domains.
  <namespace1>.schemaFile=<schema file name>
  <namespace2>.schemaFile=<schema file name>
  For example***:***
  basicModel.schemaFile=basicModel.xsd
  basicTranslationModel.schemaFile=basicTranslationModel.xsd
- Specify transformation file to go from one system1 to system 2 (can be either XSL or JESS load files or a combination of both).
  - To specify XSL transformation file.
    <namespace1>To<namespace2>.xslTransformationFileName=<XSL file name>
  - To specify JESS load file containing transformation rules.
    <namespace1>To<namespace2>.inferenceTransformationFileName=<JESS load file name>
- Specify namespaces each system is interested in (values can be a comma-delimited list enclosed in square braces).
  <namespace1>.interestedNamespaces=[<namespace2>,...]
  <namespace2>.interestedNamespaces=[<namespace1>,...]
  For example***:***
  basicModel.interestedNamespaces=[basicTranslationModel]
- A complete list of domains must be provided in the following property.
  com.cdmtech.core.domains=[<comma-delimited list of domains>]

5. Within your application initialize the various connectors as follows:
   CoreConnector connector1 = new CoreConnector(namespace1)
   CoreConnector connector2 = new CoreConnector(namespace2)
   For example
   CoreConnector connector1 = new CoreConnector("basicModel")
   CoreConnector connector2 = new CoreConnector("basicTranslationModel")

6. To perform remote query, assuming namespace1 is the local system and namespace2 is the remote system and localClassName is a valid class name in the local system:

   - Build a com.cdmtech.core.util.constraints.Criteria object containing the query constraints to query for objects of a given class name:
     Criteria criteria = new Criteria(localClassName);
   - and send the request to the remote system by invoking:
     connector1.remoteQuery(namespace2, new Criteria[] {criteria});
     If namespace2 is null, the request is forwarded to all systems registered with the Interoperability Bridge, except the system where the request originated.

7. To subscribe with a remote system for the creation, deletion, or modification of instances for the same localClassName (See chapter 5 for more information on the event model):

   - Build an com.cdmtech.core.util.events.EventCriteria object containing the subscription criteria:
     EventCriteria criteria = new EventCriteria(localClassName,true,eEventType.CREATE);
   - and send the request to the remote system by invoking:
     connector1.remoteSubscribe(namespace2,new EventCriteria[]{ criteria });
     If namespace2 is null, the request is forwarded to all systems registered with the Interoperability Bridge except the system where the request originated.
   - To unsubscribe a previous subscription request use:
     connector1.unsubscribe(namespace2, new EventCriteria[] {criteria });

9.4.2 Connecting an external system to the bridge

In addition to all the steps above, a *ConnectorInterface* (or a *ConnectionDelegate*) implementation must be developed to connect the external system to the *Interoperability Bridge*. A *Connector* can be implemented either by :

1. Extending the *Connector* class or implementing the *ConnectorInterface* interface. If implementing the *ConnectorInterface* from scratch, care should be taken to ensure the external *Connector* provides the following functionality.
   - Registers with the *Interoperability Bridge*, services offered by the system using the schema description provided in the Services.xsd schema file on initialization.
   - Likewise, unregisters the services on shutdown.
2. Using the existing *Connector* implementation class, and providing an implementation of the *ConnectionDelegate* interface. An implementation for the *XMLImportInterface* and *XMLExportInterface* must also be provided. In this case the following properties must be provided in the properties file to specify the implementation classes:
   <sourceID>.connectionDelegateClassName=<class name>
   <sourceID>.importClassName=<class name>
   <sourceID>.exportClassName=<class name>

In general, the steps involved to connect two systems are as follows:

1. Provide a schema file for each system.
2. Generate mappings between any two systems intending to communicate with each other in XSL (using MapForce) or JESS. Even if communication is expected to be uni-directional, it is necessary to have bi-directional mappings.
3. For an external system, develop a *Connector* implementation following the steps enumerated above to connect the external system to the *Interoperability Bridge*.
4. Configure the *Translation Service* by providing all the properties.

9.4.3 Using the Interoperability Web Service

To decouple the individual connectors, the Web service implementation of the Interoperability Bridge can be utilized in addition to all the steps in the above two sections.

1. Download and install a Web server such as Tomcat [3].
2. Download and install Axis [4] and follow the installation instructions.
3. Use the provided makewebservice.pl script to copy all required artifacts to the Web server Web applications directory as follows:
   <install_dir>/scr/translation/makewebservice.pl -out $TOMCAT_DIR/webapps/axis
   where $TOMCAT_DIR is the TOMCAT installation directory.
4. Start the TOMCAT server using the startup batch file.
5. Verify the Interoperability Web service is running.
6. Edit the client properties file to include:
   InteroperabilityBridge.serviceClassName=com.cdmtech.core.translation.RemoteInteroperabilityBridge
   InteroperabilityWebService.URL=<url to locate web service >
7. Start up two clients that both connect to the Interoperability Web service. Follow steps outlined in previous two sub sections to configure the clients.

9.4.4 Translation UI

An extension of the Instance Viewer application, the Translation UI, provides a user interface component to rapidly setup two systems to interact with the Interoperability Bridge. When connecting two TIRAC-based systems, only the transformation files to translate information from one system to another are required to utilize this component. This component must be installed separately in order to use it.

9.5 Examples

The distribution contains examples to demonstrate functionality making use of two example domains, namely basicModel and basicTranslationModel. Both XSL and JESS-based transformations are provided to transform all instances from basicModel to basicTranslationModel and vice-versa. The XMI files for the two domains can be found in the <install_dir>/example/xmi directory. The XSL transformation files used by the XSL-based transformation can be found in the <install_dir>/example/suite/translation/lib directory while the JESS rules to perform the mapping can be found in the <install_dir>/example/suite/translation/jess directory. The source files for the examples are located in the <install_dir>/example/translation/src directory. A description of the various source files is as follows:

- CreateObjects.java - creates instances of the given class argument and all sub-classes. For each instance, both simple and complex attribute values are set. In the case of associations, an instance of the associated class is also created.
- TestQuery.java - initializes the two Connectors, formulates a query constraint, and requests a remote query.
- TestSubscription.java - initializes two Connectors, formulates a remote subscription criteria, and requests a remote subscription.
- TestQueryAndSubscription.java – initializes two Connectors, registers remote subscriptions, and performs remote query.

9.5.1 Mapping

All XSL-based transformation files are generated by describing the mappings in MapForce. All JESS-based mappings are defined by writing JESS rules. The following techniques are used by the JESS transformations used in the examples provided with the distribution:

- All classes along with all attributes in each class are mapped as JESS deftemplates.
- The corresponding JESS deftemplate name for a given class does not contain the prefix such as com.-cdmtech.core.client.corba.
- A JESS defrule is written for each class mapping only those attributes defined in that class.
- As it is critical for the right mapping class fact to be generated as a result of the transformation, salience values are used in the various rules in order to trigger the rules in the right order. For example, consider the following class hierarchy: *BaseObject* is the parent of *Track*; *Track* is the parent of *Aircraft*. The JESS rule defining *Aircraft* mappings has higher salience than the *Track* rule. Likewise the *Track* rule has higher salience than the *BaseObject* rule. This enables the *Aircraft* rule to fire first, thereby generating the correct mapped fact for the *Aircraft* instance. On the right hand side of each rule, first an attempt is made to find a fact for the given ***_uniqueKey***. A new fact is asserted only if one does not already exist, else the existing fact is modified.
- As same rules are used for mapping instances and to transform query and subscription criteria, the ***_unique-Key*** slot plays an important role. Note that query and subscription criteria rarely contain a value for ***object-Key*** slot. In those cases, the import class generates a unique value and sets the ***_uniqueKey*** slot value.

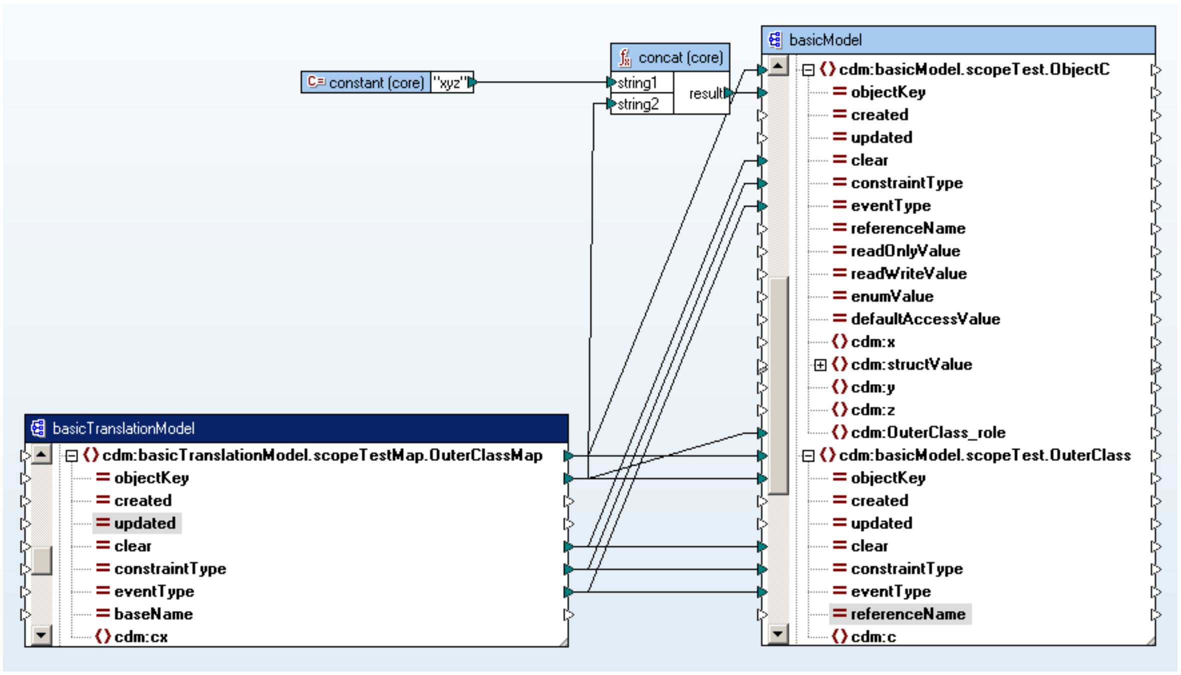


Figure 9.4: Mapping an instance in one system to two instances in the target system that are associated with each other.

9.5.2 Example suites

The provided example suites[1] demonstrate some of the salient features of the interoperability bridge framework, key among which are the translation capability employing XSL, JESS or a combination of both, ability to perform remote query and subscription, client-side and server-side import functionality. The various example suites are described below:

1. querySuite.xml – contains three examples demonstrating remote query:
   (a) xmlquerySuite.xml – demonstrates remote query employing XSL-based transformation and client-side import.
   (b) jessquerySuite.xml – demonstrates remote query employing JESS-based transformation and client-side import.
   (c) xmlqueryServerImportSuite.xml – demonstrates remote query employing XSL-based transformation and server-side import.
2. subscriptionSuite.xml – contains two examples demonstrating remote subscription:
   (a) xmlsubscriptionSuite.xml – demonstrates remote subscription employing XSL-based transformation and client-side import.
   (b) jesssubscriptionSuite.xml – demonstrates remote subscription employing JESS-based transformation and client-side import.
3. xmlassociationSuite.xml – demonstrates transforming a single instance in one domain to two instances in the target domain and associating the resulting two instances. In this case, the generated unique identifier

[1]See Appendix D for information on running execution suites.

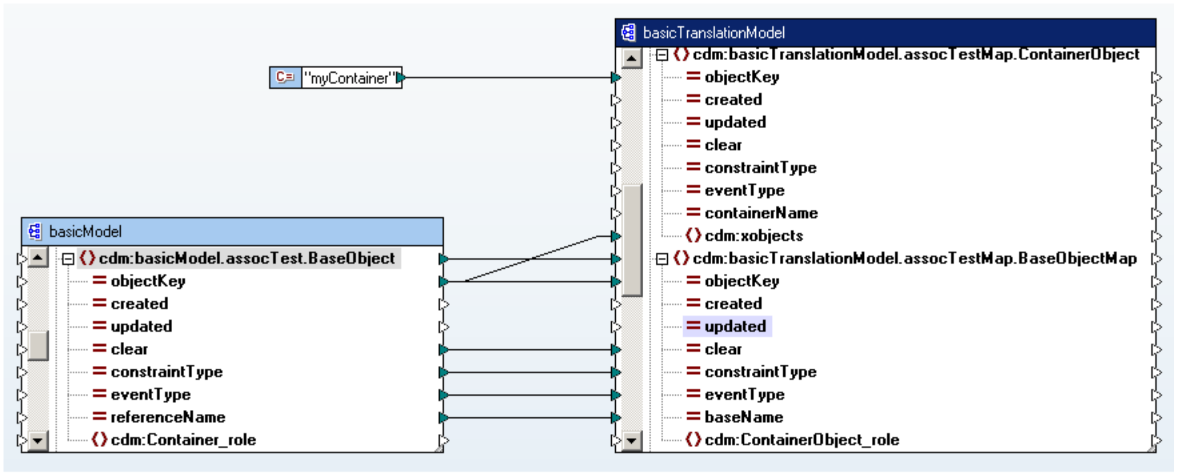


Figure 9.5: Mapping an instance of one class to an instance in the target system that is associated with a single instance of another class (in this case ContainerObject).

for the new instance must be somehow tied to the original instance in order to support future modifications (i.e., the mapping should not generate a new instance with each update, but should reuse the instance created the first time). A unique identifier is generated using the *concat* function. To insure that there is a binding between the mapped object and the original object, one of the inputs to the *concat* function is the *objectKey* value of the original object. Moreover, to keep the generated *objectKey* the same with each update, a constant value is concatenated to the *objectKey* value. Take note that the objectKey value cannot be used as is as it has to be unique for each instance and cannot be mapped to two instances at the same time. In order to associate the two resulting instances, a mapping can be made between the association role and the corresponding unique identifier of the new instance (see figure 9.4). This example demonstrates how an instance in one system can be mapped to multiple instances in the target system and the resulting instances further associated with each other. Every **OuterClassMap** in **basicTranslationModel** domain results in two new instances **OuterClass** and **ObjectC** in the **basicModel**. Further the resulting two instances are associated via the role *OuterClass_role* in **ObjectC (**Note that the role *c* in the **OuterClass** could also have been used).

4. xmlcontainerSuite.xml – demonstrates transforming an instance in one domain to another instance in the target domain and aggregating the resulting instance with a Container. Each new instance must contain a unique object identifier in order to map future modifications to the same mapped instance (see figure 9.5). In this example a **BaseObject** instance in the domain **basicModel** is mapped to a **BaseObjectMap** in the domain **basicTranslationModel**. In addition, each **BaseObjectMap** is always associated to a **ContainerObject** in the **basicTranslationModel** domain. As there is only a single **ContainerObject**, a constant unique identifier is generated for the **ContainerObject** by using the *constant* function. The **BaseObjectMap** is associated with the **ContainerObject** by mapping the *objectKey* of the **BaseObject** (which gets mapped to the **BaseObjectMap** *objectKey*) to the association role *xobjects* in the **ContainerObject** class.

5. mixedSuite.xml – demonstrates combining both XSL-based and JESS-based transformation to achieve the desired results. In example 4 above, all new BaseObject instances are associated with a given Container. When a BaseObject is deleted, the Container is automatically updated to remove the aggregate instance from its association role. Once all BaseObject instances are deleted, the Container still remains. To demonstrate usage of a mix of both JESS and XSL-based transformation, this example suite defines a JESS-based mapping to handle the Container removal when all BaseObject instances are deleted.

Appendix A - Release Notes

The following sections are the release notes for each individual component contained in the TIRAC toolkit. These release notes are provided as additional reference material and should not be the primary source of information. Since a good portion of this material is historic the information provided may be out of date (e.g. links may no longer reference existing or accessible documents). Some of the capabilities discussed in these notes pertain to the build and component maintenance infrastructure utilized internally and, therefore, may not be useful in development environments outside of CDM Technologies, Inc.

A.1 Core

Version 5.00

The Core toolkit can be best described as a "meta-framework" in that it contains the tools to facilitate the implementation of a client-server framework for distributed system development. The generated framework would be used to provide services for maintenance of a common information repository for any number of distributed applications. A reasonably complete capability is provided enabling generation of knowledge domain specific frameworks driven solely by the domain model (model driven architecture). Additionally, applications are included supporting implementation of rule-based agents facilitating incorporation of decision support acting on information maintained by generated services.

A.1.1 Requirements

- Agent Engine v5.00 - see section A.12
- Agent Management Layer v3.00 - see section A.10
- Client Support Library v4.00 - see section A.4
- Client Facade Support Library v2.00 - see section A.14
- Generic UI Components v4.00 - see section A.15
- Instance Viewer Application v3.00 - see section A.16
- JESS Agent Engine v3.00 - see section A.11
- Meta-Model Support Library v2.00 - see section A.8
- Object Graph Client Application v2.00 - see section A.18
- Object Management Library v5.00 - see section A.7
- Persistence Layer v2.00 - see section A.6
- Server Support Library v4.00 - see section A.5
- Object Shell v2.00 - see section A.17
- Support Suite v4.00 - see section A.2
- Interoperability Bridge v2.00 - see section A.13
- UML Processing Tools v2.00 - see section A.9
- Utility Class Library v4.00 - see section A.3

A.2 Support Suite

Version 4.00

The Core Support collection of tools contains scripts (written in Perl with particular attention paid towards providing platform independence) useful for maintaining an installation of components and applications.

A.2.1 Requirements

- Utility Class Library v4.00 - see section A.3
- xalan v2.5.0 - XSLT processor for transforming XML documents - Copyright ©1999-2003 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- xml-apis v2.5.1 - Xerces XML Parser Library- Copyright ©1999-2002 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- xercesImpl v2.4.0 - Xerces Implementation Library - Copyright ©1999-2002 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- jtidy v04aug2000r6 - HTML parser and pretty printer - Copyright ©1998-2000 World Wide Web Consortium (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved.
- getopt v1.0.9 - Command-line Option Processor - Copyright ©1998-2002 Aaron M. Renn

A.2.2 Usage Instructions

- update.pl -auto|a -dest|d=STRING <component name>[ver] ...
  - Requires environment variables TMP (path to directory to contain temporary downloads) and DEP_-URL (resource locator for component repository). **Note:** these environment variables may be specified on the command line in the form <variable name> = <value>.
  - If specified the '-auto' option will flag the script to perform the update(s) without any prompting.
  - The '-dest' option may be used to specify an alternative installation location (default is location pointed by CDM_HOME).
  - examples:
    - update.pl core
    - update.pl core_om -auto
    - update.pl core_om3_01
    - update.pl core_uml core_support DEP_URL=http://humpback TMP=/tmp
    - update.pl immaccs_iob -dest=e:\immaccs\iob
- remove.pl -dest|d=STRING <component name>[ver]...
  - Requires environment variables TMP (path to directory to contain temporary downloads) and DEP_-URL (resource locator for component repository). **Note:** these environment variables may be specified on the command line in the form <variable name> = <value>.
  - The '-dest' option may be used to specify an alternative installation location (default is location pointed by CDM_HOME).
- version.pl <component name> -dest|d=STRING
  - If no component name is specified then version information will be output for each installed component.

  - The ’-dest’ option may be used to specify an alternative installation location (default is location pointed by CDM_HOME).

- transform.pl -out|o=STRING -param|p=STRING -xsl|x=STRING <input file name>
  - The ’-out’ option may be used to specify the name of a file for the result document.
  - The ’-param’ option may be used to specify a stylesheet parameter in form <name> <expression>.
  - The ’-xsl’ option may be used for specification of the XML Style Sheet document.
- validate.pl -xs|x=STRING <input file name>
  - The ’-xs’ option may be used for specification of the XML Schema document. If not given then the XML Schema or DTD must be defined within the document (using DOCTYPE element).

A.2.3 Frequently Asked Questions

- How can I easily transfer existing release notes (using HTML) into XML conforming to the ReleaseNotes schema?
  - Remove all <font> tags (opening and closing) from HTML source.
  - Ensure, if the following sections are present, that the section titles contain the indicated text (within the <h2> element):
    - <h2>Description</h2>: general description of component/application.
    - <h2>Requirements</h2>: external requirements (components/applications that must be installed separately).
    - <h2>Distribution</h2>: description of component/application distribution contents.
    - <h2>Installation</h2>: installation procedure.
    - <h2>Usage</h2>: component/application usage.
    - <h2>Change</h2>: component/application release log entries.
    - <h2>Frequently</h2>: list of frequently asked questions with solutions.
    - <h2>Contact</h2>: list of points of contact.
  - Ensure HTML source is XML compliant HTML.
    - use supplied script, for example -> transform.pl core_support.html -o core_support.xhtml
    - correct problems as identified.
  - Edit resulting source to remove DTD preamble and namespace specification in root <html> tag.
  - Transform XHTML source into XML conforming to Schema for release notes (ReleaseNotes.xsd).
    - use supplied script, for example -> transform.pl core_support.xhtml -x HTMLToReleaseNotes.xsl -o core_support.xml
    - correct problems as identified.
  - Validate resulting XML document against XML schema for release notes.
    - use supplied script, for example -> validate -x ReleaseNotes.xsd core_support.xml
    - correct problems as identified.
  - **Note:** do not assume that this process will result in a complete transfer of all the original content. Inspect resulting document carefully for missing and incorrectly formatted output.

A.3 Utility Class Library

Version 4.00

Library of utility classes including support for error log management, property management, and assorted general purpose utility functionality.

A.4 Client Support Library

Version 4.00

The Core client support library provides all of the client-side services for working with distributed domain objects. These services include CorePersistence, CoreSubscription, and the base behavior for domain object factories and domain objects. The client support library provides ’wrappers’ used to establish communication and interact with these services, which are implemented by the Core server support library.

A.4.1 Requirements

- Utility Class Library v4.00 - see section A.3
- Persistence Layer v2.00 - see section A.6
- jacorb v2.1 - Java CORBA Implementation - Copyright ©Gerald Brose, Freie Universitaet Berlin/XTRADYNE Technologies AG, Germany, 1997-2004- GNU Library General Public License/The Apache Software License, Version 1.1

A.5 Server Support Library

Version 4.00

The Core server support library (core_server) provides all of the server-side services for working with distributed domain objects. These services include CorePersistence, CoreSubscription, ModelServer, and the base behavior for domain object factories and domain objects. The domain factories are generated from a domain object model XMI file. Domain objects are created and their life-cycle managed by the appropriate domain factories. The factories ensure object state is persisted and kept up-to-date via CorePersistence. Object factories also publish object events (creation, modification, deletion) with CoreSubscription as they occur. Notification of these object events is then sent to interested client subscribers by either CoreSubscription. The ModelServer provides runtime ’meta’ information which is used for activities such as server-side association management.

