COMPONENT WORKLOAD EMULATOR UTILIZATION TEST SUITE

User and Developer's Guide

Version 1.0

Contents

Preface

Enterprise distributed systems (such as air traffic management systems, cloud computing centers, and shipboard computing environments) are steadily increasing in size (e.g., lines of source code and number of hosts in the target environment) and complexity (e.g., application scenarios). To address challenges associated with developing next-generation enterprise distributed systems, the level-of-abstraction for software development is steadily increasing. Now-a-days, distributed system developers focus more on the system's "business-logic" instead of wrestling with low-level implementation details, such as development and configuration, resource management, and fault tolerance. Moreover, increasing the level-of-abstraction for software development promotes reuse of the system's "business-logic" across different application domains, which inherently reduces (re)invention of core intellectual property.

Although increasing the level-of-abstaction for software development is improving functional properties of next-generation enterprise distributed systems, system quality-of-service (QoS) properties (*e.g.*, latency, throughput, and scalability) are not validated until late in the software lifecycle, *i.e.*, at system integration time. This is due in part to the *serialized-phasing development problem* where the infrastructure- and application-level system entities, are developed during different phases of the software lifecycle. Consequently, distributed system developers do not realize the system under development does not meet its QoS requirements until its too late, *i.e.*, at complete system integration time, at the expense of overrun project cost and deadlines.

System execution modeling (SEM) is a model-driven engineering technique that helps overcome the effects of serialized-phasing development. SEM tools provide distributed system developers with the necessary artifacts for modeling system behavior and workload, such as computational attributes, resource requirements, and network communication. The constructed models are then used to validate QoS properties of the system under development during early phases of the software lifecycle. This enables distributed system developers to pinpoint potential QoS bottlenecks before they become to costly to locate and resolve in a cost-effective manner late in the software lifecycle.

The Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) is a SEM tool designed for next-generation enterprise distributed systems. Distributed system developers and testers use CUTS to model the expected behavior and workload of system components under devel-

opment using high-level domain-specific modeling languages. Model interpreters then transform the constructed behavior and workload models into source code for the target architecture, *i.e.*, the auto-generated components same interfaces and attributes as their real counterparts. Finally, system developers and testers emulate the auto-generated components in the target (or representative) environment and collect and analyze QoS metrics. This enables distributed system developers and testers to conduct system integration test at early stages of software lifecycle, instead of waiting until complete system integration time to perform such testing.

This book therefore serves as the user's guide to CUTS. It details its main functionalities, and includes concrete examples to provide better understanding of its concepts. This book is organized organized as follows:

- **Chapter 1** details how to build and install CUTS and its different toolsets. Depending on your needs, you many or more not install all the toolset. It is not a requirement to install all the toolsets.

- **Chapter 2** provides a provides a Quick Start Tutorial on using CUTS. The tutorial uses a real example that goes over the key task in CUTS, such as modeling behavior and workload, generating source code, running experiments, and analyzing collected QoS metrics.

Hope that you enjoy using CUTS as it helps you understand QoS properties during early phases of the software lifecycle.

Part I

Getting Started

Chapter 1

Building and Installing CUTS

CUTS has many small projects that comprise the entire system execution modeling (SEM) tool. Its many projects, however, can be divided into three main toolsets:

- Runtime architecture
- Modeling tools
- Analysis tools

This chapter discusses how to build and install each of the aforementioned toolsets for CUTS. Depending on your usage of CUTS, you may not need to build and install all projects in a category on the same machine. For example, you may install the runtime toolset of CUTS in your testing environment, and the modeling and analysis toolset outside of your testing environment to minimize interference with the testing process when viewing collected performance metrics in real-time. We are aware of these needs, and have setup the build process to decouple projects within a toolset from projects external to its corresponding toolset. You can, therefore, refer to each of the following sections on building and installation in isolation since toolsets do not explicitly depend on each other.