A.5.1 Requirements

- Client Support Library v4.00 - see section A.4
- Persistence Layer v2.00 - see section A.6
- Meta-Model Support Library v2.00 - see section A.8
- jess v6.1 - Java Expert System Shell - Copyright ©2002 by Ernest J. Friedman-Hill and the Sandia Corporation- JESS License (Sandia National Laboratories) **Note:** The JESS JAR must be copied into the Core installation *lib* directory as *jess6.1.jar*

A.6 Persistence Layer

Version 2.00

The Core Persistence Layer provides a general purpose set of persistence capabilities for Java objects. These capabilities include create, restore, update, and delete (CRUD) operations as well as constraint based query.

A.6.1 Requirements

- Utility Class Library v4.00 - see section A.3

A.7 Object Management Library

Version 5.00

The Object Management Layer class library provides general functionality for complete life-cycle management of objects, their attributes (characteristics) and associations (relationships). Interaction with object instances is simplified through the use of simple strings with attribute value constraint handled internally. Association management is also provided internally, alleviating the requirement (and complexity) to insure referential integrity by the using application. Management of interests is also provided - implemented internally and exposed to using applications through the standard Java event model. Additionally, support is provided for accessing multiple servers simultaneously and transparently. The primary application of this library would be for use by applications requiring little to no apriori knowledge of the object domain model(s). Internally, the required management and information is provided through runtime reflection and properties. A good example of such applications are user interfaces where a hard coded notion of the domain is expensive to both develop and manage.

A.7.1 Requirements

- Utility Class Library v4.00 - see section A.3

A.8 Meta-Model Support Library

Version 2.00

A.8.1 Requirements

- Utility Class Library v4.00 - see section A.3
- xalan v2.5.0 - XSLT processor for transforming XML documents - Copyright ©1999-2003 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- xml-apis v2.5.1 - Xerces XML Parser Library- Copyright ©1999-2002 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- xercesImpl v2.4.0 - Xerces Implementation Library - Copyright ©1999-2002 The Apache Software Foundation. All Rights Reserved.- The Apache Software License, Version 1.1
- nsmdf v0.1.1 - Novosoft Metadata Framework for Java - Copyright ©1999-2001, NovoSoft- GNU Library General Public License

A.9 UML Processing Tools

Version 2.00

A key capability provided by the Core toolkit is the ability to process project specific information representation in terms of an object model. The Unified Modeling Language (UML) standard was chosen as the high-level language for describing object models within the context of the Core framework. Additionally, the Extensible

Markup Language (XML) and, in particular, the XML Metadata Interchange (XMI) for the UML meta-model (version 1.3) was selected as the storage format for object model descriptions. The XMI-UML meta-model format is a standard that is supported by a number of modeling tools and, therefore, allows for a significant amount of flexibility in the selection of supporting applications. The UML Processing toolkit includes tools that produce various artifacts (code, reports, etc.) utilizing the XMI-UML output. These tools make extensive use of a UML meta-model based parser generated from the UML Meta-Object Facility (MOF) specification provided by the Object Management Group (OMG).

A.9.1 Requirements

- Support Suite v4.00 - see section A.2
- Utility Class Library v4.00 - see section A.3
- Meta-Model Support Library v2.00 - see section A.8
- Server Support Library v4.00 - see section A.5
- idl v2.1 - JacORB IDL to Java Compiler- Copyright ©Gerald Brose, Freie Universitaet Berlin/XTRADYNE Technologies AG, Germany, 1997-2004- GNU Library General Public License/The Apache Software License, Version 1.1
- getopt v1.0.9 - Command-line Option Processor - Copyright ©1998-2002 Aaron M. Renn

A.9.2 Usage Instructions

- makeall.pl < project name > -jikes -keep -nsref=STRING -trace -seq=STRING -xmi=STRING
  - Requires environment variable PROJECT_HOME (path to directory containing specific project directory(s)).
  - The CLASSPATH environment variable may be set (or defined on the command line) to include a path to any source code required by class implementations (such as for derived attributes or general operations).
  - The 'project name' argument is required (name of specific project directory located in PROJECT_-HOME).
  - The '-seq' option may be used to specify the type of sequence support generated for managing association ends. The available types are: primitive, sequence, or object. If not specified then simple sequence (sequence) support is generated (server-side array-type management without association management). The 'primitive' type will result in the generation of primitive array types (without server-side array-type management). The 'object' type will result in support for full server-side array-type and association management.
  - The '-xmi' option may be used to specify a single XMI file (contained in PROJECT_HOME/project name/xmi). If this option is not used then all XMI files contained in PROJECT_HOME/project name/-xmi will be processed.
  - If specified the '-jikes' option will flag the script to use the 'jikes' Java compiler instead of the default 'javac' compiler.
  - If specified the '-keep' option will flag the script to not delete generated artifacts (in PROJECT_-HOME/gen).
  - The '-nsref' option may be used to specify the initial reference used to bind to the name service (defaults to http://humpback/jacorb/NS_Ref). The value specified is used to generate the properties file required by JacORB.
  - If specified the '-trace' option will turn on remote method invocation logging (useful for testing and troubleshooting, but will impact performance).

- makeclient.pl < XMI file name > -gen=STRING -jikes -keep -out=STRING -seq=STRING
  - The CLASSPATH environment variable may be set (or defined on the command line) to include a path to any source code required by class implementations (such as for derived attributes or general operations).
  - The ’XMI file name’ argument is required (full path name of specific XMI file).
  - The ’-out’ option may be used to specify the output directory (default is current directory).
  - The ’-gen’ option may be used to specify the directory for generated output (default is the output directory).
  - The ’-seq’ option may be used to specify the type of sequence support generated for managing association ends. The available types are: primitive, sequence, or object. If not specified then simple sequence (sequence) support is generated (server-side array-type management without association management). The ’primitive’ type will result in the generation of primitive array types (without server-side array-type management). The ’object’ type will result in support for full server-side array-type and association management.
  - If specified the ’-keep’ option will flag the script to not delete generated interim artifacts.
  - If specified the ’-jikes’ option will flag the script to use the ’jikes’ Java compiler instead of the default ’javac’ compiler.
- makecool.pl < XMI file name > -out=STRING
  - The ’XMI file name’ argument is required (full path name of specific XMI file).
  - The ’-out’ option may be used to specify the output directory (default is current directory).
- makelatex.pl < XMI file name > -gen=STRING -out=STRING -pdf -pre=STRING
  - The ’XMI file name’ argument is required (full path name of specific XMI file).
  - The ’-out’ option may be used to specify the output directory (default is current directory).
  - The ’-gen’ option may be used to specify the directory for generated output (default is the output directory).
  - If specified the ’-pdf’ option will generate a PDF document from the generated LaTeX. **Note:** this requires installation of TeX/LaTeX including the PDF variations.
  - If specified the ’-pre’ option allows specification of an include file providing pre-content for the document (e.g., a title page).
- makelocal.pl < XMI file name > -gen=STRING -jikes -keep -out=STRING
  - The CLASSPATH environment variable may be set (or defined on the command line) to include a path to any source code required by class implementations (such as for derived attributes or general operations).
  - The ’XMI file name’ argument is required (full path name of specific XMI file).
  - The ’-out’ option may be used to specify the output directory (default is current directory).
  - The ’-gen’ option may be used to specify the directory for generated output (default is the output directory).
  - If specified the ’-keep’ option will flag the script to not delete generated interim artifacts.
  - If specified the ’-jikes’ option will flag the script to use the ’jikes’ Java compiler instead of the default ’javac’ compiler.
- makeproperties.pl < XMI file name > -out=STRING
  - The ’XMI file name’ argument is required (full path name of specific XMI file).
  - The ’-out’ option may be used to specify the output directory (default is current directory).

- makereport.pl < XMI file name > -out=STRING
  - The 'XMI file name' argument is required (full path name of specific XMI file).
  - The '-out' option may be used to specify the output directory (default is current directory).
- makeserver.pl < XMI file name > -gen=STRING -jikes -keep -out=STRING -seq=STRING
  - The CLASSPATH environment variable may be set (or defined on the command line) to include a path to any source code required by class implementations (such as for derived attributes or general operations).
  - The 'XMI file name' argument is required (full path name of specific XMI file).
  - The '-out' option may be used to specify the output directory (default is current directory).
  - The '-gen' option may be used to specify the directory for generated output (default is the output directory).
  - The '-seq' option may be used to specify the type of sequence support generated for managing association ends. The available types are: primitive, sequence, or object. If not specified then simple sequence (sequence) support is generated (server-side array-type management without association management). The 'primitive' type will result in the generation of primitive array types (without server-side array-type management). The 'object' type will result in support for full server-side array-type and association management.
  - If specified, the '-keep' option will flag the script to not delete generated interim artifacts.
  - If specified, the '-jikes' option will flag the script to use the 'jikes' Java compiler instead of the default 'javac' compiler.
- HTMLProducer
  - This tool will convert an xmi file generated by a UML program into a set of HTML pages in a Java API style format for ease of viewing model elements and their heirarchy, slots, and associations
  - The '-d' option may be used to specify the output directory (default is the same directory as the XMI)
  - The '-w' option may be used to specify a Web report URL. This is only used for the class diagrams generated by some UML applications. The images are expected to follow the format URL_ROOT/-Package/Package.jpg where package is the locally scoped package name.
  - The '-m' option may be used if a Web report is being used and a package name does not match the format URL_ROOT/Package/Package.jpg. In this case, the mappings should be specified in the format A:B,C:D,... where A,C,... are local package names and B:D,... are the related directories stemming from the specified Web report root. This option is ignored if no Web report is specified.
  - Alternatively, a simple GUI has been supplied for ease of use.

A.9.3 Frequently Asked Questions

- The code generation scripts fail (exceptions are thrown)
  - If the exceptions are thrown within the XMI parser it is likely that there are problems with the model XMI document. Validate the model XMI document file using the procedure described below and correct any problems indicated.
- How can I validate a model XMI file?
  - Make sure XMI model document conforms to the UML 1.4/XMI 1.1 specifications. Use the transform.-pl script as described below.
  - The DTD is specified in the DOCTYPE element (included by the XSL transform, if used) and must be defined as

        - <!DOCTYPE XMI SYSTEM "http://humpback.cdm.calpoly.edu/pub/doc/omg/uml/01-02-16.dtd">
    - Use the validate.pl script (provided by the Core Support distribution).
        - validate.pl <model file>

- How can I transform a model XMI file into a model document conforming to the current XMI/UML specifications?
    - Use the transform.pl script (provided by the Core Support distribution) along with the XSL transform provided to transform a model document file from UML 1.3/XMI 1.0 to UML 1.4/XMI 1.1 (provided by the Core Meta Support Library distribution).
        - transform.pl <input model file> -out <output model file> -xsl uml1_3xmi1_0-uml1_4xmi1_1.xsl
    - **Note:** It is not absolutely necessary to transform model documents to utilize the UML processing tools. Model files supplied in the older format will be transformed internally, however, at a slight cost in required processing time.

- How can I transform a model XMI file exported from Poseidon into a model document that can be processed?
    - Use the transform.pl script (provided by the Core Support distribution) along with the XSL transform provided to transform a model document file from Poseidon into a compliant XMI model file.
        - transform.pl <input model file> -out <output model file> -xsl PoseidonToCore.xsl

- What do the error codes, output by the scripts, mean?
    - The error codes appear to be codes output by the underlying operating system with Perl just passing the code. Unfortunately, no definitive source explaining these codes appears to be available and because they are operating system specific there will never be any platform independant definitions. However, most of these errors are typically caused by either problems in the object model or other requirements not being met (see notes above).

- The -pdf option for the makelatex.pl script fails.
    - This option requires installation of TeX/LaTeX including the PDF variations.

- The generated code fails to compile with the jikes compiler.
    - Check to make sure the JAVA_HOME environment variable is set.

- The system now successfully compiles but associations are failing when using the OML.
    - If the system was built using the -seq option set to "object" then make sure that the association manager plugins used with the OML are the "non-Managed" variants (i.e., AssociationSeq). See Object Management Layer properties for details.
    - Likewise, if the system was built using the -seq option set to "sequence" then make sure that the association manager plugins used with the OML are the "Managed" variants (i.e., ManagedAssociationSeq). See Object Management Layer properties for details.

A.10 Agent Management Library

Version 3.00

The Agent Management Layer component comprises of a generic framework API for rapid development of an agent engine for a given Rete implementation. The JESS Agent Engine distribution, which makes use of the Agent Management Layer to provide a JESS-based agent engine.

A.10.1 Requirements

- Object Management Library v5.00 - see section A.7

A.11 JESS Agent Engine

Version 3.00

The JESS agent engine component provides the ReteInterface implementation for the JESS inference engine. Together with the Agent Management Layer , it can be used to start and run a JESS-based agent engine.

A.11.1 Requirements

- Agent Management Layer v3.00 - see section A.10
- jess v6.1 - Java Expert System Shell - Copyright ©2002 by Ernest J. Friedman-Hill and the Sandia Corporation- JESS License **Note:** The JESS JAR must be copied into the Core installation *lib* directory as *jess6.1.jar*

A.11.2 Frequently Asked Questions

- How can I turn on dynamic subscriptions?
  - Change the following property in your JessAgentSession.properties file. If an interest file is also specified, subscriptions will be registered for those as well.
    - ***AgentSession.useDynamicSubscriptions=true***
- AgentSessionImpl vs InteractiveAgentSession which should I use?
  - The InteractiveAgentSession class internally starts the AgentEngine using AgentSessionImpl in a separate thread and provides interaction with the JESS inference engine. If you are using javaw or console-less start, it is preferable to use AgentSessionImpl to start the JESS Agent Engine.

A.12 CLIPS Agent Engine

Version 5.00

The Core agent engine provides management over a set of CLIPS based agents. An instance of the agent engine with a particular set of agents is called an agent session. Agents of an agent session can collaborate with each other locally using standard CLIPS mechanisms. Since the agent engine is an OML based client, agents can also collaborate in a distributed fashion with other agents, applications or human users.

A.12.1 Requirements

- Support Suite v4.00 - see section A.2
- Utility Class Library v4.00 - see section A.3
- Object Management Library v5.00 - see section A.7
- Client Support Library v4.00 - see section A.4
- clips v6.10 - C Language Integrated Production System - CLIPS License
- CLIPS User Guide v6.10 -
- CLIPS Programmers Guide v6.10 -

A.13 Translation Service

Version 2.00

The Interoperability Bridge provides a framework to connect two or more systems.

A.13.1 Requirements

- Utility Class Library v4.00 - see section A.3
- jess v6.1 - Java Expert System Shell - Copyright ©2002 by Ernest J. Friedman-Hill and the Sandia Corporation-JESS License **Note:** The JESS JAR must be copied into the Core installation *lib* directory as *jess6.1.jar*

A.13.2 Usage Instructions

- To generate the XML Schema, use the script provided with the distribution as follows:
  - **scr/translation/makeSchema.pl -out=<outdir> -xmi=<xmiFile>**
  - The various values that need to be provided are as follows
    - *outdir* – the output directory
    - *xmiFile* – the full path to the xmi file
- **Steps to start and run the example suites on the command-line, cd to your installation directory and execute as follows:**
  - ./runsuite example/suite/translationSuite.xml

A.14 Client Facade Support Library

Version 2.00

Provides support for client-side facades implementing object event notification. In addition to an object server API, this library provides a base object class, which provides instance interest management (listener registration/-notification), and a class management class, which provides class interest and base object instance management. For simplified derived attribute definition, the Derived abstract class is also provided.

A.14.1 Requirements

- Utility Class Library v4.00 - see section A.3
- Object Management Library v5.00 - see section A.7

A.15 Generic UI Component Library

Version 4.00

The Generic User Interface components provide some common components such as the following:

- Properties Editor (package: com.cdmtech.core.client.gui.prop) - The Properties Editor provides a simple user interface to view, edit, save, add, and remove properties at runtime. A component can register a BoundPropertyListener to receive notification when a property value is changed in order to dynamically update the state of the component.

- Customize Window (package: com.cdmtech.core.client.gui) - The Customize Window provides a user interface to enable selection of attributes to display for a given class. Other components such as the InstanceViewer and the AgentReport make use of this component to enable the user to select attributes to display.
- Debug Viewer (package: com.cdmtech.core.client.gui) - The Debug Viewer provides a minimal user interface to track ObjectListeners, ObjectModificationListeners, ObjectSelectionListeners, and ObjectActivationListeners registered with the POW and Template classes. The current used and available memory information is also provided.

Additional classes are included providing generic user interface capabilities common to a number of core user applications.

A.15.1 Requirements

- Object Management Library v5.00 - see section A.7

A.16 Instance Viewer Application

Version 3.00

The Instance Viewer component provides a generic user interface component that enables interaction with objects. It can be used to create, delete, and modify instances. It may also be used to formulate, perform queries, and display the results. The Instance Viewer component is comprised of

- InstanceViewer - that can be used to create, delete, edit, and to perform queries.
- QueryViewer - that can be used to display the results of a query.
- InstancesViewer - that can be used to view multiple instances.

A.16.1 Requirements

- Generic UI Components v4.00 - see section A.15

A.16.2 Frequently Asked Questions

- **How can I specify a display name for a given class object?** Specify for the 'disAttrName' class property for a given class as follows: *<className>.class.disAttrName = <attrName>* where *<className>* is a valid class name and *<attrName>* is a valid attribute in the class. **Note:** For each class a different attribute can be specified. Setting this property, results in this attribute value being set to the value typed in the 'Name' field in the InstanceViewer when creating a new object. Specifying the 'toString' class property will result in displaying this attribute value when the object is rendered in the InstanceViewer (e.g., *<className>.-class.toString = <attrName>*).
- **How can I display different images to display objects in the InstanceViewer?** Specify the class toSymbol property as follows: *<className>.class.toSymbol = image:<imageFileName>*. The default image used to render objects in the InstanceViewer is 'object.gif,' which can be found in the core_images.zip file. The following property can be edited to use a different image: *core.client.gui.objectImage = <imageFileName>*. Likewise, the image used for rendering class names in the Class Pane is 'class.gif,' also found in the core_images.zip file. To use a different image: *core.client.gui.classImage = <imageFileName>* where *<imageFileName>* is the image file name to use. Do not forget to add the image path to your classpath in the InstanceViewer batch file.

- **I don't see all the attributes in my class in the InstanceViewer.** Did you remember to set the following property to true: *core.client.gui.template.showHidden = true*? If this property is set to false, then all attributes defined as hidden (e.g., meta attributes) are not displayed in the InstanceViewer. Default value for this property is false.
- **How can I configure my InstanceViewer to view only some useful attributes in a given class instead of all?** For any class, the attributes to display in the InstanceViewer can be defined by specifying the attribute list in the following property: *<className>.class.InstanceViewer.customize = <attribute list>*. Where the *<attribute list>* is a valid set of attributes in the given class (e.g., CoreObject.class.InstanceViewer.-customize = [objectKey, created]).

A.17 Object Shell Application

Version 2.00

The Object Shell is a generic command-line based user interface. The syntax is defined by grammer specified in an annotated Java source file. Additionally, Java code is imbedded to directly invoke methods through the Object Management Layer (OML) in response to the parsing of commandline production segments (groups of tokens matching a prescribed pattern). Additionally, the Object Shell supports command completion.

A.17.1 Requirements

- Object Management Library v5.00 - see section A.7

A.18 Object Graph Application

Version 2.00

Simple graphical application for displaying objects and their associations.

A.18.1 Requirements

- Object Management Library v5.00 - see section A.7

Appendix B - Properties

B.1 Utility Properties

General utility properties.

Properties denoted with an asterisk (*) are required.

- core.logLevel = <logLevel>*: Information*
  Defines the log level. Valid values are (in descending order): Test, Debug, Information, Time, Warning, Error, Exception, and Fatal. Test, being the 'highest' log level will produce the most output, whereas Fatal, being the 'lowest' log level will produce the least output. Each 'higher' log level includes the output for that level as well as all 'lower' levels (e.g. Error also includes Exception and Fatal log levels.) If not specified, the default log level is Information.
  <logLevel> - Error log level.
  - Test
  - Debug
  - Information
  - Time
  - Warning
  - Exception
  - Fatal
- core.logFile = <fileName>
  Dump error log output to file.
  <fileName> - Name of file.
- core.writeOutputToScreen = *false*
  Flag to control output to screen console.
- core.outputDirectory = <directoryName>
  Output directory for error log files.
  <directoryName> - Name of directory.
- core.load.properties = [<fileName>,...]
  Additional property files to load. Each property file must be located relative to a path defined in the application's classpath.
  <fileName> - Name of file.
- core.properties.priority = <priority>*: 0*
  Priority of properties defined in property file. Higher priority takes precedent.
  <priority> - Property file priority.

B.2 Client Properties

Properties used by client applications. Refer to utility properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- core.domains = [<domain>,...]
  List of object model domains.
  <domain> - An object model domain name. This can be fully qualified (e.g. 'com.cdmtech.core.client.-corba.domain') or simply the 'domain' name.

- core.debugMode = *false*
  Flag to turn on log for debug event information. For each subscription client a log file is generated containing information of all events received by that client.

- Properties used by client CORBA components.

  – core.client.corba.CoreServerAPI.shutdownHook = *true*
    Enable or disable the CoreServerAPI shutdown hook, which is used to unsubscribe from all subscriptions when a client exits normally.

  – core.client.corba.invocationRetries = *2*
    Number of times to retry failed RMI before giving up.

  – core.client.corba.invocationBackoff = *500*
    Amount of time (ms) to backoff between invocation retries.

- Properties used by the Subscription Service.

  – core.client.corba.events.dispatchImmediately = *true*
    Flag indicating whether event notifications are dispatched immediately or periodically. If set to "false" then event notifications will be dispatched to listeners periodically as determined by dispatchFrequency. If set to "true," or not set, then event notifications will be dispatched to listeners as soon as possible after they are received.

  – core.client.corba.events.dispatchFrequency = *0*
    Positive integer that specifies the number of milliseconds to wait in-between event dispatching. There is no minimum or maximum value. The default is 0 if dispatchImmediately is true or 1000 if dispatchImmediately is false.

  – core.client.corba.events.orderPolicy = *1*
    Integer that indicates the type of event queue to use. The default is 1 (FIFO).

    ∗ LIFO - 0 (Last in First Out)

    ∗ FIFO - 1 (First in First Out)

    ∗ Priority - 2 (Priority Ordering – not supported yet)

        - Deadline - 3 (Deadline Ordering)

    - core.client.corba.events.eventRetry = *3*
      Integer that indicates to the server the number of times to retry sending a notification of event(s) to a subscriber before "dropping" the event(s).

    - core.client.corba.events.subscriberRetry = *60*
      Integer that indicates to the server the number of times to retry connecting to a subscriber (Subscription Client callback object) before assuming the subscriber no longer exists, and removing all of the subscriber's subscriptions. There is no minimum or maximum value.

    - core.client.corba.events.maxEventCount = *1000*
      Maximum number of events to send in a single notification. Setting this value too small can significantly degrade performance if a client can be expected to receive a large number of events in a relatively short period of time. It is recommended to use the default unless the client has specific memory resource considerations. Must be between 1 and 1000 (inclusive). Default is 1000.

B.3 Server Properties

Properties used by domain base services. All server properties (properties that begin with "core.server.corba") can optionally be prefixed with the domain name. This is useful if services for multiple domains will be configured from the same properties file. Refer to persistence layer properties and utility properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- Properties used by the Subscription Service.

    - [<domain>.]core.server.corba.events.useConstraintBasedSubscriptionService = <flag>*: false*
      For each domain, either the simple or constraint based Subscription Service will be used. By default, the simple Subscription Service is used.
      <domain> - Unqualified model domain name.
      <flag> - Boolean flag.

        - *true* | *on* | *enabled*
        - *false* | *off* | *disabled*

    - Properties that control event dispatch.

        - [<domain>.]core.server.corba.events.dispatchImmediately = <flag>*: false*
          Flag indicating whether queued events are dispatched immediately or periodically. If set to "false" then events will be dispatched periodically as determined by dispatchFrequency. If set to "true" then events will be dispatched as soon as possible after they are received.
          <domain> - Unqualified model domain name.
          <flag> - Boolean flag.

            - *true* | *on* | *enabled*
            - *false* | *off* | *disabled*

    - [<domain>.]core.server.corba.events.dispatchFrequency = *5000*
      Positive integer that specifies the number of milliseconds to wait in-between queued event dispatching. There is no minimum or maximum value.
      <domain> - Unqualified model domain name.

  - Log debug information to file about subscription events being sent to client.

    - [<domain>.]core.server.corba.events.debugMode = <flag>*: false*
      Enable or disable events debug mode.
      <domain> - Unqualified model domain name.
      <flag> - Boolean flag.

      - *true | on | enabled*
      - *false | off | disabled*
    - [<domain>.]core.server.corba.events.logQueuing = <flag>*: false*
      Enable or disable logging of event queuing when in debug mode.
      <domain> - Unqualified model domain name.
      <flag> - Boolean flag.

      - *true | on | enabled*
      - *false | off | disabled*

- Properties used specifically by the Constraint Based Subscription Service.

  - [<domain>.]core.server.corba.events.useCreateConstraints = <flag>*: true*
    Flag indicating whether or not constraints are used with object creation events.
    <domain> - Unqualified model domain name.
    <flag> - Boolean flag.

    - *true | on | enabled*
    - *false | off | disabled*
  - [<domain>.]core.server.corba.events.checkHasSubscriptions = <flag>*: true*
    Flag indicating whether or not to check if there is an interest in a particular object event before sending the object event to the filter engine.
    <domain> - Unqualified model domain name.
    <flag> - Boolean flag.

    - *true | on | enabled*
    - *false | off | disabled*
  - [<domain>.]core.server.corba.events.filter.loadFile = <cbssLoadFile>*: events.load*
    Can be used to change the default Constraint Based Subscription Service load file.
    <domain> - Unqualified model domain name.
    <cbssLoadFile> - Name of Constraint Based Subscription Service load file.

  - [<domain>.]core.server.corba.events.filter.debugMode = <flag>*: false*

    <domain> - Unqualified model domain name.
    <flag> - Boolean flag.

    - *true | on | enabled*

    * *false | off | disabled*
  - [<domain>.]core.server.corba.events.filter.debugFile = <logFileName>*: cbss_filter.out*

    <domain> - Unqualified model domain name.
    <logFileName> - Name of file to log information to.

- Properties used by domain factories. Defines behavior for all factories or for specific factories.

  - The lazyActivation property defines when objects are activated. Set to 'on,' 'enabled,' or 'true' to have objects activated lazily (i.e., only when actually used via a method invocation). This can provide performance enhancements for object creation and resolution, but should be set based upon intended object usage. If not set or if set to anything other than 'enabled,' 'on,' or 'true' then lazy object activation will not be used (objects will be activated during object creation or resolution).

    * [<domain>.]core.server.corba.factory.lazyActivation = <flag>*: disabled*
      The lazyActivation property for all factories
      <domain> - Unqualified model domain name.
      <flag> - Boolean flag.

      · *true | on | enabled*
      · *false | off | disabled*
    * [<domain>.]core.server.corba.factory.<factory>.lazyActivation = <flag>*: disabled*
      The lazyActivation property for specific factories
      <domain> - Unqualified model domain name.
      <factory> - Factory name.
      <flag> - Boolean flag.

      · *true | on | enabled*
      · *false | off | disabled*
  - The servantMgr property defines which servant manager implementation to use. Set to 'activator' (case ignored) to use the ServantActivator implementation, which uses the POA active object map (AOM). This can result in a small performance enhancement, with the tradeoff being resource utilization and a modest risk of thread race conditions (hopefully to be eliminated in the future.) If not set or if set to anything other than 'activator,' then the ServantLocator implementation will be used. The locator approach does not use the AOM, thus resulting in a minimal performance penalty.

    * [<domain>.]core.server.corba.factory.servantMgr = <servantMgr>*: locator*
      The servantMgr property for all factories
      <domain> - Unqualified model domain name.
      <servantMgr> - Servant manager type indicator.

      · locator
      · activator
    * [<domain>.]core.server.corba.factory.<factory>.servantMgr = <servantMgr>*: locator*
      The servantMgr property for specific factories
      <domain> - Unqualified model domain name.
      <factory> - Factory name.
      <servantMgr> - Servant manager type indicator.

      · locator

      · activator

- The ’maxSize’ property defines the maximum number of active objects to cache. If not set to a valid (maxSize >= 1) integer value then defaults to 100.

  * [<domain>.]core.server.corba.factory.objectCache.maxSize = *100*
    The maxSize property for all factories
    <domain> - Unqualified model domain name.

  * [<domain>.]core.server.corba.factory.<factory>.objectCache.maxSize = *100*
    The maxSize property for specific factories
    <domain> - Unqualified model domain name.
    <factory> - Factory name.

- The ’freeFactor’ property defines the percentage of active objects to free-up (deactivate) before adding to an already full cache. If not set to a valid (0.0 <= freeFactor >= 1.0) float value then defaults to 0.25F (i.e., the 25% least active objects of the object cache will be deactivated before activating an object when the object cache is full).

  * [<domain>.]core.server.corba.factory.objectCache.freeFactor = *0.25*
    The freeFactor property for all factories
    <domain> - Unqualified model domain name.

  * [<domain>.]core.server.corba.factory.<factory>.objectCache.freeFactor = *0.25*
    The freeFactor property for specific factories
    <domain> - Unqualified model domain name.
    <factory> - Factory name.

- The ’timeToLive’ property defines the minimum number of seconds that objects will remain in the active object cache during periods of inactivity. The maximum number of seconds that objects will remain in the active object cache during periods of inactivity is twice the minimum. If not set to a valid (timeToLive >= 1) integer value, then defaults to 300 (i.e., the active object cache will be cleared after between 5 and 10 minutes of inactivity).

  * [<domain>.]core.server.corba.factory.objectCache.timeToLive = <period>*: 300*
    The timeToLive property for all factories
    <domain> - Unqualified model domain name.
    <period> - Time period (in seconds).

  * [<domain>.]core.server.corba.factory.<factory>.objectCache.timeToLive = <period>*: 300*
    The timeToLive property for specific factories
    <domain> - Unqualified model domain name.
    <factory> - Factory name.
    <period> - Time period (in seconds).

- Event priority for events generated in a given factory. Default behavior is first in first out (FIFO) policy. Clients wishing to prioritize events must not only set a value greater than 0 for those factory events but also specify PRIORITY ordering in their client properties file.

  * [<domain>.]core.server.corba.factory.eventPriority = *0*
    The eventPriority property for all factories
    <domain> - Unqualified model domain name.

- ∗ [<domain>.]core.server.corba.factory.<factory>.eventPriority = *0*
  The eventPriority property for specific factories
  <domain> - Unqualified model domain name.
  <factory> - Factory name.

B.4 Persistence Layer Properties

Properties used by the Core Persistence Layer. All persistence layer properties (properties that begin with "core.-persist") can optionally be prefixed with the domain name. This is useful if services for multiple domains will be configured from the same properties file. Refer to utility properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- Properties used by all persistence implementations.

  – [<domain>.]core.persist.class = <persistClass> | *com.cdmtech.core.util.persist.serial.SerialPersistence* | *com.cdmtech.core.util.persist.jdbc.JDBCPersistence*
  Defines the implementation class of the Persistence interface. If not specified, SerialPersistence will be used.
  <domain> - Unqualified model domain name.
  <persistClass> - Name of class implementing Persistence.

  – [<domain>.]core.persist.type = *serial* | *jdbc*
  Defines the type of com.cdmtech.core.util.persist.Persistence implementation. This is a convenience for indicating the kind of desired persistence without having to specify the actual implementation class. There is no default value.
  <domain> - Unqualified model domain name.

  – [<domain>.]core.persist.autoConnect = <flag>*: enabled*
  Indicate whether the connection to persistence is automatically established. If not set, or set to anything other than ’disabled,’ ’off,’ or ’false’ then autoConnect is enabled.
  <domain> - Unqualified model domain name.
  <flag> - Boolean flag.

    ∗ *true* | *on* | *enabled*
    ∗ *false* | *off* | *disabled*

  – [<domain>.]core.persist.autoClose = <flag>*: disabled* | <period>
  Indicate whether the connection to persistence is automatically closed (after a period of inactivity). If set to ’enabled,’ ’on,’ or ’true’ then auto-close is enabled with the default inactivity period. If set to a positive integer, then auto-close is enabled using the specified inactivity period (in seconds). If not set, or set to some other value, then auto-close is disabled.
  <domain> - Unqualified model domain name.
  <flag> - Boolean flag.

    ∗ *true* | *on* | *enabled*
    ∗ *false* | *off* | *disabled*

  <period> - Time period (in seconds).

  - [<domain>.]core.persist.cleanUp = <flag>*: disabled* | <period>
    Indicate whether periodic clean-up (garbage collection) is to be performed. If set to 'enabled,' 'on,' or 'true' then clean-up is enabled with the default clean-up period of 60 seconds (once a minute). If set to a positive integer, then clean-up is enabled using the specified clean-up period (in seconds). If not set, or set to 'disabled,' 'off,' or 'false' then clean-up is disabled.
    <domain> - Unqualified model domain name.
    <flag> - Boolean flag.

    * *true* | *on* | *enabled*
    * *false* | *off* | *disabled*

    <period> - Time period (in seconds).

- Properties used by com.cdmtech.core.util.persist.serial.SerialPersistence.

  - [<domain>.]core.persist.serial.location = *data*
    Defines the root persistent object store directory of CorePersistence when using the SerialPeristence implementation. The directory path specified can be absolute (e.g. C:/my_data) or relative to the directory in which CorePersistence is started. If not specified, defaults to relative directory path 'data.'
    NOTE: if base services for multiple domains are all started from the same directory, be sure to specify a different data location for each. Also, it is STRONGLY advised to put your root directory at a fairly high level (i.e., do NOT specify a deeply nested directory).
    <domain> - Unqualified model domain name.

  - [<domain>.]core.persist.serial.storage = <storage>*: enabled*
    Defines the persistent object storage configuration of CorePersistence when using the SerialPeristence implementation. Set to 'initialize' to allow for initialization only (initialize CorePersistence from the object instances pre-existing in the object instance store, but then disable persistent storage following initialization.) Set to 'disabled,' 'off,' or 'false' to completely disable use of persistent object store. If not set or if set to anything other than 'initialize,' 'disabled,' 'off,' or 'false' then persistent object storage will be enabled.
    <domain> - Unqualified model domain name.
    <storage> - Storage type indicator.

    * *true* | *on* | *enabled*
    * *false* | *off* | *disabled*
    * initialize

  - Properties relating to the object cache of SerialPersistence.

    * [<domain>.]core.persist.serial.objectCache.maxSize = *0*
      If maxSize is specified as a positive integer, the object cache will have this maximum size. Otherwise, the object cache will have an unlimited size.
      <domain> - Unqualified model domain name.

    * [<domain>.]core.persist.serial.objectCache.freeFactor = *0.25*
      If freeFactor is specified as a percentage in the range of 0.0 to 1.0 (inclusive), this percentage of objects will be removed from the object cache whenever the object cache reaches its maximum size. If maxSize is not specified, freeFactor is not used.
      <domain> - Unqualified model domain name.

    * [<domain>.]core.persist.serial.objectCache.timeToLive = <period>*: 0*
      If timeToLive is specified as a positive integer (in seconds), the object cache will be flushed after the specified period of inactivity. Otherwise, the object cache will not be flushed periodically.
      <domain> - Unqualified model domain name.
      <period> - Time period (in seconds).

    * [<domain>.]core.persist.serial.objectCache.delayedUpdate = <period>*: 0*
      If delayedUpdate is specified as a positive integer (in seconds), updates to cached object attributes will only be written (to disk) after the specified period. Otherwise, delayed updates are disabled, and all attribute updates will be written immediately.
      <domain> - Unqualified model domain name.
      <period> - Time period (in seconds).

- Properties used by com.cdmtech.core.util.persist.jdbc.JDBCPersistence.

  - *[<domain>.]core.persist.jdbc.driver =
    Defines the JDBC driver class name.
    <domain> - Unqualified model domain name.

  - *[<domain>.]core.persist.jdbc.url =
    Defines the JDBC database URL.
    <domain> - Unqualified model domain name.

  - [<domain>.]core.persist.jdbc.classpath =
    Defines the classpath to JAR containing driver.
    <domain> - Unqualified model domain name.

  - The JDBC-ODBC bridge driver

    * [<domain>.]core.persist.jdbc.driver = *sun.jdbc.odbc.JdbcOdbcDriver*

      <domain> - Unqualified model domain name.

    * [<domain>.]core.persist.jdbc.url = *jdbc:odbc:[ODBC data source name]*

      <domain> - Unqualified model domain name.

  - The MySQL Connector/J native-protocol pure Java driver

    * [<domain>.]core.persist.jdbc.driver = *com.mysql.jdbc.Driver*

      <domain> - Unqualified model domain name.

    * [<domain>.]core.persist.jdbc.url = *jdbc:mysql://[hostname]/[database name]*

      <domain> - Unqualified model domain name.

    * [<domain>.]core.persist.jdbc.classpath = *mysql-connector-java.jar*

      <domain> - Unqualified model domain name.

B.5 Object Management Layer Properties

Properties used by the Object Management Layer (OML). Refer to client properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- *core.domains = [<domain>,...]
  This is a required property that specifies a list of object domains that the client application needs to access. The domain com.cdmtech.core.client.corba contains the object classes defined in the base domain model and is required for interaction with systems built using this toolkit. Technically, this is the only required domain, however, this domain defines only abstract object classes and would not be useful for client applications requiring access to object instances (which can only have been produced from concrete classes). Therefore, typically, an application will be configured with at least two additional domains - one domain defining specialized classes and the other containing any user defined data types required by the domain classes. The type domain is not an absolute requirement, but is typically defined.
  <domain> - Fully qualified domain name.

- *<domain>.properties = [<propertyFile>,...]
  For each domain specified in the core.domains property, a set of class properties, contained in separate files, must be defined and referenced. When an object domain model is processed using the UML processing tools (see UML Processing Tools) a set of class properties are generated as part of the general build process. These class property files will be loaded as a system resource, searching the class path specified in the Java runtime used to start the client application.
  <domain> - Fully qualified domain name.
  <propertyFile> - Name of file containing domain class properties.

- *<domain>.serverAPI = <serverAPIClass>
  For each domain specified in the core.domains property, a Java class implementing the com.cdmtech.core.-client.om.ObjectServerAPI interface must be provided. Domains that define distributed object classes must specify the com.cdmtech.core.client.corba.CoreServerAPI. Data type domains must use the com.cdmtech.-core.client.om.NullServerAPI since instances defined from these domains are not distributed objects - they have no remote servant counterpart and therefore have no requirement for a server interface.
  <domain> - Fully qualified domain name.
  <serverAPIClass> - Name of class implementing object server API.

- *<domain>.objectServer = <serverClass>
  For each class domain an object server management implementation class must be specified.
  <domain> - Fully qualified domain name.
  <serverClass> - Name of class implementing object server management functionality.

- <domain>.isCacheable = *true*
  Enable/disable client-side caching
  <domain> - Fully qualified domain name.

- The following properties specify what mechanism to use for cache clearing, and the properties that mechanism should use. These properties must be specified on each domain and the above ’isCacheable’ property must also be set to true for that domain.

- <domain>.resourceManager = <resourceManagerClass>
  Cache resource management implementation.
  <domain> - Fully qualified domain name.
  <resourceManagerClass> - Name of class implementing cache resource management.

- <domain>.resourceManager.retirementAge = *300000*
  When memory clearing begins, objects that have not been read since the retirement age will first be cleared. If this does not meet the deallocation percentage of all objects, the retirement age will be decreased by 1/3 and cache will be cleared with the lower retirement age. If this does not meet the deallocation percentage of all objects, the cache will simply be cleared object by object until the deallocation percentage is met or all objects are cleared. When memory runs low, the frequency of cache clearing will increase, and (approximately) the entire cache will be cleared before an out of Memory Error occurs. Only uncleared objects are counted towards meeting the deallocation percentage requirement.
  <domain> - Fully qualified domain name.

- <domain>.resourceManager.deallocationPercentage = *0.2*
  When a cache clearing is triggered, the percentage (represented as a float 0.0 to 1.0) of all objects that must be cleared. Only uncleared objects are counted towards satisfying the deallocation percentage requirement.
  <domain> - Fully qualified domain name.

- <domain>.resourceManager.memoryThreshold = *0.1*
  A percentage (represented as a float 0.0 to 1.0) indicating at what percentage of maximum memory availability should the resource manager start clearing cached values.
  <domain> - Fully qualified domain name.

• <domain>.<attrType>.class = <attrManagerClass>
For each class/type domain a series of attribute manager plugin classes may be specified. Attribute manager plugin classes implement functionality required to manage specific types of attributes (e.g., array typed attributes, association ends, etc.) If not specified then simple attribute manager plugins will be utilized (e.g., primitive array type support, etc.)
<domain> - Fully qualified domain name.
<attrType> - Attribute type (ATTRIBUTE, ENUMERATION, STRUCT, ASSOCIATION, AGGREGATION, NUMERIC, BOOLEAN, or user defined type).
<attrManagerClass> - Name of class specializing Attribute manager.

  - *<domain>.ATTRIBUTE.class = *com.cdmtech.core.client.om.Attribute*

  - *<domain>.ENUMERATION.class = *com.cdmtech.core.client.om.EnumAttr*

  - *<domain>.STRUCT.class = *com.cdmtech.core.client.om.StructAttr*

  - *<domain>.ASSOCIATION.class = *com.cdmtech.core.client.om.Association*

  - *<domain>.AGGREGATION.class = *com.cdmtech.core.client.om.Aggregation*

  - *<domain>.NUMERIC.class = *com.cdmtech.core.client.om.NumAttr*

  - *<domain>.BOOLEAN.class = *com.cdmtech.core.client.om.BoolAttr*

- <domain>.<attrType>.accessor = <attrAccessorClass>
  For each class domain a series of accessors must be defined for all attribute types.
  <domain> - Fully qualified domain name.
  <attrType> - Attribute type (ATTRIBUTE, ENUMERATION, STRUCT, ASSOCIATION, AGGREGATION, NUMERIC, BOOLEAN, or user defined type).
  <attrAccessorClass> - Name of class implementing attribute accessor.

  – *<domain>.ATTRIBUTE.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.AttributeAccessor*

  – *<domain>.ENUMERATION.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.-EnumAttrAccessor*

  – *<domain>.STRUCT.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.StructAttrAccessor*

  – *<domain>.ASSOCIATION.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.-AssociationSeqAccessor*

  – *<domain>.AGGREGATION.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.-AggregationSeqAccessor*

  – *<domain>.NUMERIC.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.NumAttrAccessor*

  – *<domain>.BOOLEAN.accessor = *com.cdmtech.core.client.mgmt.implementation.core.accessor.BoolAttrAccessor*

- <domain>.<attrType>.memberTypeClass = <memberTypeClass>
  For each class/type domain a series of type manager classes must be defined for all member types.
  <domain> - Fully qualified domain name.
  <attrType> - Attribute type (ATTRIBUTE, ENUMERATION, STRUCT, ASSOCIATION, AGGREGATION, NUMERIC, BOOLEAN, or user defined type).
  <memberTypeClass> - Name of class implementing member type manager class.

  – *<domain>.ATTRIBUTE.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.-membertype.AttributeType*

  – *<domain>.ENUMERATION.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.-membertype.EnumerationType*

  – *<domain>.STRUCT.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.membertype.-AssociationType*

  – *<domain>.ASSOCIATION.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.-membertype.AssociationType*

  – *<domain>.AGGREGATION.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.-membertype.AssociationType*

  – *<domain>.NUMERIC.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.membertype.-AttributeType*

  - *<domain>.BOOLEAN.memberTypeClass = *com.cdmtech.core.client.mgmt.implementation.core.membertype.-EnumerationType*

- Properties used for XML import/export.

  - <domain>.importClassName = <class>*: com.cdmtech.core.client.xml.XMLToPOWImport*
    The XML import class to use for each domain.
    <domain> - Fully qualified domain name.
    <class> - Class name.

  - <domain>.exportClassName = <class>*: com.cdmtech.core.client.xml.POWToXMLExport*
    The XML export class to use for each domain.
    <domain> - Fully qualified domain name.
    <class> - Class name.

- Root.class = <class>

  <class> - Class name.

- *<class>.class.refAttrName = <attrName>

  <class> - Class name.
  <attrName> - Name of reference (key) attribute.

- core.client.om.interestThread = *true*
  Indicates whether interest management will be handled in a separate thread.

- core.client.om.invocationRetries = *5*
  Number of client retries on object remote method invocations.

B.6 JESS Agent Session Properties

Properties used by the JESS agent engine. Refer to utility properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- Core Properties

  - *core.client.agentLoadFile = <agentLoadFile>
    The load file to load constructs
    <agentLoadFile> -
  - core.client.interestsFile = <agentInterestsFile>
    The interests file to register for event notification
    <agentInterestsFile> -

- AgentSession.usePartialTemplates = *true*
  Uses the attribute file defined below to generate partial templates if set to true. To default back to the previous version of the JessAE set this property to false

- AgentSession.includeAttributeFileName = <fileName>
  The file that is to be loaded in NORMAL mode or saved to in INITIALIZE mode to generate partial templates. This file contains class and attribute names being used in agent rules
  <fileName> - Name of file.

- AgentSession.currentMode = <sessionMode>
  The mode in which the Agent Session is started (can be overriden by a system property)
  <sessionMode> - Agent session mode. INITIALIZE mode is used to generate the attributes file and interests file, while NORMAL mode is used for normal execution of the Agent Engine. In DEBUG mode a debug file is generated containing all Fact creations, modifications, and retractions. Default value is NORMAL and can be overridden by specifying system property on the command line when starting the Agent Engine.