1.1 CUTS Runtime Toolset

The runtime toolset for CUTS allows developers and testers to emulate system experiments on their target architecture. CUTS also provides the mechanisms to monitor and collect performance metrics for the executing system. In order to build the CUTS runtime architecture, you will need the following technologies installed on the target machine(s):

- **DOC Group Middleware** - This set of middleware is used to abstract away complexities associated with implementing applications that operate on many different operating systems

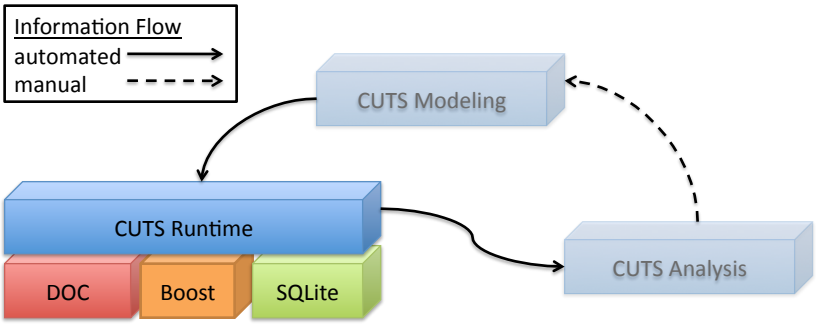


Figure 1.1. Building blocks for the CUTS runtime architecture

(ACE) and perform distributed communication (TAO). You can download DOC Group Middleware from the following location: http://www.dre.vanderbilt.edu.

- **Boost** - This set of libraries is used for implementing the parsers (Boost Spirit) used within different artifacts of CUTS. You can download Boost from the following location: http://www.boost.org.
- **SQLite** - This set of libraries is used to support flat file archives for test results. You can download SQLite at the following location: http://www.sqlite.org.
- **PCRE (not pictured)** - This set of libraries is used to enable PERL regular expression support in CUTS. You can download PCRE at the following location: http://www.pcre.org.
- **XSC (not pictured)** - This set of libraries and applications is used to convert XML documents to/from objects. You can download XSC from the following location: svn://svn.dre.vanderbilt.edu/XSC/trunk.

1.1.1 Obtaining Source Files

The quickest method to obtain the latest snapshot of the CUTS runtime architecture is to download and execute the following Python script:

https://svn.dre.vanderbilt.edu/viewvc/CUTS/trunk/CUTS/etc/scripts/download_sources.py

This script will download CUTS's source code from its the DOC Group Subversion repository at the following location:

svn://svn.dre.vanderbilt.edu/DOC/CUTS/trunk/CUTS

It will also download all third-party library source files (see Appendix A), and define a configuration script to set the correct environment variables. If you need to download a stable version of the source code, then you can access at one of the subdirectories in the repository at the following location:

`svn://svn.dre.vanderbilt.edu/DOC/CUTS/tags`

Unlike the latest snapshot, you are responsible for *manually* downloading all third-party source files, and defining a configuration file to set the correct environment variables.

1.1.2 System Configuration

If you use the Python script above, you can skip to the next section. Otherwise, you must for configure you environment before you can build CUTS. This is done by setting the following environment variables:

```
%> export CUTS_ROOT=location of CUTS
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUTS_ROOT/lib
%> export PATH=$PATH:$CUTS_ROOT/bin
```

Please see Appendix A for instructions for configuring, building, and installing third-party libraries needed by CUTS, such as DOC Group Middleware, Boost, and SQLite.

1.1.3 Installation

We use Makefile, Workspace, Project Creator (MPC) [1] to assist in building CUTS on different operating systems, and with different compilers. Before you can build the runtime architecture, you must first generate the target workspace. Use the following command to generate the workspace:

```
%> $ACE_ROOT/bin/mwc.pl -type TYPE [-features FEATURES] CUTS.mwc
```

where TYPE is your compiler type, and FEATURES is a comma-separated list of features for your build of the CUTS runtime architecture.[2] The complete set of features for CUTS is located in `$CUTS_ROOT/default.features.tmpl`. It is recommended that you copy the template feature file to `$CUTS_ROOT/default.features` and set the appropriate features for your workspace by modifying the new file. This way you do not have to use the `-features` command-line option unless you want to override your default feature selection. Once you have generated the workspace, you can build the solution using your specified compiler.