  - NORMAL
  - INITIALIZE
  - DEBUG

- AgentSession.useDynamicSubscription = *false*
  Setting this property to false makes use of an interests file to register subscriptions. If not using an interests file, then this property must be set to true. Default value is false.

- AgentSession.avoidClasses = [<class>,...]
  Deftemplates for the given classes and subclasses will not be automatically generated by the AgentSession. Default value is an empty list. So deftemplates for all classes are generated. (deprecated: This property need not be set if you are using partial templates.)
  <class> - Class name.

• Properties useful for debuging

- AgentSession.debugMode = *false*
  Log debug information to file JessAEDebug.xls.

- AgentSession.debugFileName = <fileName>
  The filename to log debug information
  <fileName> - Name of file.

- AgentSession.logAgentMgrInfo = *false*
  Logs AgentManager details such as times taken to process given activations etc, when the log level is set to 'Time'

- AgentSession.logSNMgrInfo = *false*
  Logs SemanticNetManager details such as times taken to process given object events etc, when the log level is set to 'Time'

• Optional properties used by the Agent Engine

- Properties defining extension clases to use in lieu of existing classes. Do not modify these properties unless you know what your are doing. These are only provided as convenience to enable experienced users who may write their own extensions. If you want to use the default classes you do not need to set these properties.

    - AgentSession.agentSessionTypeClassName = <class>
      Agent session implementation class.
      <class> - Class name.

        - AgentSession.agentSessionTypeClassName = *core.client.aml.jess.JessReteImpl*
          Jess Rete Implementation.

    - AgentSession.snMgrClassName = <class>
      Agent session manager implementation class.
      <class> - Class name.

        - AgentSession.snMgrClassName = *core.client.aml.jess.JessSNMgr*
          Jess SNMgr Implementation.

    - AgentSession.agentMgrClassName = <class>
      Agent manager implementation class.
      <class> - Class name.

        - AgentSession.agentMgrClassName = *core.client.aml.jess.JessAgentMgr*
          Jess AgentMgr Implementation.

    - AgentSession.agentSessionClassName = <class>
      Agent session main class.
      <class> - Class name.

        - AgentSession.agentSessionClassName = *core.client.aml.AgentSessionImpl*
          Jess Agent session main class.

    - AgentSession.subscriptionMgrClassName = <class>
      The SubscriptionManager class to use.
      <class> - Class name.

        - AgentSession.subscriptionMgrClassName = *core.client.aml.SubscriptionManager*
          Default SubscriptionManager class. Does not add dynamic subscriptions, but only uses the interests file.

        - AgentSession.subscriptionMgrClassName = *core.client.aml.jess.JessSubscriptionManager*
          Jess SubscriptionManager class. This class performs dynamic subscription management as new rules are added if the property to use dynamic subscriptions is set to true. In addition, if an interest file is provided, additional subscriptions are registered from that file.

    - AgentSession.writeOutputRouters = *false*
      Record all JESS output to log file. This output is recorded to file only if a log file name has been provided in the property core.client.logFile.

- AgentSession.outputDirectory = <directoryName>
  The directory to which the interests file and the attributes file is saved. Do not forget to add the path

up to but not including this directory to your classpath.
<directoryName> - Name of directory.

- core.logFile = <agentLogFile>
  The log file to log debug output. If defined then all AgentSession output is recorded to the given file.
  <agentLogFile> -
- Optional Agent class properties.
  - core.client.agentClassName = <className>
    The Agent class name.
    <className> -
  - core.client.agentIdAttrName = <attrName>
    The Agent identification attribute name.
    <attrName> -
  - core.client.activityAttrName = <attrName>
    The Agent activity attribute name.
    <attrName> -
- Optional container properties.
  - core.client.containerObjectName = <containerName>
    A display name can be specified for the container object instead of objectKey. In case a display name is being provided make sure the display name property is defined for the container class (see below)
    <containerName> - Container name (either object key or display name)
  - core.client.containerClassName = <className>
    Container class name.
    <className> -
  - core.client.collectionRoleName = <roleName>
    Container collection role name.
    <roleName> -
  - <class>.class.disAttrName = <attrName>

    <class> - Class name.
    <attrName> - Display attribute name.
  - core.client.removeCollectablesAtStartup = *true*
    Remove collectables at startup (if a container is defined)
  - core.client.removeCollectablesAtShutdown = *true*
    Remove collectables at shutdown (if a container is defined or will delete all objects created by the AgentSession)

B.7 CLIPS Agent Session Properties

Properties used by the CLIPS agent engine. Refer to utility properties for additional property definitions.

Properties denoted with an asterisk (*) are required.

- Core properties used to initialize an agent session.
  - *core.client.agentLoadFile = <fileName>
    The batch file used to load CLIPS constructs.
    <fileName> - Name of file.
  - *core.client.interestsFile = <fileName>
    The interests file used to register for event notification.
    <fileName> - Name of file.
  - core.client.instancesLoadFile = <fileName>
    A batch file used to load CLIPS COOL instances.
    <fileName> - Name of file.
- Optional session management properties.
  - core.client.containerObjectName = <containerObjectName>
    Unique object name of the container for which to initiate an agent session.
    <containerObjectName> - Unique object name.
  - core.client.containerClassName = <className>
    Name of the container class.
    <className> - Class name.
  - core.client.collectionRoleName = <roleName>
    Collection role name of the container class.
    <roleName> - Association role name.
  - core.client.collectableClassName = <className>
    Name of the collectable class.
    <className> - Class name.
  - core.client.containerRoleName = <roleName>
    Container role name of the collectable class.
    <roleName> - Association role name.
  - core.client.collectableTimestampAttrName = <attrName>
    Name of the integer timestamp attribute of the collectable class.
    <attrName> - Attribute name.
  - core.client.collectableOwnerAttrName = <attrName>
    Name of the owner attribute of the collectable class.
    <attrName> - Attribute name.
  - core.client.collectableOwnerName = <attrValue>*: AGENTS*
    The owner attribute value to use for all collectable objects created by the agent session.
    <attrValue> - Attribute value.

  - core.client.removeCollectablesAtStartup = *false*
    Boolean that indicates whether or not to remove collectables at startup (if a container is defined).

  - core.client.removeCollectablesAtShutdown = *false*
    Boolean that indicates whether or not to remove collectables at shutdown (if a container is defined).

  - core.client.ae.allowNonCollectableObjects = *false*
    Boolean that indicates whether or not to allow objects which are not part of the container's collection (if a container is defined).

  - core.client.ae.systemTimeUpdate = <flag>*: false* | <period>
    Boolean flag that indicates whether or not the system time manager will create and update a 'SYSTEM-TIME' COOL instance, which can be used for time sensitive pattern matching.
    <flag> - Boolean flag.

    * *true* | *enabled* | *on*
    * *false* | *disabled* | *off*

    <period> - Time period (in seconds).

  - core.client.ae.enableCommandServer = *false*
    Boolean flag that indicates whether or not to enable the command server that opens a server socket to listen for commands. The command server is disabled by default.

- Optional agent management properties.

  - core.client.agentClassName = <className>
    The Agent class name.
    <className> - Class name.

  - core.client.agentIdAttrName = <attrName>
    Name of the attribute of the Agent class that indicates the Agent's (possibly unique) identification.
    <attrName> - Attribute name.

  - core.client.ae.agentRunList = <agentIdList>
    List of agent ids that indicate the order in which to run the set of agents.
    <agentIdList> - List of agent ids.

  - core.client.activityAttrName = <attrName>
    Name of the integer attribute of the Agent class that indicates an Agent's activity level.
    <attrName> - Attribute name.

  - core.client.activeAttrName = <attrName>
    Name of the boolean attribute of the Agent class that indicates whether an Agent is active or not.
    <attrName> - Attribute name.

- core.logFile = <fileName>
  The log file to log debug output. If defined then all AgentSession output is recorded to the given file.
  <fileName> - Name of file.

B.8 Translation Service Properties

Properties used by the Interoperability Bridge Framework.

Properties denoted with an asterisk (*) are required.

- Properties used by Interoperability Bridge.

  - InteroperabilityBridge.serviceClassName = <class>*: core.translation.InteroperabilityBridgeImpl*
    The Interoperability Bridge implementation class to use. Default is the InteroperabilityBridgeImpl class. Another option is the RemoteInteroperabilityBridge, which uses the Web services component.
    <class> - Fully qualified class name.

  - InteroperabilityWebService.URL = <URL>*: http://localhost:8080/axis/InteroperabilityWebService*
    The web service URL to use, if using the RemoteInteroperabiliytBridge.
    <URL> - Universal Resource Locator.

- Properties used by Connection Service.

  - <domain>.importClassName = <class>*: core.client.xml.XMLToPOWImport*
    The XML import class to use for each domain. Default is core.client.xml.XMLToPOWImport.
    <domain> - Fully qualified domain name.
    <class> - Fully qualified class name.

  - <domain>.exportClassName = <class>*: core.client.xml.POWToXMLExport*
    The XML export class to use for each domain. Default is core.client.xml.POWToXMLExport.
    <domain> - Fully qualified domain name.
    <class> - Fully qualified class name.

  - <domain>.connectionDelegateClassName = <class>
    Connection delegate class name if using the existing Connector implementation.
    <domain> - Fully qualified domain name.
    <class> - Fully qualified class name.

- Properties used by Translation Service.

  - *<domain>.schemaFile = <schemaFile>
    For each domain model namespace, the schema file to use.
    <domain> - Fully qualified domain name.
    <schemaFile> - Name of schema file.

  - *<domain>.interestedNamespaces = [<domain>,...]
    Comma-delimited list of one or more external domains that this domain is interested in (i.e., requires a translation service for).
    <domain> - Fully qualified domain name.
    <domain> - Fully qualified domain name.

- <domain>To<domain>.xslTransformationFile = <fileName>
  The XSL transformation file to translate from one domain to another. Either this transformation or a JESS-based transformation file must be provided for translation.
  <domain> - Fully qualified domain name.
  <domain> - Fully qualified domain name.
  <fileName> - Name of file.

- <domain>To<domain>.inferenceBasedTransformationFile = <fileName>
  The JESS transformation load file to translate from one domain to another. Either this transformation or an XSL-based transformation file must be provided for translation.
  <domain> - Fully qualified domain name.
  <domain> - Fully qualified domain name.
  <fileName> - Name of file.

- <domain>.includeAttributesFile = <fileName>
  Attributes file used to define partial deftemplates containing only a subset of classes and attributes for systems using JESS-based translation.
  <domain> - Fully qualified domain name.
  <fileName> - Name of file.

- <class>.class.alwaysExport = [<attribute>,...]
  The additional attributes to always export when exporting object modfication events. A class property containing a comma-delimited list of class attributes for each class.
  <class> - Fully qualified class name.
  <attribute> - Class attribute names.

* Properties used by inference-based translation.

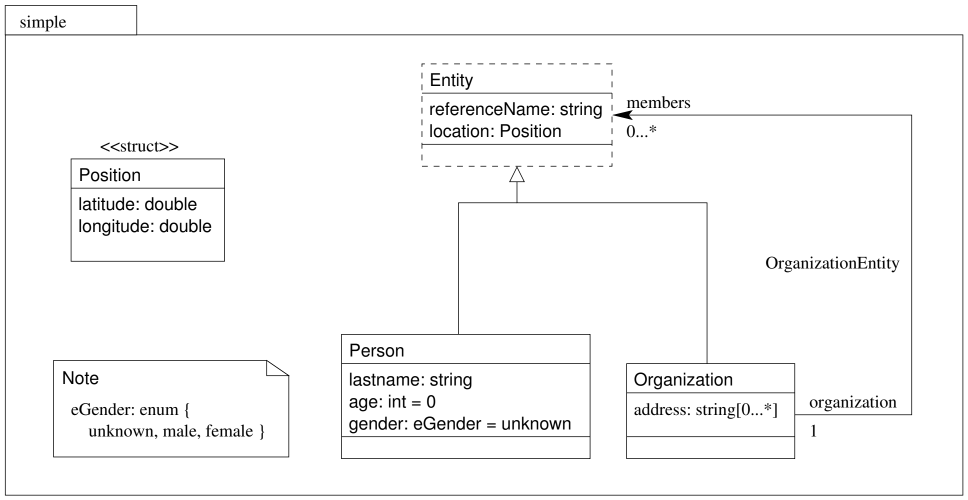
  - <domain>.inferenceBasedImportClassName = <class>
    Inference-based import class name. Required if using JESS-based translation.
    <domain> - Fully qualified domain name.
    <class> - Fully qualified class name.

  - <domain>.inferenceBasedExportClassName = <class>
    Inference-based export class name. Required if using JESS-based translation.
    <domain> - Fully qualified domain name.
    <class> - Fully qualified class name.

Appendix C - Examples

C.1 Example Models

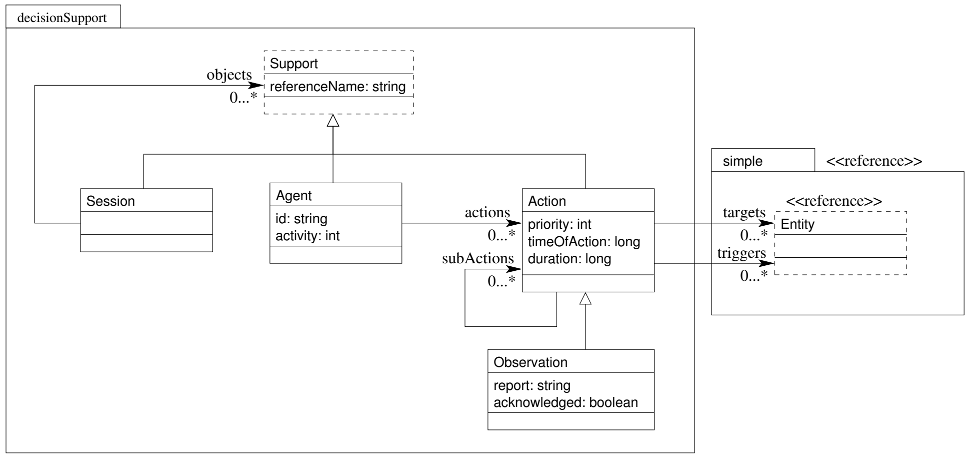
C.1.1 Simple Example model



```
<?xml version="1.0"?>
<XMI xmlns:UML="org.omg.xmi.namespace.UML" xmi.version="1.1">
  <XMI.header>
    <XMI.metamodel xmi.version="1.4" xmi.name="UML"/>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id="ExampleModel" name="exampleModel">
      <UML:Namespace.ownedElement>
        <UML:Package xmi.id="SimplePackage" name="simple"
                     stereotype="Namespace">
          <UML:Namespace.ownedElement>
            <UML:Class xmi.id="EntityClass" name="Entity" isAbstract="true">
              <UML:Classifier.feature>
                <UML:Attribute name="referenceName" visibility="public"
                               changeability="changeable" type="StringType"/>
                <UML:Attribute name="location" visibility="public"
                               changeability="changeable" type="PositionType"/>
              </UML:Classifier.feature>
            </UML:Class>
            <UML:Class xmi.id="PersonClass" name="Person"
                       isAbstract="false" generalization="PersonEntity">
              <UML:Classifier.feature>
                <UML:Attribute name="lastname" visibility="public"
                               changeability="changeable" type="StringType"/>
                <UML:Attribute name="age" visibility="public"
                               changeability="changeable" type="IntType">
                  <UML:Attribute.initialValue>
                    <UML:Expression body="0"/>
                  </UML:Attribute.initialValue>
                </UML:Attribute>
                <UML:Attribute name="gender" visibility="public"
                               changeability="frozen" type="GenderType">
                  <UML:Attribute.initialValue>
                    <UML:Expression body="unknown"/>
                  </UML:Attribute.initialValue>
                </UML:Attribute>
              </UML:Classifier.feature>
              <UML:Namespace.ownedElement>
                <UML:Enumeration xmi.id="GenderType" name="eGender">
                  <UML:Enumeration.literal>
                    <UML:EnumerationLiteral name="unknown"/>
                    <UML:EnumerationLiteral name="male"/>
                    <UML:EnumerationLiteral name="female"/>
                  </UML:Enumeration.literal>
                </UML:Enumeration>
              </UML:Namespace.ownedElement>
            </UML:Class>
            <UML:Class xmi.id="OrganizationClass" name="Organization"
                       isAbstract="false" generalization="OrganizationEntity">
              <UML:Classifier.feature>
                <UML:Attribute name="address" visibility="public"
                               changeability="changeable" type="StringType">
                  <UML:StructuralFeature.multiplicity>
                    <UML:Multiplicity>
                      <UML:Multiplicity.range>
                        <UML:MultiplicityRange lower="0" upper="-1"/>
                      </UML:Multiplicity.range>
                    </UML:Multiplicity>
                  </UML:StructuralFeature.multiplicity>
                </UML:Attribute>
              </UML:Classifier.feature>
            </UML:Class>
            <UML:DataType xmi.id="PositionType" name="Position"
                          stereotype="Struct">
              <UML:Classifier.feature>
                <UML:Attribute name="latitude" visibility="public"
                               changeability="changeable" type="DoubleType"/>
                <UML:Attribute name="longitude" visibility="public"
                               changeability="changeable" type="DoubleType"/>
              </UML:Classifier.feature>
            </UML:DataType>
          </UML:Namespace.ownedElement>
        </UML:Package>
        <UML:Association xmi.id="OrganizationEntityAssoc"
                         name="OrganizationEntity">
          <UML:Association.connection>
            <UML:AssociationEnd name="organization" isNavigable="false"
                                aggregation="none"
                                participant="OrganizationClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
```

```
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="members" isNavigable="true"
                                aggregation="none"
                                participant="EntityClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Generalization xmi.id="PersonEntity" parent="EntityClass"
                            child="PersonClass"/>
        <UML:Generalization xmi.id="OrganizationEntity" parent="EntityClass"
                            child="OrganizationClass"/>
        <UML:DataType xmi.id="StringType" name="string"/>
        <UML:DataType xmi.id="IntType" name="int"/>
        <UML:DataType xmi.id="DoubleType" name="double"/>
        <UML:Stereotype xmi.id="Namespace" name="namespace"/>
        <UML:Stereotype xmi.id="Struct" name="struct"/>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

C.1.2 Decision Support model



```
<?xml version="1.0"?>
<XMI xmlns:UML="org.omg.xmi.namespace.UML" xmi.version="1.1">
  <XMI.header>
```

```
    <XMI.metamodel xmi.version="1.4" xmi.name="UML"/>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id="DecisionModel" name="decisionModel">
      <UML:Namespace.ownedElement>
        <UML:Package xmi.id="SimplePackage" name="simple"
                     stereotype="Reference">
          <UML:Namespace.ownedElement>
            <UML:Class xmi.id="EntityClass" name="Entity"
                       stereotype="Reference"/>
          </UML:Namespace.ownedElement>
        </UML:Package>
        <UML:Package xmi.id="DecisionSupportPackage" name="decisionSupport"
                     stereotype="Namespace">
          <UML:Namespace.ownedElement>
            <UML:Class xmi.id="SupportClass" name="Support" isAbstract="true">
              <UML:Classifier.feature>
                <UML:Attribute name="referenceName" visibility="public"
                               changeability="changeable" type="StringType"/>
              </UML:Classifier.feature>
            </UML:Class>
            <UML:Class xmi.id="SessionClass" name="Session"
                       isAbstract="false" generalization="SessionSupport"/>
            <UML:Class xmi.id="AgentClass" name="Agent"
                       isAbstract="false" generalization="AgentSupport">
              <UML:Classifier.feature>
                <UML:Attribute name="id" visibility="public"
                               changeability="changeable" type="StringType"/>
                <UML:Attribute name="activity" visibility="public"
                               changeability="changeable" type="IntType">
                  <UML:Attribute.initialValue>
                    <UML:Expression body="0"/>
                  </UML:Attribute.initialValue>
                </UML:Attribute>
              </UML:Classifier.feature>
            </UML:Class>
            <UML:Class xmi.id="ActionClass" name="Action"
                       isAbstract="false" generalization="ActionSupport">
              <UML:Classifier.feature>
                <UML:Attribute name="priority" visibility="public"
                               changeability="changeable" type="IntType">
                  <UML:Attribute.initialValue>
                    <UML:Expression body="0"/>
                  </UML:Attribute.initialValue>
                </UML:Attribute>
                <UML:Attribute name="timeOfAction" visibility="public"
                               changeability="changeable" type="LongType"/>
                <UML:Attribute name="duration" visibility="public"
                               changeability="changeable" type="LongType"/>
              </UML:Classifier.feature>
            </UML:Class>
            <UML:Class xmi.id="ObservationClass" name="Observation"
                       isAbstract="false" generalization="ObservationAction">
              <UML:Classifier.feature>
                <UML:Attribute name="report" visibility="public"
                               changeability="changeable" type="StringType"/>
                <UML:Attribute name="acknowledged" visibility="public"
                               changeability="changeable" type="BooleanType"/>
              </UML:Classifier.feature>
            </UML:Class>
          </UML:Namespace.ownedElement>
        </UML:Package>
        <UML:Association xmi.id="SessionSupportAssoc" name="SessionSupport">
          <UML:Association.connection>
            <UML:AssociationEnd name="session" isNavigable="false"
                                aggregation="none"
                                participant="SessionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="objects" isNavigable="true"
                                aggregation="none"
                                participant="SupportClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Association xmi.id="AgentActionAssoc" name="AgentAction">
          <UML:Association.connection>
            <UML:AssociationEnd name="agent" isNavigable="false"
                                aggregation="none"
                                participant="AgentClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="actions" isNavigable="true"
                                aggregation="none"
                                participant="ActionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Association xmi.id="ActionActionAssoc" name="ActionAction">
          <UML:Association.connection>
            <UML:AssociationEnd name="action" isNavigable="false"
                                aggregation="none"
                                participant="ActionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="subActions" isNavigable="true"
                                aggregation="none"
                                participant="ActionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Association xmi.id="ActionTargetAssoc" name="ActionTarget">
          <UML:Association.connection>
            <UML:AssociationEnd name="targetAction" isNavigable="false"
                                aggregation="none"
                                participant="ActionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="targets" isNavigable="true"
                                aggregation="none"
                                participant="EntityClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
```

```
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Association xmi.id="ActionTriggerAssoc" name="ActionTrigger">
          <UML:Association.connection>
            <UML:AssociationEnd name="triggerAction" isNavigable="false"
                                aggregation="none"
                                participant="ActionClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="1" upper="1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
            <UML:AssociationEnd name="triggers" isNavigable="true"
                                aggregation="none"
                                participant="EntityClass">
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange lower="0" upper="-1"/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
        <UML:Generalization xmi.id="SessionSupport" parent="SupportClass"
                            child="SessionClass"/>
        <UML:Generalization xmi.id="AgentSupport" parent="SupportClass"
                            child="AgentClass"/>
        <UML:Generalization xmi.id="ActionSupport" parent="SupportClass"
                            child="ActionClass"/>
        <UML:Generalization xmi.id="ObservationAction" parent="ActionClass"
                            child="ObservationClass"/>
        <UML:DataType xmi.id="StringType" name="string"/>
        <UML:DataType xmi.id="BooleanType" name="boolean"/>
        <UML:DataType xmi.id="IntType" name="int"/>
        <UML:DataType xmi.id="LongType" name="long"/>
        <UML:Stereotype xmi.id="Namespace" name="namespace"/>
        <UML:Stereotype xmi.id="Reference" name="reference"/>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

C.2 Example Build Suites

C.2.1 System build suite

This suite performs the following:

- Generate, compile, and archive (in a JAR) all code required by client interface for example model (if required).
- Generate, compile, and archive (in a JAR) all code required to implement domain services and object servants for example model (if required).
- Generate OML class properties for example model (if required).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../../doc/xsd/SuiteSchema.xsd'
  basepath="../../..">

  <path id="scr.path" ref="*">scr</path>
  <path id="xmi.path" ref="*">example/xmi</path>
  <path id="uml.path" ref="scr.path">uml</path>
  <command id="perl"><exec>perl</exec></command>
  <classpath id="domain.classpath"/>

  <property id="model">
    <path ref="xmi.path"><property ref="model.name"/>.xmi</path>
  </property>

  <property id="seq.type">object</property>
  <property id="domain.name"><property ref="model.name"/></property>

  <select property="seq.type">
    <when match="s.*">
      <property id="assoc.type">Managed</property>
    </when>
    <otherwise>
      <property id="assoc.type"/>
    </otherwise>
  </select>

  <suite id="build">
    <path id="lib.path" ref=".">lib</path>
    <path id="kb.path" ref=".">kb</path>

    <suite id="${model.name}" name="Build System for ${model.name}">
      <description>
        Build system for <property ref="model.name"/> with associations
        managed by
        <select property="seq.type">
          <when match="object">object</when>
          <otherwise>simple</otherwise>
        </select> sequences.
      </description>

      <path id="gen.path" ref=".">gen</path>

      <case id="check" name="Check Model">
        <description>
          Generate model report for <property ref="model.name"/>.
        </description>
        <command ref="perl">
          <exec path="uml.path">makereport.pl</exec>
          <option id="out"><path ref="gen.path"/></option>
          <arg><property ref="model"/></arg>
        </command>
        <artifact path="gen.path"><property ref="model.name"/>.xml</artifact>
      </case>
      <case id="client" name="Generate Client Library">
        <description>
          Build client interface library for <property ref="model.name"/>.
        </description>
        <command ref="perl">
          <exec path="uml.path">makeclient.pl</exec>
          <option id="out"><path ref="lib.path"/></option>
          <option id="gen"><path ref="gen.path"/></option>
          <option id="seq"><property ref="seq.type"/></option>
          <arg><property ref="model"/></arg>
        </command>
        <artifact path="lib.path">
          <property ref="model.name"/>_c.jar
        </artifact>
      </case>
      <case id="server" name="Generate Server Library">
        <description>
          Build service support library for <property ref="model.name"/>.
        </description>
        <command ref="perl">
          <exec path="uml.path">makeserver.pl</exec>
          <option id="out"><path ref="lib.path"/></option>
          <option id="gen"><path ref="gen.path"/></option>
          <option id="seq"><property ref="seq.type"/></option>
          <arg>CLASSPATH=<classpath ref="domain.classpath"/></arg>
          <arg><property ref="model"/></arg>
        </command>
        <artifact path="lib.path">
          <property ref="model.name"/>.jar
        </artifact>
      </case>
      <case id="defclasses" name="Generate COOL Defclasses">
        <description>
          Build COOL defclasses for <property ref="model.name"/>.
        </description>
        <command ref="perl">
          <exec path="uml.path">makecool.pl</exec>
          <option id="out"><path ref="kb.path"/></option>
```

```
          <arg><property ref="model"/></arg>
        </command>
        <artifact path="kb.path"><property ref="model.name"/>.kbc</artifact>
      </case>
      <case id="classproperties" name="Generate Class Properties">
        <description>
          Build class properties for <property ref="model.name"/>.
        </description>
        <command ref="perl">
          <exec path="uml.path">makeproperties.pl</exec>
          <option id="out"><path ref="lib.path"/></option>
          <arg><property ref="model"/></arg>
        </command>
        <artifact path="lib.path">
          <property ref="model.name"/>.properties
        </artifact>
      </case>
      <case id="omlproperties">
        <command ref="perl">
          <exec path="scr.path">makefile.pl</exec>
          <arg>project_om.properties.tpl</arg>
          <arg>
            <path ref="lib.path">
              <property ref="model.name"/>_om.properties
            </path>
          </arg>
          <arg>domain=<property ref="domain.name"/></arg>
          <arg>assocStatus=<property ref="assoc.type"/></arg>
        </command>
        <artifact path="lib.path">
          <property ref="model.name"/>_om.properties
        </artifact>
      </case>
    </suite>
  </suite>
</suite>
```

C.2.2 Simple Example build suite

This suite performs the following:

- Build system for example model (see section C.2.1).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  basepath="../.." id="example">

  <property id="model.name">exampleModel</property>

  <include>include/systemBuild.xml</include>
</suite>
```

C.2.3 Decision Support build suite

This suite performs the following:

- Build system for decision-support model (see section C.2.1).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  basepath="../.." id="example">

  <property id="model.name">decisionModel</property>
  <property id="domain.name">decisionModel,exampleModel</property>

  <include>include/systemBuild.xml</include>
</suite>
```

C.3 Example Execution Suites

C.3.1 Name service suite

This suite performs the following:

- Generate properties for JacORB (if required).
- Startup name service (if required).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../../doc/xsd/SuiteSchema.xsd'>
  <property id="ns.path"><path ref=".">NS_Ref</path></property>
  <property id="ns.url"><url protocol="file" ref=".">NS_Ref</url></property>
  <path id="tpl.path" ref="*">scr/tpl</path>
  <path id="lib.path" ref=".">lib</path>

  <classpath id="boot.classpath">
    <path ref="*">lib/jacorb_v2_1.jar</path>
    <path ref="*">lib/logkit-1.2.jar</path>
    <path ref="*">lib/avalon-framework-4.1.5.jar</path>
    <path ref="*">lib/concurrent-1.3.2.jar</path>
    <path ref="*">lib/antlr-2.7.2.jar</path>
  </classpath>

  <case id="jacorbproperties">
    <command>
      <class>com.cdmtech.core.tool.build.MakeFile</class>
      <arg><path ref="tpl.path">jacorb.properties.tpl</path></arg>
      <arg><path ref="lib.path">jacorb.properties</path></arg>
      <arg>nsref=<property ref="ns.url"/></arg>
    </command>
    <artifact path="lib.path">jacorb.properties</artifact>
  </case>

  <case id="nameserver" name="Name Service">
    <description>
      Launch name service. Executes as a background process after case
      condition met - allowing subsequent cases to execute and interact
      with the service.
    </description>
    <command>
      <classpath><path ref="lib.path"/></classpath>
      <class>org.jacorb.naming.NameServer</class>
      <option>-Xbootclasspath/p:<classpath ref="boot.classpath"/></option>
      <arg><property ref="ns.path"/></arg>
    </command>
    <artifact path=".">nameserver.run</artifact>
    <condition>JacORB V 2.1</condition>
  </case>
</suite>
```

C.3.2 System service startup suite

This suite performs the following:

- Startup name service (if required, see section C.3.1).
- Startup base and domain services (if required).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../../doc/xsd/SuiteSchema.xsd'
  basepath="../../..">

  <path id="xmi.path" ref="*">example/xmi</path>
  <path id="build.path" ref=".">build</path>
  <path id="lib.build.path" ref="build.path">lib</path>
  <classpath id="add.classpath"/>

  <property id="persist.type">serial</property>

  <suite id="exec" name="Base Services for ${model.name}">
    <description>
      Demonstrate basic service startup for
      <property ref="model.name"/> system.
    </description>

    <include>nameServer.xml</include>
```

```
<case id="${model.name}Server" name="Start ${model.name} Server">
  <description>
    Launch basic services. These services include the domain factory
    services, the persistence service, and the notification/subscription
    services. Executes as a background process after case condition met,
    allowing subsequent cases to execute and interact with the services.
  </description>
  <command>
    <classpath ref="add.classpath">
      <path ref="*">lib</path>
      <path ref="*">lib/core_server.jar</path>
      <path ref=".">lib</path>
      <path ref="xmi.path"/>
      <path ref="lib.build.path"/>
      <path ref="lib.build.path"><property ref="model.name"/>_c.jar</path>
      <path ref="lib.build.path"><property ref="model.name"/>.jar</path>
    </classpath>
    <class>StartServer_<property ref="model.name"/></class>
    <property id="org.omg.CORBA.ORBClass">
      org.jacorb.orb.ORB
    </property>
    <property id="org.omg.CORBA.ORBSingletonClass">
      org.jacorb.orb.ORBSingleton
    </property>
    <property id="core.properties">
      <property ref="model.name"/>_server.properties
    </property>
    <select property="persist.type">
      <when match="jdbc">
        <property id="core.persist.class">
          com.cdmtech.core.util.persist.jdbc.JDBCPersistence
        </property>
        <property id="core.persist.jdbc.driver">
          com.mysql.jdbc.Driver
        </property>
        <property id="core.persist.jdbc.classpath">
          mysql-connector-java.jar
        </property>
        <property id="core.persist.jdbc.url">
          jdbc:mysql://localhost/<property ref="model.name"/>
        </property>
      </when>
      <otherwise>
        <property id="core.persist.serial.location">
          <path ref=".">data/<property ref="model.name"/></path>
        </property>
      </otherwise>
    </select>
  </command>
  <artifact path="."><property ref="model.name"/>Server.run</artifact>
  <condition>factories are started and ready</condition>
  <condition type="failure" pattern="EXCEPTION">
    exception occurred during service initialization
  </condition>
```

```
    </case>
  </suite>
</suite>
```

C.3.3 Simple Example service suite

This suite performs the following:

- Build system for example model (if required, see section C.2.2).
- Startup base and domain services for example model (if required, see section C.3.2).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  basepath="../..">

  <include>exampleModelBuild.xxs</include>

  <suite id="example">
    <property id="model.name">exampleModel</property>

    <include>include/systemStart.xml</include>
  </suite>
</suite>
```

C.3.4 Decision Support service suite

This suite performs the following:

- Build system for decision-support model (if required, see section C.2.3).
- Start example model system services (if required, see section C.3.3).
- Startup base and domain services for decision-support model (if required, see section C.3.2).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  basepath="../..">

  <include>decisionModelBuild.xxs</include>
  <include>exampleModelStart.xss</include>

  <suite id="example">
    <path id="build.path" ref=".">build</path>
    <path id="lib.build.path" ref="build.path">lib</path>
    <classpath id="add.classpath">
      <path refpath="lib.build.path">exampleModel_c.jar</path>
    </classpath>

    <property id="model.name">decisionModel</property>

    <include>include/systemStart.xml</include>
  </suite>
</suite>
```

C.3.5 CLIPS Agent Engine suite

This suite performs the following:

- Startup services for decision-support model (if required, see section C.3.4).
- Startup CLIPS based agent engine with example agent loaded (see section C.5).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  name="CLIPS Agent Engine" basepath="../..">
  <description>
    Demonstrate basic system startup and CLIPS Agent Engine example session.
  </description>

  <include>decisionModelStart.xss</include>

  <suite id="example">
    <path id="lib.build.path" refpath=".">build/lib</path>
    <path id="clips.agent.path" refpath=".">agentEngine/clips</path>

    <suite id="exec">
      <case id="clipsagentsession" name="Agent Session">
        <command>
          <class>com.cdmtech.core.client.ae.AgentSession</class>
          <classpath>
            <path>lib</path>
            <path refpath="*">lib</path>
            <path refpath="*">lib/core_ae.jar</path>
            <path refpath=".">lib</path>
            <path refpath="clips.agent.path"/>
            <path refpath="lib.build.path"/>
            <path refpath="lib.build.path">exampleModel_c.jar</path>
            <path refpath="lib.build.path">decisionModel_c.jar</path>
          </classpath>
          <property id="org.omg.CORBA.ORBClass">
            org.jacorb.orb.ORB
          </property>
          <property id="org.omg.CORBA.ORBSingletonClass">
            org.jacorb.orb.ORBSingleton
          </property>
          <property id="core.properties">
            example_clipsae.properties
          </property>
          <property id="core.logLevel">
            Information
          </property>
        </command>
        <condition>waiting for object events</condition>
        <condition type="failure" pattern="ERROR">
          error occurred during agent initialization
        </condition>
      </case>
    </suite>
  </suite>
</suite>
```

C.3.6 Test Execution suite

This suite performs the following:

- Startup services for example model (if required, see section C.3.3).
- Execute client script (JavaScript) to create sample objects (see section C.4).

```
<?xml version="1.0"?>
<?xml-stylesheet href="../../doc/xsd/SuiteSchema.css" type="text/css"?>
<suite
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='../../doc/xsd/SuiteSchema.xsd'
  name="Example System Execution" basepath="../.." exit="10">
  <description>
    Demonstrate basic system startup and simple client execution example.
  </description>

  <include>exampleModelStart.xss</include>

  <suite id="example">
    <path id="lib.build.path" ref=".">build/lib</path>

    <suite id="exec" name="Simple Client">
      <case id="simpleexample" name="Example OML Client Script">
        <description>
          Execute simple example using OML client interface.
        </description>
        <command>
          <class>org.mozilla.javascript.tools.shell.Main</class>
          <classpath>
            <path>lib</path>
            <path ref="*">lib</path>
            <path ref="*">lib/core_support.jar</path>
            <path ref="*">lib/core_om.jar</path>
            <path ref="*">lib/core_client.jar</path>
            <path ref="*">lib/js.jar</path>
            <path ref=".">lib</path>
            <path ref="lib.build.path"/>
            <path ref="lib.build.path">exampleModel_c.jar</path>
          </classpath>
          <property id="org.omg.CORBA.ORBClass">
            org.jacorb.orb.ORB
          </property>
          <property id="org.omg.CORBA.ORBSingletonClass">
            org.jacorb.orb.ORBSingleton
          </property>
          <property id="core.logLevel">
            Information
          </property>
          <property id="core.properties">
            exampleModel_om.properties
          </property>
          <arg><path>include/simpleExample.js</path></arg>
        </command>
      </case>
    </suite>
  </suite>
</suite>
```

C.4 Example Test Script

This script (written in JavaScript/ECMAScript as implemented by the Mozilla JavaScript interpreter - Rhino) performs the following actions:

- Create simple.Organization object and set its referenceName attribute.
- Create two simple.Person objects and set their referenceName attributes.
- Add the simple.Person objects as members to the simple.Organization object.
- Print the attribute and association role values for the simple.Organization object.

```
/**
 *
 */
importPackage(Packages.java.lang);
importPackage(Packages.com.cdmtech.core.client.om);

try {
   organizationTemplate = Template.getTemplate("simple.Organization");

   ourorganization = organizationTemplate.createObject();
   ourorganization.set("referenceName", "our organization");
   ourorganization.add("address", "2975 McMillan Ave., Suite 272");
   ourorganization.add("address", "San Luis Obispo, CA 93401");
   ourorganization.post();
   personTemplate = Template.getTemplate("simple.Person");

   me = personTemplate.createObject();
   me.set("referenceName", "me");
   me.set("gender", "male");
   me.set("age", "44");
   me.set("location", "35\t-120");

   me.post();

   you = personTemplate.createObject();
   you.set("referenceName", "you");
   you.set("gender", "female");
   you.set("age", "38");
   you.set("location.latitude", "35");
   you.set("location.longitude", "-120");

   you.post();

   ourorganization.add("members", me);
   ourorganization.add("members", you);

   positionTemplate = Template.getTemplate("simple.Position");

   ourlocation = positionTemplate.createObject();
   ourlocation.set("latitude", "35");
   ourlocation.set("longitude", "-120");
   ourorganization.set("location", ourlocation);

   ourorganization.post();

   ourorganization.print();
}
catch (e) {
   e.printStackTrace();
   System.exit(1);
}
```

C.5 Example Agents

These examples, written in JESS and CLIPS, illustrate a simple agent that responds to the creation of Organization objects by creating an Observation reporting on the organization size (based on number of members). The following rules are defined:

createMyAgent Creates agent object with id “AGENT1” if one does not exist.

initializeOrganizationFacade Create organization facade object associated to organization object if one does not exist. Set memberSize to number of organization members.

updateOrganizationFacade Update organization facade object if associated organization object changed.

removeOrganizationFacade Remove organization facade object if associated organization object deleted.

createGroupClassReport Asserts group size classification report fact for organization based on member size and group size range.

removeGroupClassReport Retracts group size classification report.

createGroupClassObservation Create group size observation object for created organization.

updateGroupClassObservation Update group size observation object for modified organization.

deleteGroupClassObservation Delete group size observation object for deleted organization.

C.5.1 JESS-based Agent

```
(deffacts AGENT1::BaseKnowledge
  (GROUP-CLASS-RANGE SMALL 0 9)
  (GROUP-CLASS-RANGE MEDIUM 10 99)
  (GROUP-CLASS-RANGE LARGE 100 -1)
)

(deftemplate AGENT1::OrganizationFacade
  (slot organization)
  (slot memberSize (type INTEGER))
)

(defrule AGENT1::createMyAgent
  (declare (salience 10000))
  (not (decisionSupport.Agent (id "AGENT1")))
=>
  (bind ?agent (make-instance decisionSupport.Agent
    (referenceName "Agent #1")
    (id "AGENT1")
  ))
)

(defrule AGENT1::initializeOrganizationFacade
  ?org <- (simple.Organization (name ?orgName))
  (not (OrganizationFacade (organization ?orgName)))
=>
  (assert (OrganizationFacade
    (organization ?orgName)
    (memberSize (length$ (send ?org get-members)))
  ))
)

(defrule AGENT1::updateOrganizationFacade
  ?org <- (simple.Organization (name ?orgName)
    (members $?memList))
  ?orgFacade <- (OrganizationFacade
    (organization ?orgName)
    (memberSize ?memSize)
  )
  (test (<> ?memSize (length$ $?memList)))
=>
  (retract ?orgFacade)
  (assert (OrganizationFacade
    (organization ?orgName)
    (memberSize (length$ $?memList))
  ))
)

(defrule AGENT1::removeOrganizationFacade
  (declare (salience 5000))
  ?orgFacade <- (OrganizationFacade (organization ?orgName))
  (not (simple.Organization (name ?orgName)))
=>
  (retract ?orgFacade)
)

(defrule AGENT1::createGroupClassReport
  (GROUP-CLASS-RANGE ?gc ?min ?max)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  (test (and (>= ?memSize ?min) (or (< ?max 0) (<= ?memSize ?max))))
  (not (GROUP-CLASS ? ?orgName))
=>
  (assert (GROUP-CLASS ?gc ?orgName))
)

(defrule AGENT1::removeGroupClassReport
  (declare (salience 1000))
  (GROUP-CLASS-RANGE ?gc ?min ?max)
  ?gcf <- (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  (test (or (< ?memSize ?min) (and (> ?max 0) (> ?memSize ?max))))
=>
  (retract ?gcf)
)

(defrule AGENT1::createGroupClassObservation
  ?agent <- (decisionSupport.Agent (id "AGENT1"))
  (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName))
  ?org <- (simple.Organization (name ?orgName))
  (not (decisionSupport.Observation
    (targets $? ?orgName $?)
  ))
=>
  (bind ?alert (make-instance decisionSupport.Observation
    (referenceName
      (str-cat "Observation on organization " (send ?org get-referenceName)))
  ))
  (send ?alert add-assoc targets ?org)
  (send ?alert add-assoc agent ?agent)
)

(defrule AGENT1::updateGroupClassObservation
  (decisionSupport.Agent (name ?agentName) (id "AGENT1"))
  (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  ?org <- (simple.Organization (name ?orgName))
  ?alert <- (decisionSupport.Observation
    (agent ?agentName)
    (targets $? ?orgName $?)
    (report ?report)
  )
  (test (not (str-index (str-cat "with " ?memSize " members") ?report)))
=>
  (bind ?alertMsg (str-cat "Organization '" (send ?org get-referenceName)
      "' is classified as " ?gc " with " ?memSize " members"))
  (send ?alert update-slot report ?alertMsg)
)

(defrule AGENT1::deleteGroupClassObservation
  (decisionSupport.Agent (name ?agentName) (id "AGENT1"))
  ?gcf <- (GROUP-CLASS ?gc ?orgName)
  ?alert <- (decisionSupport.Observation
    (agent ?agentName)
    (targets $? ?orgName $?)
  )
  (not (OrganizationFacade (organization ?orgName)))
=>
  (retract ?gcf)
  (send ?alert delete)
)
```

C.5.2 CLIPS-based Agent

```
(deffacts AGENT1::BaseKnowledge
  (GROUP-CLASS-RANGE SMALL 0 9)
  (GROUP-CLASS-RANGE MEDIUM 10 99)
  (GROUP-CLASS-RANGE LARGE 100 -1)
)

(deftemplate AGENT1::OrganizationFacade
  (slot organization)
  (slot memberSize (type INTEGER))
)

(defrule AGENT1::createMyAgent
  (declare (salience 10000))
  (not (object (is-a decisionSupport.Agent) (id "AGENT1")))
=>
  (bind ?agent (make-instance of decisionSupport.Agent
    (referenceName "Agent #1")
    (id "AGENT1")
  ))
)

(defrule AGENT1::initializeOrganizationFacade
  ?org <- (object (is-a simple.Organization) (name ?orgName))
  (not (OrganizationFacade (organization ?orgName)))
=>
```

```
  (assert (OrganizationFacade
    (organization ?orgName)
    (memberSize (length$ (send ?org get-members)))
  ))
)

(defrule AGENT1::updateOrganizationFacade
  ?org <- (object (is-a simple.Organization) (name ?orgName)
    (members $?memList))
  ?orgFacade <- (OrganizationFacade
    (organization ?orgName)
    (memberSize ?memSize)
  )
  (test (<> ?memSize (length$ $?memList)))
=>
  (retract ?orgFacade)
  (assert (OrganizationFacade
    (organization ?orgName)
    (memberSize (length$ $?memList))
  ))
)

(defrule AGENT1::removeOrganizationFacade
  (declare (salience 5000))
  ?orgFacade <- (OrganizationFacade (organization ?orgName))
  (not (object (is-a simple.Organization) (name ?orgName)))
=>
  (retract ?orgFacade)
)

(defrule AGENT1::createGroupClassReport
  (GROUP-CLASS-RANGE ?gc ?min ?max)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  (test (and (>= ?memSize ?min) (or (< ?max 0) (<= ?memSize ?max))))
  (not (GROUP-CLASS ? ?orgName))
=>
  (assert (GROUP-CLASS ?gc ?orgName))
)

(defrule AGENT1::removeGroupClassReport
  (declare (salience 1000))
  (GROUP-CLASS-RANGE ?gc ?min ?max)
  ?gcf <- (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  (test (or (< ?memSize ?min) (and (> ?max 0) (> ?memSize ?max))))
=>
  (retract ?gcf)
)

(defrule AGENT1::createGroupClassObservation
  (object (is-a decisionSupport.Agent) (name ?agentName) (id "AGENT1"))
  (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName))
  ?org <- (object (is-a simple.Organization) (name ?orgName))
  (not (object (is-a decisionSupport.Observation)
    (targets $? ?orgName $?)
  ))
=>
  (bind ?alertName (make-instance of decisionSupport.Observation
    (referenceName
      (str-cat "Observation on organization " (send ?org get-referenceName)))
  ))
  (bind ?alert (InstanceAddress ?alertName))
  (send ?alert add-assoc targets ?orgName)
  (send ?alert add-assoc agent ?agentName)
)

(defrule AGENT1::updateGroupClassObservation
  (object (is-a decisionSupport.Agent) (name ?agentName) (id "AGENT1"))
  (GROUP-CLASS ?gc ?orgName)
  (OrganizationFacade (organization ?orgName) (memberSize ?memSize))
  ?org <- (object (is-a simple.Organization) (name ?orgName))
  ?alert <- (object (is-a decisionSupport.Observation)
    (agent ?agentName)
    (targets $? ?orgName $?)
    (report ?report)
  )
  (test (not (str-index (str-cat "with " ?memSize " members") ?report)))
=>
  (bind ?alertMsg (str-cat "Organization '" (send ?org get-referenceName)
      "' is classified as " ?gc " with " ?memSize " members"))
  (send ?alert update-slot report ?alertMsg)
)

(defrule AGENT1::deleteGroupClassObservation
  (object (is-a decisionSupport.Agent) (name ?agentName) (id "AGENT1"))
  ?gcf <- (GROUP-CLASS ?gc ?orgName)
  ?alert <- (object (is-a decisionSupport.Observation)
    (agent ?agentName)
    (targets $? ?orgName $?)
  )
  (not (OrganizationFacade (organization ?orgName)))
=>
  (retract ?gcf)
  (send ?alert delete)
)
```

Appendix D - Execution Framework

D.1 Introduction

The execution framework, provided as part of the toolkit, serves as a system process description language used to both control system execution as well as to provide structured documentation. The framework is built around a logical model (shown in figure D.1) and implemented using an XML schema with supporting processing tools (classes that implement the framework model and logic required to manage system process execution as defined by XML documents based on the framework schema).