[1] For more information on MPC, please see the following location: `http://www.ociweb.com/products/mpc`

[2] You can type `$ACE_ROOT/bin/mwc.pl --help` to view the command-line options for MPC.

1.2 CUTS Modeling Toolset (Windows-only)

CUTS modeling tools provide system developers and testers with an enviroment for rapidly constructing experiments for distributed component-based systems, and generating testing system for their target architecture. The modeling tools are built on the following technologies:

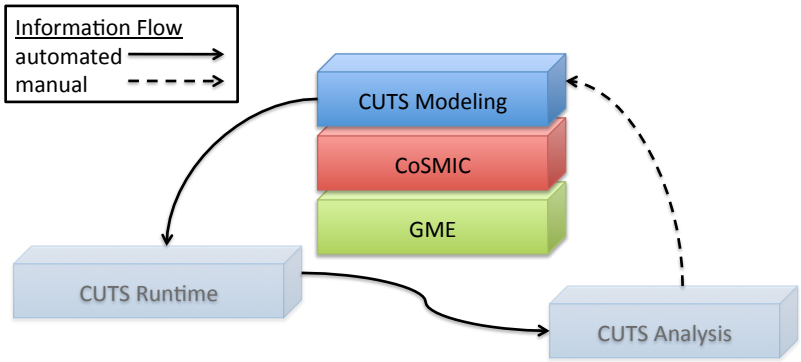


Figure 1.2. Building blocks for the CUTS modeling tools

- **GME** - The Generic Modeling Environment (GME) is a graphical modeling tool for creating domain-specific modeling languages (DSMLs). You can download GME from the following location: `http://www.isis.vanderbilt.edu/projects/GME`.
- **CoSMIC** - The Component Synthesis via Model Integrated Computing (CoSMIC) is a tool suite designed to address the complexities of building large-scale component-based systesm, such as sytem assembly, deployment, packaging, and planning. You can download CoSMIC from the following location: `http://www.dre.vanderbilt.edu/cosmic`.

1.2.1 Installing Prebuilt Version

The easiest and quickest way to install the CUTS modeling tools is to install them using the .msi installer. You can download the latest version of the CUTS modeling tools from the following location: `http://www.dre.vanderbilt.edu/CUTS/downloads`. Before you can install the CUTS modeling tools, please make sure you have installed GME and CoSMIC. Once both GME and CoSMIC are installed (in that order), then you can install the CUTS modeling tools.

1.2.2 Building from Sources (advanced)

If you choose, you can build the modeling tools from source. This is given you have downloaded all the source from the CUTS source code repository (see Section 1.1.1). To build the CUTS

modeling tools, first install the latest version of GME. Then, you **MUST** build CoSMIC from source as well. This is required because the CUTS modeling tools are built on top of CoSMIC, and its therefore dependent on CoSMIC. You can find instructions for building CoSMIC from source at the following location:

http://www.dre.vanderbilt.edu/cosmic/downloads

After you have built and installed CoSMIC from sources, you are ready to build the CUTS modeling tools from source. Please use the following commands to build the modeling tools using your flavor of Visual C++:

```
%> cd %CUTS_ROOT%
%> mwc.pl -type [vc type] -features modeling=1,cosmic=1 CUTS_CoSMIC.mwc
%> open solution and build
```

The build process will ensure that all the interpreters are installed and configured for your environment.