An execution case defines a discrete process as an arbitrary command (understood by the underlying operating system) or as a Java main class. Execution cases may define conditions which upon successfully meeting the case will complete allowing subsequent execution cases to process. Once all success conditions are met, with no failure conditions, the case execution process will continue to execute asynchronously. An execution suite extends the logical definition of an execution case to include definition of logical case groups. Execution suites in and of themselves do not define processes but instead act as containers for collections of cases which when executed simply results in the execution of each contained case (which may also be nested suites) in sequence. The full details describing the elements defined within the execution framework schema may be found in section D.4.

D.2 Execution Suite Processor

Given a well defined suite XML document the provided suite processor may be used to execute the suite. Once the suite has completed, the results become an integral part of the suite document (through use of an XSL transform) aiding the documentation effort. The suite processor may be invoked using the core_support.jar directly. Command line usage takes the following form:

```
java -jar [install location]/lib/core_support.jar -g -q -d -a↩
  -o [output location] <XML suite document>
```

-g | –gui Use this option to present a graphical display of suite execution status.

-q | –quiet Use this option to turn off all output to the console.

-d | –deprecated Use this option to turn on output to the console (assuming the **-q** option is not specified) showing document use of deprecated features along with suggested replacements.

-a | –all Fire all status events.

-o | –out [output location] Use this option to specify an alternative suite result output location. If not specified then suite results will be output to suite base path (path defined by suite basepath attribute in suite document as a relative path from the suite document location).

Note: To remain location independent it is recommended that relative paths always be used (in fact the suite schema promotes it). The execution of suites that are location independent must be started from a well known location (i.e., the location the suite paths are relative to). The suites provided as examples all define paths that are relative to the install location. For example, to execute the example model build suite contained in the ***[install location]/example/suite*** directory perform the following:

- ⇒java -jar [install location]/lib/core_support.jar -qg [install location]/example/suite/exampleModelBuild.xxs

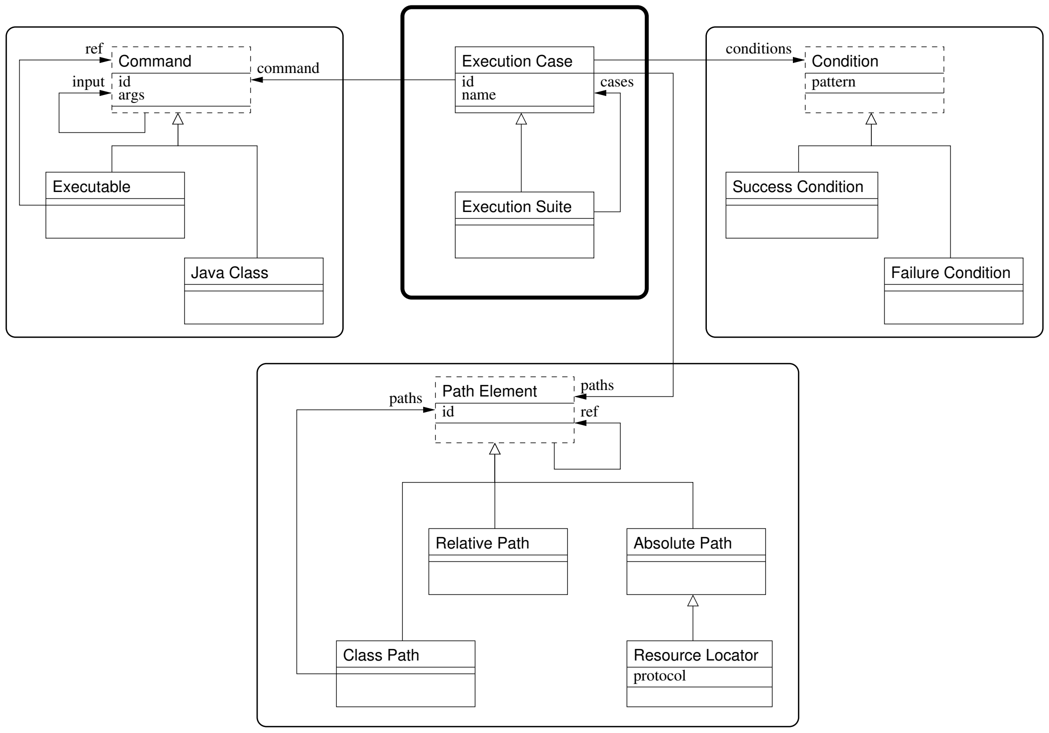


Figure D.1: The Execution Framework Logical Model

Alternatively, suite documents may be located independent of the software installation location. Any files that are not locatable as system resources (e.g., paths referenced to "*") must be located relative to the suite documents that require them. All resources defined within ***classpath*** and ***command*** elements are locatable as system resources. For example, to execute the example suites from another location perform the following:

- ⇒copy [install location]/example to [other location]
- ⇒copy [install location]/kb to [other location]
- ⇒java -jar [install location]/lib/core_support.jar -qg [other location]/example/suite/exampleModelBuild.xxs

Note: The ***kb*** directory was also copied to the new location. The files contained in this directory are required by the agents defined in the examples and are not locatable as system resources.

If an alternative suite result output location is specified, using the **-o** option, then the generated suite result and captured execution process output will be placed at this alternate location independant of the suite document location. This option may be used to isolate suite results and allows for multiple suite executions retaining the results for each execution separately. This can be useful, for example, in the execution of test suites where results from prior runs are required for regression analysis.

Additionally, specific file types are understood by the suite processor. These file types, along with any special treatment, include the following:

xxs XML Execution Suite - defines a general execution suite.

xss XML Service Suite - defines an execution suite for service startup.

xcs XML Client Suite - defines an execution suite for client application startup. No status display will be presented even if **-g** option is specified.

If the display environment supports associating open actions to file types then these file types could be associated to the suite processor application allowing for convenient suite execution.

D.3 Execution Suite Presentation

Since execution suites, written to conform to the supplied schema, are stored as XML it is a fairly simple matter to apply a transform (e.g., in XSLT) to provide a presentable form of the document. Along with the execution suite processor, an XSL transform is provided that allows presentation of the suite document in HTML suitable for display in any Web browser. In fact, any browser that understands XSL stylesheets may be used to display the document directly (i.e., the browser applys the transform defined by the specified XSL stylesheet). For example, if the XML process instruction, ***<?xml-stylesheet href="../../doc/xsd/SuiteToHTML.xsl" type="text/xsl"?>*** is included in the document preamble then, assuming the XSL transform ***SuiteToHTML.xsl*** is located at the specified (relative) path, the document should be directly presentable through an XML aware browser.

Additionally, processing of suites (through the suite processor) that directly contain execution cases will have result output generated in the form of an XML document. This document follows a schema very similar to the execution suite document schema with some differences allowing for additional output data (e.g., timing results, target component versions, etc.) These suite result output documents may also be presented through use of an XSL transform. In fact, if the status display, for the suite processor, is turned on (i.e., the **-g** option is specified) then upon completion of a suite (successful or otherwise) the result output may be displayed by selecting the suite area in the status display. The result output is presented using an XSL transform (***SuiteOutput.xsl***) for display in HTML. Also, note that selection of a case indicator icon will present the captured case output (the verbatum standard output for successful case execution or error output for unsuccessful case execution).

Since the execution suite documents are XML, the documents may be created and modified through use of an XML editor. XML editors that also understand XML Schema can also aid in the development of a document by constraining the document structure based on the schema. As an example, figure D.2 shows a display of an XML execution suite as presented in the Morphon XML Editor[1]. This editor also makes use of Cascading Style Sheet (CSS) definitions to form the document display. The CSS file used is included with the toolkit.

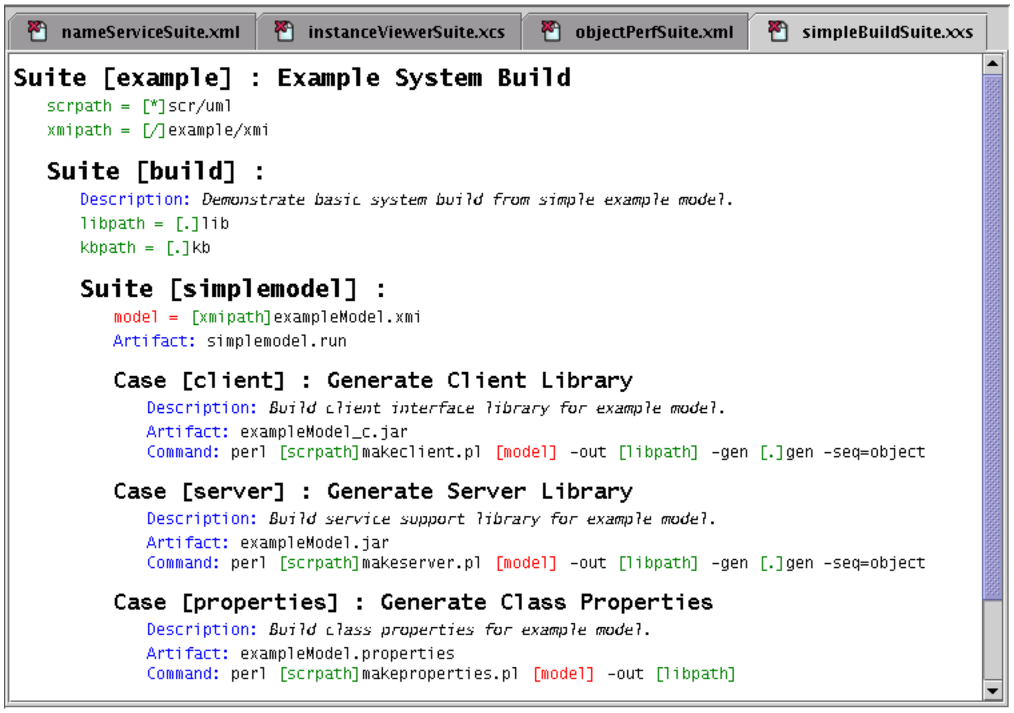
D.4 Execution Suite Schema

D.4.1 suite

Description

- A suite defines a sequence of inter-dependant cases and suites.
- Execution of a suite will result in an ordered (document order) execution of each contained case and suite (recursive). Execution will fail if any case condition is not met. If a suite is set to exit then, upon completion (successful or otherwise), each contained case and suite will be exitted in reverse document order. Path and classpath elements may be defined and will be applicable within the scope of the suite or any contained cases or suites. External suite documents may also be included with all in-scope path and classpath elements accessible. A directory will be created for a suite contained in the directory of any containing suite (or at the current working directory in the case of the root suite).
- A suite document must contain a root <suite> element. <suite> elements must include an 'id' attribute if the suite is concrete. A <suite> element may optionally (highly suggested) include a more descriptive 'name' attribute. A <suite> defined without an 'id' attribute will be treated as a virtual element, inheriting all characteristics from its' parent but providing no additional structure. The primary use for a virtual suite would be in the definition of reuseable suites that may be included (see <include> element description below) in a core suite document.

[1]The Morphon XML Editor, while no longer supported, is available for free download from http://www.morphon.com.



```
Suite [example] : Example System Build
   scrpath = [*]scr/uml
   xmipath = [/]example/xmi

   Suite [build] :
      Description: Demonstrate basic system build from simple example model.
      libpath = [.]lib
      kbpath = [.]kb

      Suite [simplemodel] :
         model = [xmipath]exampleModel.xmi
         Artifact: simplemodel.run

         Case [client] : Generate Client Library
            Description: Build client interface library for example model.
            Artifact: exampleModel_c.jar
            Command: perl [scrpath]makeclient.pl [model] -out [libpath] -gen [.]gen -seq=object

         Case [server] : Generate Server Library
            Description: Build service support library for example model.
            Artifact: exampleModel.jar
            Command: perl [scrpath]makeserver.pl [model] -out [libpath] -gen [.]gen -seq=object

         Case [properties] : Generate Class Properties
            Description: Build class properties for example model.
            Artifact: exampleModel.properties
            Command: perl [scrpath]makeproperties.pl [model] -out [libpath]
```

Figure D.2: Execution Suite Document Displayed in XML Editor.

- <suite> elements may be nested within other <suite> elements. This allows for the grouping of suites comprising cases that constitute a logical unit. Additionally, it provides a segregation of case results organized by suite hierarchy (e.g., all output captured from case processes will be placed into directories arranged according to the suite structure). In addition to the attributes described above for the <suite> element the following example includes the 'exit' attribute. Use of this attribute implies that upon successful completion of this suite's cases that any running processes should be terminated. The numeric value specifies a delay (in seconds) prior to a forced exit. Inclusion of this attribute allows for definition of suites requiring subsequent processing after condition satisfaction (presumably by a succeeding suite).

- Example: <suite id="examplesuite" name="Example Suite" exit="10" basepath=".."> ... </suite>

Attributes

- **id**[string] - Suite identifier.
- **name**[string] - Descriptive name.
- **type** - Execution type.
  - **conditional** - [default] Case will execute on condition of previous case success and failure will prevent further execution.
  - **critical** - Case will execute unconditionally and failure will prevent further execution.
  - **noncritical** - Case will execute on condition of previous case success and failure will not prevent further execution.
  - **fault** - Case will execute on condition of previous case success and nonfailure of case will prevent further execution.

  - **user** - Case will execute on user initiation and on condition of previous case success. Subsequent cases will execute independant of this case execution.

- **exit**[integer] - After successful completion exit after specified delay.
- **basepath**[string] - Base path establishing root reference for suite (should be defined as a relative path from document).
- **version**[string] - The required processor version for this suite.
- **output**[string] - Name of suite result output (no path or file type). If not defined, then implies no result output for suite.

* indicates required attribute.

Elements

- (description D.4.17 source D.4.5 artifact D.4.6 path D.4.3 url D.4.4 classpath D.4.11 property D.4.15 target D.4.16 requirement D.4.20 command D.4.7 include D.4.10 case D.4.2 suite D.4.1 select D.4.21) [0...*]

D.4.2 case

Description

- A case defines a process startup and condition for successful execution. Processes may be defined as an arbitrary command or as a Java based application. Case processes may have multiple arguments that may reference (and extend) path or classpath elements defined within scope. General case processes must define a command element containing the process startup (that may be any external command) and may reference a path element (in preference to specification of a path as part of command line). Java case processes must define a class element containing the fully qualified name of the application main class. Java cases may also specify (or reference/extend) a classpath for use by the Java runtime. Additionally, properties may be specified that will be included as system properties to the Java runtime.
- Successful completion of a case is defined as either a successful exit status (0 exit code) or as a condition defined as a pattern (Java 1.4 regular expression) match on the process standard output stream. The exit status condition is the default (no explicit condition element defined) and implies that the process will execute syncronously (i.e., subsequent cases will not execute until the current case executes to successful completion). The pattern match condition is specified through the condition element and implies that the process will execute syncronously until the pattern is matched at which time the process will fork and continue to operate asyncronously allowing subsequent cases to execute. If the pattern is not matched and the process never exits then the overall case will appear to "hang" and no other cases will execute. If the process exits and the condition pattern is not matched then the case will fail. All standard output for a process is captured in a file named after the case id with ".out" appended. The standard error is also captured in a file named after the case id with ".err" appended. Both files are placed into the containing suite directory.
- Example: <case id="jacorbproperties" name="Generate JacORB Properties"> ... </case>

Attributes

- * **id**[string] - Case identifier.
- **name**[string] - Descriptive name.
- **type** - Execution type.

  - **conditional** - [default] Case will execute on condition of previous case success and failure will prevent further execution.
  - **critical** - Case will execute unconditionally and failure will prevent further execution.
  - **noncritical** - Case will execute on condition of previous case success and failure will not prevent further execution.
  - **fault** - Case will execute on condition of previous case success and nonfailure of case will prevent further execution.
  - **user** - Case will execute on user initiation and on condition of previous case success. Subsequent cases will execute independant of this case execution.

* indicates required attribute.

Elements

- (description D.4.17 source D.4.5 artifact D.4.6 requirement D.4.20 command D.4.7 condition D.4.14 select D.4.21) [0...*]

D.4.3 path

Description

- The path element is used to define path-like structures. Path elements may reference another path element to represent the path root basis (note that only the directory part of referenced paths will be used). Any additional content provided will extend the root basis. Paths defined directly within the scope of a case or suite must specify an id. Paths used within ancillary elements (e.g., arg, etc) can only reference path elements defined within any enclosing case or suite. In either case path elements may extend referenced path elements. Built-in paths ('.', '..', '/', '*') may be referenced. The '.' path represents the path to the current suite scope. The '..' path represents the path to the enclosing (parent) suite scope (or current working directory in the case of the root suite). The '/' path represents the path to the current working directory. The '*' path indicates that the defined path will be searched for as a system resource (i.e., the relative path must be located somewhere in the resource search path). If no reference path is specified, then the path will be defined relative to the document location.
- Example: <path id="projectlib" ref=".">lib</path>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).
- **type** - Path resolution type.
  - **relative** - [default] Specified path relative to base.
  - **absolute** - Specified path converted to absolute (path specified as relative to base).
- **ref**[string] - Reference path identifier.

* indicates required attribute.

Elements

- (path D.4.3 property D.4.15) [0...*]

D.4.4 url

Description

- The url element extends path through inclusion of ’protocol’ and ’host’ attributes providing a path-like structure in a form tagged for use as a Uniform Resource Locator (URL).
- Example: <arg>nsref=<url protocol="file" ref=".">NS_Ref</url></arg>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).
- **type** - Path resolution type.
  - **relative** - [default] Specified path relative to base.
  - **absolute** - Specified path converted to absolute (path specified as relative to base).
- **ref**[string] - Reference path identifier.
- **protocol**[string] - URL protocol (may be ’file’, ’http’, ’ftp’, ’mailto’).
- **host**[string] - Machine host name where resource is physically located. May be required dependant on selected protocol.

* indicates required attribute.

Elements

- (path D.4.3 property D.4.15) [0...*]

D.4.5 source

Description

- Path to case source artifact.
- Required case source artifacts may be specified through the <source> element. A case will not be processed if all of its output artifacts exist and are younger than any source artifacts. A previously defined path may be referenced through the ’path’ attribute (see <path> element).
- Example: <source path="xmipath">basicModel.xmi</source>

Attributes

- **path**[string] - Reference path identifier.

* indicates required attribute.

Elements

- property D.4.15 [0...*]

D.4.6 artifact

Description

- Path to case output artifact.
- Required case output artifacts may be specified through the <artifact> element. The case will fail if any specified output artifact is not created as a result of case execution. Additionally, a case will not be processed if all of its artifacts exist and are younger than any source artifacts. A previously defined path may be referenced through the 'path' attribute (see <path> element).
- Example: <artifact path="projectlib">jacorb.properties</artifact>

Attributes

- **path**[string] - Reference path identifier.

* indicates required attribute.

Elements

- property D.4.15 [0...*]

D.4.7 command

Description

- Used for definition of an execution command.
- General commands as well as Java execution classes may be specified through the <command> element. A previously defined command may be referenced through the 'ref' attribute. Referenced commands will be treated as command interpreters (i.e., commands that process other commands). An example of such a command is 'perl,' where the Perl executable is a command that is used to interpret (process) scripts written in the Perl language. Additionally, a command may be referenced using the 'input' attribute to define a command input chain. The output of this referenced command will be used as input, providing a command input/output pipe. Commands chained together using this input/output pipe will be executed as a sequence of commands linking their respective input and output streams. Referenced commands must be defined prior to referencing and must be within scope (i.e., defined within the immediate case or any enclosing suite).
- Example: <command id="update" ref="perl">...

Attributes

- **id**[string] - Command identifier used for subsequent reference (should be unique within scope of usage).
- **ref**[string] - Reference command identifier.
- **input**[string] - Identifier of command providing input.

* indicates required attribute.

Elements

- description D.4.17 [0...*]
- classpath D.4.11 [0...1]
- (exec D.4.8 class D.4.9) [0...1]
- (property D.4.15 option D.4.13 arg D.4.12 select D.4.21) [0...*]

D.4.8 exec

Description

- Used for definition of a path to an external command to be executed. Note that a command specified using the <exec> element is passed as a single command string even if the string contains spaces. In effect this is equivilent to passing the complete command string surrounded by quotes. Do not pass arguments or options in an <exec> element as unexpected command string interpretation will result.
- General commands may be specified through the <exec> element. A previously defined path may be referenced through the ’path’ attribute (see <path> element).
- Example: <exec path="scr.path">update.pl</path>

Attributes

- **path**[string] - Reference path identifier.

* indicates required attribute.

Elements

- property D.4.15 [0...*]

D.4.9 class

Description

- Name of Java class implementing main entry for case process. Do not pass arguments or options in a <class> element as unexpected class string interpretation will result.
- If no classpath is specified within the command scope of the class then the class will execute within the processor virtual machine (i.e., will not be executed as a separate process) and will be loaded using a class loader inheriting the parent loader resources.
- Example: <class>org.jacorb.naming.NameServer</class>

Elements

- property D.4.15 [0...*]

D.4.10 include

Description

- The <include> element is used to specify an external suite document for inclusion. The contents of the file are treated as if the defined elements were specified directly within the context of the <include> element. The included document must specify a root <suite> element that may be virtual (i.e., no 'id' attribute defined) in which case the contents of the <suite> will treated as the contents of the current suite. A previously defined path may be referenced through the 'path' attribute (see <path> element). If no reference path is specified then the path will be relative to the document location.

- Example: <include>common/omlTestSuite.xml</include>

Attributes

- **path**[string] - Reference path identifier.

* indicates required attribute.

Elements

- property D.4.15 [0...*]

D.4.11 classpath

Description

- Used to define a group of paths used for class search (only useful for Java based case processes). <classpath> elements may include an 'id' attribute when used within the scope of a suite. A <classpath> element may reference another <classpath> element using the 'ref' attribute with additional <path> elements extending the referenced <classpath> element.

- Example: <classpath><path ref="projectlib"/><path ref="libpath">jacorb_v1_4_1.jar</path></classpath>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).

- **ref**[string] - Reference classpath identifier.

* indicates required attribute.

Elements

- (path D.4.3 classpath D.4.11) [0...*]

D.4.12 arg

Description

- Used for definition of a single argument string to be passed to case process. Note that arguments specified using <arg> elements are passed as single arguments even if the argument value contains spaces. In effect, this is equivilent to passing the complete argument string surrounded by quotes. Do not pass multiple arguments in a single <arg> element as unexpected argument interpretation will result.
- <path> elements may be used within an <arg> element to provide an argument value requiring a path. Note that this feature may be used to constrain use of ancillary files (whether they are used for input or output) to paths that are referenced to the containing suite context thereby enabling isolation of case execution (hopefully providing less chance of a collision with input/output used in other cases).
- Example: <arg>-o</arg> <arg><path ref="..">diffs1.out</path></arg>

Elements

- (path D.4.3 url D.4.4 classpath D.4.11 property D.4.15) [0...*]

D.4.13 option

Description

- Used for definition of a single option string to be passed to case process. Note that options specified using <option> elements are passed as single option values even if the value contains spaces. In effect, this is equivilent to passing the complete option string surrounded by quotes. Do not pass multiple options in a single <option> element as unexpected option interpretation will result.
- <option> elements are similar to <arg> elements with the exception of the definition of an option key. If specified, the option key ('id') will be used to identify the specific option to the command. The option will be added to the command line with its key prepended with a '-' and its value (if given) added immediately following a space. If the option key is not given then the <option> element is equivilent to the <arg> element with the exception that all options will appear before arguments on the command line.
- Example: <option id="d"><path ref=".">client</path></option>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).

* indicates required attribute.

Elements

- (path D.4.3 url D.4.4 classpath D.4.11 property D.4.15) [0...*]

D.4.14 condition

Description

- Condition string defining successful case completion. Use of a <condition> element implies that the case process is to be executed as a background process and that completion of the case is considered successful once the condition pattern (Java 1.4 regular expression) is matched in the process standard output (by default the pattern defines any substring match). If the condition starts with '!' then the pattern must not be matched for the condition to be satisfied. Once the condition is satisfied subsequent cases will be allowed to execute with the current case process forked to the background (i.e., the process continues to execute to completion, or otherwise, independant of the parent process).
- Example: <condition pattern="RootPOA - ready">access to base services ready</condition>

Attributes

- **type** - Condition type indicating expected success (default: success).
  - **success** - [default] Case is expected to satisfy condition for success.
  - **failure** - Satisfaction of any condition implies case failure.
- **pattern**[string] - Condition satisfaction based on pattern match. Pattern may be specified using Java 1.4 regular expression syntax. If pattern starts with '!' then condition will fail if pattern matched. If pattern specified as simple string then the pattern will match if string occurs as a substring.

* indicates required attribute.

D.4.15 property

Description

- Used for definition of a single property name/value pair for subsequent reference. If used within a case command that defines a Java process, the property will be passed as a system property. If a property is defined multiple times within nested suites the value defined in the outermost suite will take precedent (system properties passed to the top-level execution process have highest precedent).
- Example: <property id="core.properties">test_server.properties</property>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).
- **ref**[string] - Reference property identifier.

* indicates required attribute.

Elements

- (path D.4.3 url D.4.4 classpath D.4.11 property D.4.15 description D.4.17) [0...*]

D.4.16 target

Description

- Used for definition of suite targeted component (i.e., a component that plays a significant role in the execution suite). Inclusion will result in component information being output to the result output.
- Example: <target id="core_client"/>

Attributes

- **id**[string] - Identifier used for subsequent reference (should be unique within scope of usage).

* indicates required attribute.

D.4.17 description

Description

- Description elements may be included within any primary elements to provide inline documentation. This documentation may also be used in a visual presentation of the case definitions (through the use of an XSL transform to HTML for example). The use of description elements is highly encouraged.
- Example: <description>Test suite configured to use the notification service for subscription and event delivery.</description>

Elements

- (link D.4.18 image D.4.19 property D.4.15) [0...*]

D.4.18 link

Description

- Link elements may be used within descriptions to provide linked references to external documentation. The link resource reference must be provided. Link elements may reference a previously defined path (or url) element to define a relative path to the link resource. The content of the link element is used for link display.
- Example: <description>Test suite extending previous suite to provide post processing of event logs leading to a <link src="eventtimeplot.png" path=".">graphical plot of event timing results</link>.</description>

Attributes

- **src**[string] - Resource reference.
- **path**[string] - Reference path identifier.

* indicates required attribute.

D.4.19 image

Description

- Image elements may be used within descriptions to provide for inclusion of an embedded image. The image resource reference must be provided. Image elements may reference a previously defined path (or url) element to define a relative path to the image resource. The content of the image element is used as a caption.
- Example: <description> Object timing results are collected and processed into a plot showing object creation/deletion cummulative times: <image path="." src="objecttimeplot.png"> Object Creation/Deletion Timing Results </image> </description>

Attributes

- **src**[string] - Resource reference.
- **path**[string] - Reference path identifier.

* indicates required attribute.

D.4.20 requirement

Description

- Requirements may be referenced through use of this element. References indicate requirements that are addressed by the case/suite. Requirement references are formed using a path-like structure indicative of the requirement hierarchy.
- Example: <requirement ref="corecore/system/service/startup"/>

Attributes

- **src**[string] - Resource reference.
- **path**[string] - Reference path identifier.

* indicates required attribute.

D.4.21 select

Description

- Select block construct defining a property used in selection test criteria. Contains sequence of <when> elements each defining a block selected conditionally based on test satisfaction. May also contain an <otherwise> element defining a block selected only if all <when> clause conditions are unsatisified.
- Example: <select property="persist.type">

Attributes

- **property**[string] - Reference to defined property used in selection.

* indicates required attribute.

Elements

- when D.4.22 [1...*]
- otherwise D.4.23 [0...1]

D.4.22 when

Description

- Selection block defining a pattern match (regular expression) test criteria. Block selected if pattern match successful.
- Example: <when match="j.*">

Attributes

- **match**[string] - Regular expression defining match pattern.

* indicates required attribute.

Elements

- (case D.4.2 suite D.4.1 path D.4.3 url D.4.4 classpath D.4.11 property D.4.15 include D.4.10 exec D.4.8 arg D.4.12 option D.4.13) [0...*]

D.4.23 otherwise

Description

- Default selection block.

Elements

- (case D.4.2 suite D.4.1 path D.4.3 url D.4.4 classpath D.4.11 property D.4.15 include D.4.10 exec D.4.8 arg D.4.12 option D.4.13) [0...*]

D.5 Example Suite

This example suite document illustrates basic element semantics as expected/implemented in the suite execution framework. As an aid in the description of the element usage, consider this example document to be located in the following path: [$HOME/core/test/examples/exampleSuite.xml] with the suite execution framework (and all dependencies) installed at [$HOME/core].

An XML suite document must contain a root <suite> element. <suite> elements must include an 'id' attribute if the suite is concrete. A <suite> element may optionally (highly suggested) include a more descriptive 'name' attribute. A <suite> defined without an 'id' attribute will be treated as a virtual suite, inheriting all characteristics from its' parent but providing no additional structure. The primary use for a virtual suite would be in the definition of reuseable suites that may be included (see <include> element description below) in a core suite document. Root suites should also specify a 'basepath' defining the result root location relative to the suite document. This

’basepath’ value is used for subsequent document presentation with references to result documents (hence the need to specify a location relative to the document). The following <suite> element defines a ’basepath’ that will result in the root location of [$HOME/core] (i.e., two directory levels up from the document location).

<suite id="root" name="Root Suite" basepath="../..">

<path> elements may be defined within the scope of a <suite>. In this context a <path> element must define an ’id’ attribute whose value must be unique within scope. <path> elements may also use other <path> elements as a reference (through use of the ’ref’ attribute) with the content of the new <path> element extending the referenced <path> element. The built-in paths ’/’, ’.’, ’..’, and ’*’ may be referenced and represent the result root directory, the current suite result context, the parent suite result context (or the result root directory for the outer suite), and a system resource respectively.

The following <path> elements define paths to a ’lib’ directory referenced to the parent context (in this case the result root directory since it is defined within the scope of the outer suite) and a ’scr/uml’ directory locatable as a system resource (the directory exists and is located relative to any path contained in the list of resource search paths). In this example, "lib.path" would point to [$HOME/core/lib] and "scr.path" might point to [$HOME/core/-scr/uml] (if $HOME/core were included in the list of resource search paths).

<path id="lib.path" ref=".."> lib**</path>**

<path id="scr.path" ref="*"> scr/uml**</path>**

<suite> elements may be nested within other <suite> elements. This allows for the grouping of suites comprising cases that constitute a logical unit. Additionally, it provides a segregation of case results organized by suite hierarchy (e.g., all output captured from case processes will be placed into directories arranged according to the suite structure). In addition to the attributes described above for the <suite> element the following example includes the ’exit’ attribute. Use of this attribute implies that upon successful completion of this suite’s cases that any running processes should be terminated. The numeric value specifies a delay (in seconds) prior to a forced exit.

<suite id="examplesuite" name="Example Suite" exit="10">

<property> elements may be used to define property/value pairs for subsequent reference.

<property id="ns.path">

<path ref="."> NS_Ref**</path>**

</property>

The following <property> element includes use of a <url> element to provide a URL based path. The following will result in the property value set to [file://$HOME/core/root/examplesuite/NS_Ref].

<property id="ns.url">

<url protocol="file" ref="."> NS_Ref**</url>**

</property>

The following <path> element shows use of the built-in ’.’ path as a reference. This results in a path to a ’lib’ directory referenced to the current context (i.e., [$HOME/core/root/examplesuite/lib] reflecting the current suite scope).

<path id="lib.suite.path" ref="."> lib**</path>**

<description> elements may be included within any primary elements (<case>, <suite>, <path>, <classpath>) to provide inline documentation. This documentation may also be used in a visual presentation of the suite definitions (through the use of an XSL transform to HTML, for example). The use of <description> elements is highly encouraged.

<description> Example suite illustrating usage of basic suite definition elements defined by the Suite XML Schema.**</description>**

<command> elements are used to define command execution. <command> elements defined using an ’id’ attribute may be referenced by subsequent <command> elements. In the context of an execution suite, commands may only be defined for later reference. Commands defined within suites that do not define an id will be ignored.

<command id="perl">

<exec> elements are used to specify an external executable. The ’path’ attribute may be used to reference a previously defined path. If no path is specified then the executable will be assumed to be locatable based on the underlying system execution search path.

<exec> perl**</exec>**

</command>

<case> elements are used to define case process execution. <case> elements must define an ’id’ attribute and optionally (highly suggested) a more descriptive ’name’ attribute.

<case id="jacorbproperties" name="Generate JacORB Properties">

Required case artifacts may be specified through the <artifact> element. The case will fail if any specified artifact is not created as a result of case execution. Additionally, a case will not be processed if all of its artifacts exist prior to execution. A previously defined path may be referenced through the ’path’ attribute. The following example defines an expected artifact of [$HOME/core/root/examplesuite/lib/jacorb.properties].

<artifact path="lib.suite.path"> jacorb.properties**</artifact>**

General commands may be specified through the <command> element. A previously defined command may be referenced through the ’ref’ attribute. The following <command> element defines a Perl script execution which is executed by the previously defined ’perl’ command.

<command ref="perl">

The following <exec> element specifies the script to be executed. The script will be located using the path [$HOME/core/scr/makefile.pl] (assuming that the path $HOME/core is in the list of resource search paths).

<exec path="*"> scr/makefile.pl**</exec>**

Arguments may be supplied through the use of <arg> elements.

<arg> jacorb.properties.tpl**</arg>**

<path> elements may be used within an <arg> element to provide an argument value requiring a path. Note that this feature may be used to constrain use of ancillary files (whether they are used for input or output) to paths that are referenced to the containing suite context thereby enabling isolation of case execution (hopefully providing less chance of a collision with input/output used in other cases). The following will result in an argument constructed with the path [$HOME/core/root/examplesuite/lib/jacorb.properties].

<arg>

<path ref="lib.suite.path"> jacorb.properties**</path>**

</arg>

The following <arg> element shows reference to a <property> element resulting in an argument string of [nsref=file:/-/$HOME/core/root/examplesuite/NS_Ref].

<arg> nsref= **<property ref="ns.url"> </property>**

</arg>

</command>

</case>

<classpath> elements are used to define a grouping of <path> elements that will be used to define a class search path sequence. <classpath> elements may include an ’id’ attribute when used within the scope of a suite. A

<classpath> element may reference another <classpath> element using the 'ref' attribute with additional <path> elements extending the referenced <classpath> element.

<classpath id="boot.classpath">

<path ref="*"> lib/jacorb_v2_1.jar**</path>**

<path ref="*"> lib/logkit-1.2.jar**</path>**

<path ref="*"> lib/avalon-framework-4.1.5.jar**</path>**

<path ref="*"> lib/concurrent-1.3.2.jar**</path>**

<path ref="*"> lib/antlr-2.7.2.jar**</path>**

</classpath>

The following <case> element illustrates use of elements defined specifically to support Java based processes.

<case id="nameserver" name="Start Name Service">

<command>

The <class> element is used to define the Java class containing the process main entry.

<class> org.jacorb.naming.NameServer**</class>**

<classpath>

<path ref="lib.suite.path"> </path>

</classpath>

<option> elements are similar to <arg> elements but may be used to specifically define command options. If an 'id' attribute is defined for the <option> element then the value will be output as an argument using standard option specification (i.e., the option id prepended with a '-' is passed as a single argument). The option id may be left undefined for those cases where this standard is not applicable (such as in the following example). Note that options defined for Java based processes will be passed to the Java virtual machine, not to the Java main class.

<option> -Xbootclasspath/p: **<classpath ref="boot.classpath"> </classpath>**

</option>

The following argument references a previously defined property resulting in the path [$HOME/core/root/examplesuite/-NS_Ref].

<arg>

<property ref="ns.path"> </property>

</arg>

</command>

A <condition> element may be defined for a case. Use of a <condition> element implies that the case process is to be executed as a background process and that completion of the case is considered successful once the condition pattern is matched in the process standard output. Once the condition is satisfied subsequent cases will be allowed to execute with the current case process forked to the background (i.e., the process continues to execute to completion, or otherwise, independant of the parent process).

<condition> NameServer-POA - ready**</condition>**

</case>

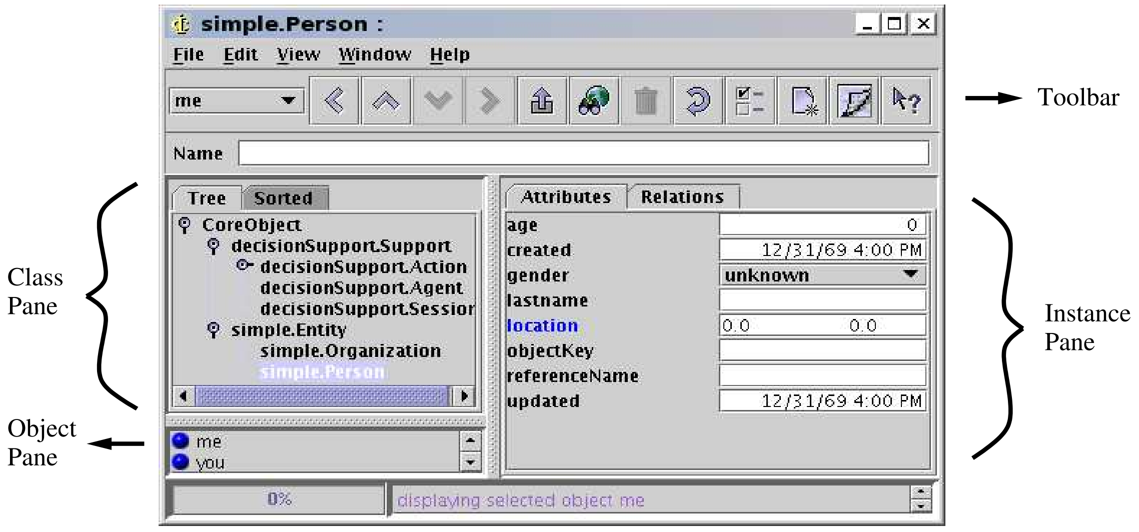
The <include> element is used to specify an external Case XML document for inclusion. The contents of the file are treated as if the defined elements were specified directly within the context of the <include> element. The included document must specify a root <suite> element which may be virtual (i.e., no 'id' attribute defined) in which case the contents of the <suite> will be treated as contents of the current suite. Note that since no 'path' is specified, the included document is expected to be located relative to the current suite document (i.e., in this case in the same directory).

<include> omlTestSuite.xml**</include>**

</suite>

</suite>

Appendix E - Instance Viewer



CoreObject
- decisionSupport.Support
  - decisionSupport.Action
  - decisionSupport.Agent
  - decisionSupport.Session
- simple.Entity
  - simple.Organization
  - simple.Person

age	0
created	12/31/69 4:00 PM
gender	unknown
lastname	
location	0.0 0.0
objectKey	
referenceName	
updated	12/31/69 4:00 PM

● me
● you

0% displaying selected object me

Figure E.1: Instance Viewer Main Window.

E.1 Main Window

The Instance Viewer can be used to create, delete, and modify instances for a class defined in the object model. It can also be used to formulate and perform queries for instances existing in the object server.

E.1.1 Class Pane

Displays the class hierarchy starting at the class for which the Instance Viewer is being displayed. To traverse up the tree use the toolbar button ***expand*** [⌃]. To traverse down a tree use the ***narrow*** toolbar button [⌄]. To display the instance attributes for a given class in the Instance Pane (see below), select the class name in the tree.

E.1.2 Object Pane

Displays a list of local objects for the class selected in the Class Pane. To display the instance information for an object double-click the object in the list. Objects in the Object Pane can be permanently deleted by selecting one or more objects and pressing the ***[delete]*** key.

E.1.3 Instance Pane

Displays the instance information for an object selected in the Object Pane or the attributes for a class name selected in the Class Pane. The attributes displayed are separated into two broad categories: attributes and relations. The relations tabbed pane contains all associations and aggregations. All other attributes fall in the "attributes" category. The following layout is used for the attributes listed :

- All visible attributes are displayed in a two column table.
- The first column represents the attribute name and the second column represents the attribute value.
- Initially, the default value for that attribute is displayed.
- All enumeration attributes are rendered as combo boxes, with a list of possible values. A new value can be set by choosing a value from the combo box.
- All boolean attributes are rendered as a check box that can be selected or de-selected as necessary.
- All other fields are represented by a text field, where new values can be typed in to set the value and must be succeeded by the carriage return.
- All struct attributes are displayed with a highlighted (blue) color. Struct fields can be displayed by selecting the struct attribute and double-clicking once. To roll-up the fields, double-click again.
- Array values are displayed in a combo box.
- Once an object has been instantiated, all read-only attributes are disabled and cannot be edited.
- By default hidden attributes are not displayed. The following property must be set to true to display all attributes in a class:

  com.cdmtech.core.client.gui.template.showHidden=true
- All associations and aggregations are displayed separately as nodes in a tree under the ’relations’ tabbed pane.
- The display of attributes can be customized by selecting the attributes in the Customize Window.

E.1.4 Instance Viewer Toolbar Functions

This toolbar contains a scroll-down list of objects that have been viewed in the current Instance Viewer. It also includes a text field to input a display name when creating a new object and a set of buttons whose actions are defined below:

‹	back	Move back to previous object.
⌃	expand	Traverses up the class tree, expanding to include the parent class of the root class if it exists.
⌄	narrow	Traverses down the class tree.
›	forward	Move forward to next object.
⇧	post	Posts any changes for the current instance.
🔍	query	Querys for objects.
🗑	delete	Deletes the current instance permanently from the object server.
↻	reset	Resets any attributes that have been modified since last posting.

☑	customize	Displays a customization dialog in which attributes can be selected or deselected.
	new	Opens a new Instance Viewer
	clear	Clears the current object and display an empty template for the selected class.
	help	Displays context sensitive help.

To reopen an instance previously viewed, select it from the scroll-down list. When creating a new object, the display name must first be entered in the text field. This creates a template for the object being created and sets the attribute defined in the following class property to the value entered in the ***Name*** field:

<className>.class.disAttrName=<attrName>

where ***<className>*** is a valid class name in the object model and ***<attrName>*** is a valid attribute name in the class. For example, a class called BaseObject that contains an attribute called "objectName," the property can be set as follows:

BaseObject.class.disAttrName=objectName

E.1.5 Steps to use the Instance Viewer

Follow the steps below to create, delete, and modify objects:

Create a new object

- Select the class name in the Class Pane to create an instance of that class
- Type in a display name in the ***Name*** field and hit the ***[enter]*** key.
- Set attribute values as necessary. All read-only attributes must be set prior to posting the object for the first time. See section below on modifying existing objects on how to set values for different attribute types.
- Click on the ***post*** button [ ] to submit the object.

Delete an existing object

- Select the class name in the Class Pane.
- The existing objects for the selected class are displayed in the Object Pane. Selecting the root class name displays all locally existing objects.
- Select the object to be deleted in the Object Pane and press the ***[delete]*** key.
- If the object is currently being viewed in the Instance Viewer, the ***delete*** button [ ] in the toolbar can be clicked on to delete the object.