1.3 CUTS Analysis Toolset

The CUTS analysis tools allow developers to view and analyze system performance metrics. The analysis tools are developed using the following key technologies:

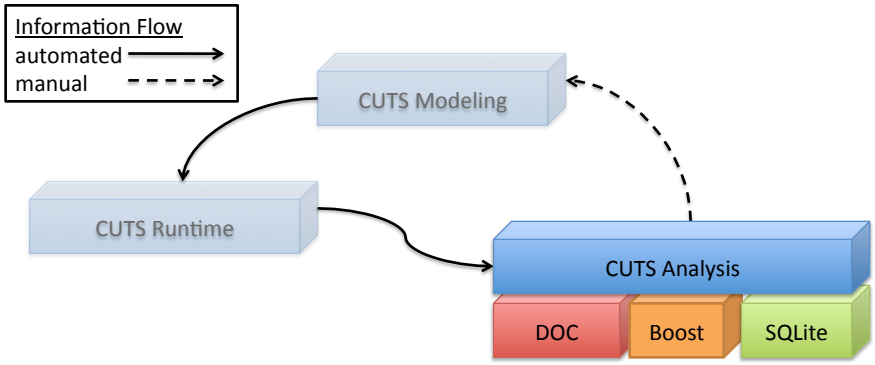


Figure 1.3. Building blocks for the CUTS analysis tools

As shown in Figure 1.3, the same technologies used to develop the runtime toolset (see Figure 1.1) are used to develop the analysis toolset. Because the same technologies are used in the analysis toolset, the core projects in analysis toolset is also built while building the runtime toolset.

Chapter 2

Quick Start Tutorial

This chapter provides a quick start tutorial for using CUTS. Before beginning this tutorial, you should already be familiar with the following technologies:

- Component Integrated ACE ORB (CIAO)
- Component Synthesis with Model Integrated Computing (CoSMIC)
- Deployment And Configuration Engine (DAnCE)

The purpose of this tutorial is to give you a quick introduction to using CUTS. For this tutorial, you will be creating a simple client/server application and measuring the oneway event latency (*i.e.*, measuring how long it takes an event to travel between two components). At the end of this tutorial, you will have a basic understanding of the following:

- Add simple behavior to a component. This is discussed in Section 2.1.
- Generate source code for the components for emulation. This is discussed in Section 2.2.
- Run a CUTS emulation on a single host. This is discussed is Section 2.3.
- Analyze the system execution traces generated by the emulation using UNITE. This is discussed in Section 2.4.

2.1 Modeling Behavior and Workload

Adding behavior and workload to a PICML model is done using two domain specific modeling langauges (DSMLs) named the *Component Behavior Modeling Language (CBML)* and the *Workload Modeling Language (WML)*, respectively. Both DSMLs have been integrated into PICML so

users can reference the structural model of their systems when modeling the system's behavior and workload.

Before we begin adding behavior client/server example, we first need the following file into GME:

```
$(CUTS_ROOT)/tutorials/QuickStart/models/QuickStart.xme
```

Right now, behavior and workload modeling is supported at the component level. In this example, we will be adding behavior the `PingComponent`, which is located at `InterfaceDefinitions/PingPong/PingPong`. In this tutorial, we have already added behavior and workload to the `PongComponent` since it is more complex.

The PingComponent Behavior

This PingComponent already contain model elements for its output port. What remains is associating the correct behavior and workload with this output port for emulation purposes. With this in mind, we are going to assume the PingComponent sends periodic events to the any component connected to its output port. Therefore complete the following steps (elements will be automatically be generated as well):

1. Add a `PeriodicTask` model element to the active model and set its name to eventProducer and its `Hertz` attribute to 10 (*i.e.*, 10 events/sec).

2. Insert an `InputAction` element, change its name to `eventProducer`, and connect the `PeriodicTask` to the `InputAction`.

3. Insert an `OutputAction`. This will automatically generate a new `State` element and connect it with the inserted `OutputAction`.

4. Connect the final State with the originating `InputAction`, *i.e.*, `periodicPing`, to signify end of the behavior.

5. Insert a `Variable` model element and set it to reference the `UnsignedLongInteger` element in the `PredefinedTypes` folder. Name the variable `eventCount` and set the initial value of the variable to 0.