Modify an existing object

- Select the class name in the Class Pane. The existing objects for the selected class are displayed in the Object Pane.
- In the Object Pane double-click the object you wish to edit. This will activate the object, displaying its attribute values in the Instance Pane and its name in the window title bar. A specific instance must be active in the Instance Pane in order to edit and post any changes.
- Modify attribute values as outlined below and click on the ***post*** button [ ] in the toolbar to submit the changes.
  - To set boolean values, toggle the checkbox.
  - To set enumeration type attribute values, the value can be selected in the drop-down list of the ComboBox.
  - To set numeric or string values, the attribute name can be selected and the value typed in the text field. For numeric values the unit can also be specified in addition to the magnitude.
  - To add an element to an array attribute, type in a new value in the editable field for the given array attribute. To remove an existing element, select the element in the drop-down combo box and press the ***[delete]*** key.
  - To modify struct fields, select the attribute name and double-click to display the fields. Values can be modified normally as described above. Double-clicking the attribute name again, rolls up the struct fields.
  - To add a struct to a struct array, select the struct attribute name and press the ***[insert]*** key. This adds a struct to the array list. Struct fields can be modified, as described above, by first selecting the struct to be modified in the drop-down list and then double-clicking to display the fields.
  - To add an association or aggregation:
    - Select the ***Relations*** tab to display the association role names.
    - Select the role name to associate an object.
    - Press the ***[insert]*** key to display the Instance Viewer for the associated class.
    - To create a new instance to associate, follow the steps enumerated above to create a new instance. Once the instance is posted, the association is automatically handled by the Instance Viewer.
    - To select an existing object to associate, select the association role name and then select the object in another Instance Viewer (if open) or press the ***[insert]*** key to open the Instance Viewer for the associated class.
  - To remove a struct value from a struct array list:
    - Select the struct value to remove from the drop-down list.
    - Press the ***[delete]*** key to remove the selected value.
  - To remove an association or aggregation:
    - Select the ***Relations*** tab to display the association role names.
    - Select the role name to display the associated objects.
    - Select the object to disassociate and press the ***[delete]*** key.
- Remember to click the ***post*** button [ ] in the toolbar to submit the changes.

Viewing Instances

- Select the class name in the Class Pane.

- If the object you would like to view exists locally, it will be displayed in the Object Pane. If the object does not exist locally, press the ***query*** button [ ]. To load the object locally, select the object key from the result list displayed in the Query Viewer. The selected object will appear in the Object Pane of the Instance Viewer.
- Select the object in the Object Pane and double-click to view it in the InstancePane.

Class Queries

- Select the class name in the Class Pane.
- Click the ***query*** button [ ]
- The results of the query are displayed in a separate window.
- One or more objects can be selected to load them locally.

Complex Queries

- Select the class name in the Class Pane.
- To constrain the query by attribute value, select the attribute (must not be an array or struct typed attribute) and change its value. The entered value will be used to constrain the query to only those objects whose attribute value matches. Any number of attributes values may be set to further constrain the query.
- Click the ’query’ button [ ] to display the results of the query.

Saving and Restoring objects

- Select the class name in the Class Pane.
- Select the object to save in the Object Pane and double-click to view it.
- From the ***Instance*** menu select ***save*** to save the object to a file.
- Select ***restore*** from the menu to restore an object.

E.2 Query Viewer

The results of a query are displayed in the Query Viewer (see figure E.2). A list of object keys are displayed. To view the selected instances in an Instances Viewer, click the ***view*** button [ ].

E.2.1 Query Pane

The Query Pane displays a list of object keys for objects that satisfy the query performed. A display attribute name can be displayed instead by specifying the following class property:

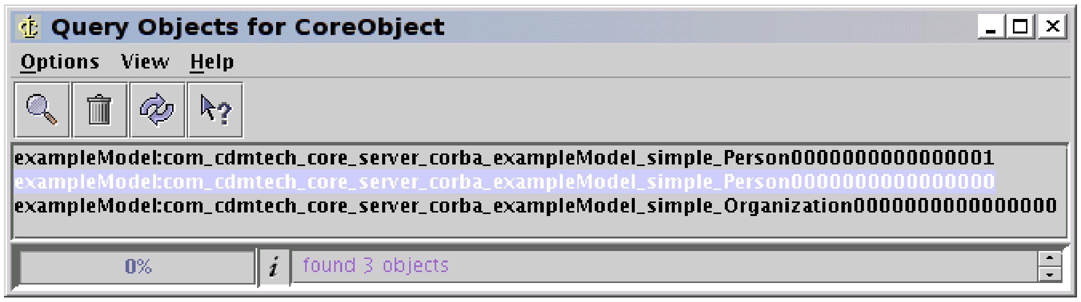
<className>.class.queryAttrName=<attrName>

where,

<className> is a valid class name in the object model

<attrName> is a valid attribute name in the above class

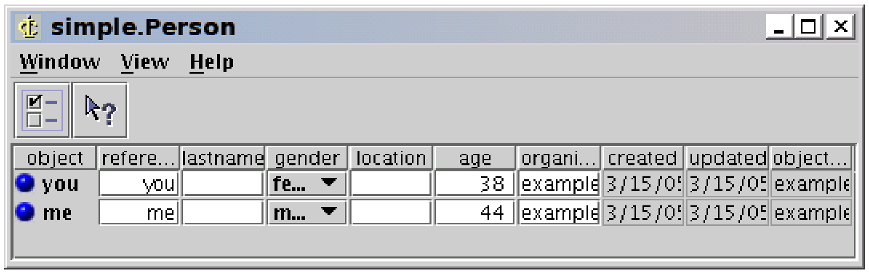
One or more objects can be selected to load them locally or viewed in the Instances Viewer.



exampleModel:com_cdmtech_core_server_corba_exampleModel_simple_Person0000000000000001
exampleModel:com_cdmtech_core_server_corba_exampleModel_simple_Person0000000000000000
exampleModel:com_cdmtech_core_server_corba_exampleModel_simple_Organization0000000000000000

found 3 objects

Figure E.2: Sample Query Viewer Window.



object	refere...	lastname	gender	location	age	organi...	created	updated	object...
● you	you		fe...		38	example	3/15/0	3/15/0	example
● me	me		m...		44	example	3/15/0	3/15/0	example

Figure E.3: Sample InstancesViewer Window.

E.2.2 Query Viewer Toolbar

The functions of the various buttons contained in the Query Viewer are as follows:

🔍	view	View selected objects in Instances Viewer.
🗑	delete	Delete selected object.
⟳	refresh	Performs the query again and refreshes the list displayed.
↖?	help	Context sensitive online help.

E.3 Instances Viewer

Multiple instances can be displayed in the Instances Viewer. This is mainly used to display a list of query objects. Attributes for the root class, for which the query was performed, are displayed. Further customization can be performed to narrow down the attributes being displayed. When making an association, an object can be selected in the Instances Viewer by double-clicking the object in the table. An Instance Viewer can also be displayed by double-clicking the object in the table. Figure E.3 displays a sample Instances Viewer.

E.3.1 Instances Pane

The Instances Pane displays a table for a class of objects. Attributes are displayed starting at column two. By default all the attributes for the root class are displayed. For each object, the corresponding values of each attribute are displayed. The attributes displayed can be customized. Click the mouse on the column header, to select that specific column to be sorted in ascending order.

E.3.2 Instances Viewer Toolbar

The Instances Viewer Toolbar contains the following buttons whose actions are defined below:

☑	customize	Displays a customization dialog in which attributes can be selected or deselected.
?	help	Displays context sensitive help

Appendix F - Object Shell

F.1 Overview

The Object Shell is a generic command-line based user interface built using the Metamata parser generator[1]. The syntax is defined by grammer specified in an annotated Java source file. Additionally, Java code is imbedded to directly invoke methods through the Object Management Layer (OML) in response to the parsing of command-line production segments (groups of tokens matching a prescribed pattern). Additionally, the Object Shell supports command completion.

F.2 Usage

The command line syntax provided in the Object Shell is fairly minimal in content, focusing instead on object interaction. There are no built-in commands provided, instead it relies on object characteristics/behavior to provide additional functionality. The syntax consists of a symbolic language providing basic operations to create objects, set object values/associations, get objects and values/associations, post objects and values/associations, and delete objects. The Object Shell command line also supports command completion for class names, attribute names, variable names, and enumerated attribute values. Partial names may be typed followed by the ***[Tab]*** key to invoke command completion. If multiple matches are found then the choices will be displayed and the command fragment may be updated with a partial completion that contains any common matched substring.

The following summerizes some of the possible operations that may be performed with the next section giving specific examples.

- create object (local create - post object to instantiate)

  ```
  objectName=[ClassName]
  ```

- create object with initial values (local create - post object to instantiate)

  ```
  newObject=[ClassName].(attrName=value ...)
  ```

- set object value (local modify - post object to set)

  ```
  newObject.attrName=value
  ```

- add element to object array value (local modify - post object to set)

  ```
  newObject.attrName+=value
  ```

- remove element from object array value (local modify - post object to set)

  ```
  newObject.attrName-=value
  ```

[1] Metamata Parse, Metamata, Inc, http://www.metamata.com

- set object association (local modify - post object to set)

  ```
  newObject.assocRole=object
  ```

- get object

  ```
  object=newObject
  ```

- get object value

  ```
  value=object.attrName
  ```

- get object association

  ```
  value=object.assocRole
  ```

- find objects

  ```
  objects=<ClassName>
  ```

- find objects with constraint

  ```
  objects=<ClassName>.(attrName=value ...)
  ```

- post object (may be combined with create/set operations)

  ```
  ->newObject
  ```

- delete object

  ```
  <-object
  ```

- show value

  ```
  value
  ```

- show attribute/association value

  ```
  object.attrName
  ```

- show class attributes

  ```
  [ClassName]
  ```

- show number of values

  ```
  #value
  ```

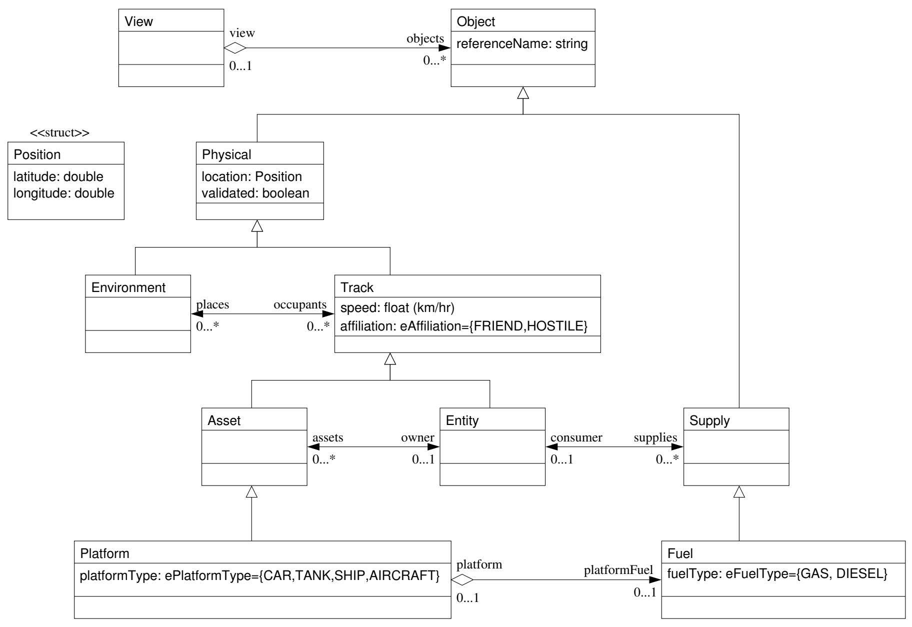


Figure F.1: Example Object Model.

F.2.1 Examples

For the following examples use the model shown in figure F.1.

```
// create a new Platform instance of type TANK and assign it to myTank
myTank=[Platform].platformType=:TANK

// set location of myTank to a new Position with latitude
// and longitude set to (35, -122)
myTank.location=[Position].(latitude=35 longitude=-122)

// set referenceName of myTank to "My Tank"
myTank.referenceName=:My Tank

// instantiate myTank (i.e., invoke its constructor)
->myTank

// create a new Fuel instance of type DIESEL and assign it to myFuel
myFuel=[Fuel].fuelType=:DIESEL

// set platformFuel of myTank to myFuel and update instance
// (i.e., invoke its set method)
->myTank.platformFuel=myFuel

// get Platform_role of myFuel and display (note that this
// was set when the platformFuel association role was set
// for myTank previously)
myFuel.Platform_role
```

```
    // delete myTank instance (deletes myFuel as well since it
    // is an aggregate part of myTank)
    <-myTank

    // recreate a new Tank instance as above, but using a single
    // command line
    ->myTank=[Platform].(platformType=:TANK \
     location=[Position].(latitude=35 longitude=-122) \
     referenceName=:My Tank \
     platformFuel=[Fuel].(fuelType="DIESEL" referenceName="My Fuel"))

    // query for all Platform instances and assign result
    allPlatforms=<Platform>

    // print number of Platform instances found
    #allPlatforms

    // print location attribute value for first Platform
    allPlatforms(0).location

    // query for Platform instances of type TANK
    allTanks=<Platform>.platformType=:TANK

    // query for specific TANK Platform instance with
    // referenceName set to myTank
    aTank=<Platform>.(platformType="TANK" referenceName="My Tank")

    // change location for last found tank
    ->aTank.location=[Position].(latitude=0 longitude=0)
```

F.3 Syntax

F.3.1 Tokens

```
    < ID: ['a'-'z','A'-'Z'] ( ['a'-'z','A'-'Z','0'-'9','_'] )* >
          | < NID: ['['] (<ID> ['.'])* <ID> [']']>
          | < SID: ['<'] (<ID> ['.'])* <ID> ['>']>
          | < POST: ['-'] ['>'] >
          | < DELT: ['<'] ['-'] >
          | < HELP: ['?'] >
          | < ADDI: ['@'] >
          | < REMI: ['~'] >
          | < SET: ['='] >
          | < ADD: ['+'] ['='] >
          | < REM: ['-'] ['='] >
          | < DECIMAL_LITERAL: ['0'-'9'] (['0'-'9'])* >
          | < #HEX_LITERAL: "0" ['x','X'] (['0'-'9','a'-'f','A'-'F'])+ >
          | < #OCTAL_LITERAL: "0" (['0'-'7'])+ >
          | < NUMBER_LITERAL: ( <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> ) >
          | < INTEGER_LITERAL: ( ['-'] )? (
               <DECIMAL_LITERAL> (['l','L'])?
             | <HEX_LITERAL> (['l','L'])?
```

```
        | <OCTAL_LITERAL> (['l','L'])? )
      >
    | < FLOATING_POINT_LITERAL: ( ['-'] )? (
          (['0'-'9'])+ "." (['0'-'9'])* (<EXPONENT>)? (['f','F','d','D'])?
        | "." (['0'-'9'])+ (<EXPONENT>)? (['f','F','d','D'])? )
      >
    | < #EXPONENT: ['e','E'] (['+','-'])? (['0'-'9'])+ >
    | < TOKEN_LITERAL:
        ":"
        ( (~['\"','\n','\r'])
        )*
      >
    | < STRING_LITERAL:
        "\""
        ( (~['\"','\n','\r'])
        )*
        "\""
      >
```

F.3.2 Productions

input

```
 setExpr() ["\n" | "\r"]
|
 getExpr() ["\n" | "\r"]
|
 postExpr() ["\n" | "\r"]
|
 deleteExpr() ["\n" | "\r"]
|
 helpExpr() ["\n" | "\r"]
|
 addiExpr() ["\n" | "\r"]
|
 remiExpr() ["\n" | "\r"]
```

setExpr

```
 object() "." <ID> op() "(" valueGroup() ")"
|
 object() "." <ID> op() value()
|
 <ID> <SET> value()
```

getExpr

```
 "#" value()
|
 value()
```

postExpr

```
 <POST> setExpr()
|
 <POST> object() "." <ID>
|
 <POST> object()
```

deleteExpr

```
 <DELT> object()
```

helpExpr

```
 <HELP> <NID> "." <ID>
|
 <HELP> <NID>
|
 <HELP> object() "." <ID>
|
 <HELP> object()
|
 <HELP>
```

addiExpr

```
 <ADDI> <SID> "." <ID>
|
 <ADDI> <SID>
|
 <ADDI> object() "." <ID>
|
 <ADDI> object()
```

remiExpr

```
 <REMI> <SID> "." <ID>
|
 <REMI> <SID>
|
 <REMI> object() "." <ID>
|
 <REMI> object()
```

op

```
 <SET>
|
 <ADD>
|
 <REM>
```

object

```
 <ID> "(" <INTEGER_LITERAL> ")"
|
 <ID>
```

setGroup

```
 ( set() )+
```

set

```
 <ID> op() "(" valueGroup() ")"
|
 <ID> op() value()
```

valueGroup

```
 ( value() )+
```

value

```
 <NID> ".(" setGroup() ")"
|
 <NID> "." set()
|
 <NID>
|
 <SID> ".(" setGroup() ")"
|
 <SID> "." set()
|
 <SID>
|
 object() "." <ID>
|
 object()
|
 <NUMBER_LITERAL>
|
 <TOKEN_LITERAL>
|
 <STRING_LITERAL>
```

Appendix G - Glossary of Terms and Acronyms

agent A software module capable of limited reasoning about information defined within the context of a knowledge domain.

agent session A collection of agents logically grouped by domain, typically acting on information presented in a particular view. See: view.

aggregate A class that represents the whole in an aggregation (whole-part) relationship. See: aggregation.

aggregation A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

API Application Programmer's Interface. A set of interfaces comprising a library, system, or application.

architecture The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components, and subsystems.

artifact A physical piece of information that is used or produced by a software development process. Examples of Artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component. Synonym: product. Contrast: component.

association The semantic relationship between classifiers that specifies connections among their instances.

association end The endpoint of an association, which connects the association to a classifier.

attribute A feature within a classifier that describes a range of values that instances of the classifier may hold.

cardinality The number of elements in a set. Contrast: multiplicity.

child In a generalization relationship, the specialization of another element, the parent. See: subclass. Contrast: parent.

class A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. See: interface.

classifier A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

CLIPS C Language Integrated Production System. A rule-based expert system shell originally developed by NASA.

component A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers (e.g., implementation classes) that reside on it, and may be implemented by one or more artifacts (e.g., binary, executable, or script files). Contrast: artifact.

CORBA Common Object Request Broker Architecture[12]. A standard specification for distributed objects and services.

datatype A descriptor of a set of values that lack identity and whose operations do not have side effects. Datatypes include primitive pre-defined types and user-definable types. Predefined types include numbers, string and time. User-definable types include enumerations.

design The part of the software development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

development process A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

domain An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

element An atomic constituent of a model.

enumeration A list of named values used as the range of a particular attribute type. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with values from the set {false, true}.

feature A property, like operation or attribute, which is encapsulated within a classifier, such as an interface, a class, or a datatype.

framework A reusable architecture for all or part of a system. Frameworks typically include classes, patterns or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks. See: pattern.

generalization A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: inheritance.

IDL Interface Definition Language[12].

IOR Initial Object Reference used by the ORB to resolve to an initial service (e.g., the name service).

inheritance The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See generalization.

instance An entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations. See: object.

interface A named set of operations that characterize the behavior of an element.

JESS Java Expert System Shell. A rule-based expert system shell developed by the Sandia National Laboratories.

meta-data Information describing model structure.

meta-model A model that defines the language for expressing a model.

model An abstraction of a physical system with a certain purpose. See: physical system.

model element An element that is an abstraction drawn from the system being modeled.

multiplicity A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: cardinality.

namespace A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: name.

object An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. See: class, instance.

ORB Object Request Broker - see CORBA[12].

package A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

parent In a generalization relationship, the generalization of another element, the child. See: subclass. Contrast: child.

primitive type A pre-defined basic datatype without any substructure, such as an integer or a string.

process

1. A heavyweight unit of concurrency and execution in an operating system. Contrast: thread, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
2. A software development process defining the steps and guidelines by which to develop a system.
3. To execute an algorithm or otherwise handle something dynamically.

property A named value denoting a characteristic.

physical system

1. The subject of a model.
2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast: system.

reference

1. A denotation of a model element.
2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: pointer.

relationship A semantic connection among model elements. Examples of relationships include associations and generalizations.

role The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

service A specialized reusable component that serves a common purpose. See component.

stereotype A type of modeling element that extends the semantics of the meta-model. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes.

subclass In a generalization relationship, the specialization of another class; the superclass. See: generalization. Contrast: superclass.

subpackage A package that is contained in another package.

superclass In a generalization relationship, the generalization of another class; the subclass. See: generalization. Contrast: subclass.

system A top-level subsystem in a model. Contrast: physical system.

tagged value The explicit definition of a property as a name-value pair. In a tagged value, the name is referred to as the tag.

TIRAC Toolkit for Information Representation and Agent Collaboration. A framework and toolkit used for building collaborative decision support systems.

top level A stereotype of package denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces see outwards.

UML Unified Modeling Language. A language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

view A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

visibility An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

XMI XML Metadata Interchange. A standard format for storing model information in terms of meta-model specifications.

XML Extensible Markup Language. A standard markup language for storing human and machine readable information in a structured form.

XSL XML Stylesheet Language. A standard language typically used to transform an XML document, stored using a specific schema, into another format. Allows separation of document content from presentation format.

Bibliography

[1] Ernest Friedman-Hill. *Jess In Action*. Manning Publications, 2003.

[2] Sandia National Laboratories. Jess the Rule Engine for the Java Platform. herzberg.ca.sandia.gov.

[3] Apache Jakarta Project. Tomcat. jakarta.apache.org/tomcat.

[4] Apache Web Services Project. Axis. ws.apache.org/axis.

[5] Altova. MapForce. www.altova.com/products_mapforce.html.

[6] Embarcadero Technologies. Describe. www.embarcadero.com/products/describe.

[7] Gentleware AG. Poseidon for UML. www.gentleware.com/products.

[8] IBM. Rational Software. www.ibm.com/software/rational.

[9] LaTeX Project. LaTeX, A Document Preparation System. www.latex-project.org.

[10] Novosoft. Novosoft UML Library. www.novosoft-us.com.

[11] Object Management Group, Inc. XML Metadata Interchange (XMI). www.omg.org, November 2000.

[12] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification. www.omg.org, February 2001.

[13] Object Management Group, Inc. IDL to Java Language Mapping Specification. www.omg.org, May 2001.

[14] Object Management Group, Inc. OMG Unified Modeling Language Specification. www.omg.org, September 2001.

[15] Leslie Lamport. *LaTeX, A Document Preparation System*. Addison-Wesley, second edition, 1994.

[16] Jens Pohl, Kym Jason Pohl, Russell Leighton, Michael Zang, Steven Gollery, and Mark Porczak. The TIRAC Development Toolkit: Purpose and Overview. Technical Report CDM-17-04, CDM Technologies, Inc., August 2004.

[17] Kym Jason Pohl and Lakshmi Vempati. A Translational Web Services Bridge Solution for Meaningful Interoperability Between Potentially Disparate Systems. Paper to be presented at the 2005 IEEE Aerospace Conference, March 5-12, 2005, Big Sky, MT., March 2005.

Keyword Index