6. Select the connection between the `InputAction` and the `State`. Type the following in the `Effects` attribute field: `this->eventCount_ ++;`

7. Open the `OutputAction` and insert a `SimpleProperty`. Set it to reference the `UnsignedLongInteger` element in the `PredefinedTypes` folder, and set its `Value` attribute to the following: `this->eventCount_`.

Figure 2.1 illustrates the complete behavior model of the `PingComponent` in PICML.

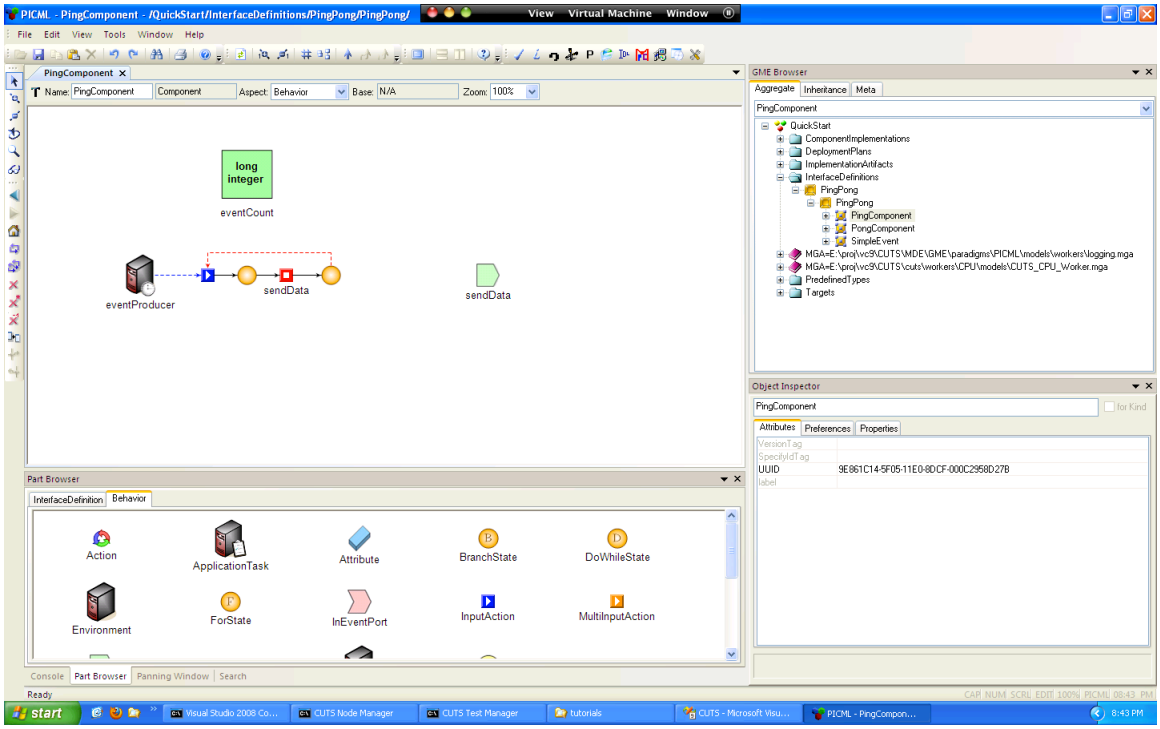


Figure 2.1. Complete behavior model for the PingComponent.

2.2 Generating Source Code for Emulation

After modeling the behavior and workload, the next step is to generate source code from the model. This will enable emulation of the test system on its target architecture. To generate source code from the model, launch the CUTS interpreter and execute the following steps:

1. Enter the target output directory
2. Select **Component Integrated ACE ORB (CIAO)** in the listbox
3. Click the **OK** button

Once the CUTS interpreter finishes generating source code, the IDL files need to be generated in order to compile the source code. These files are created by the IDL Generator provided with CoSMIC. Finally, there will be a `QuickStart.mwc` file located in the output directory selected for source code generation. This is a *Makefile, Project and Workspace Creator (MPC)* workspace file that contains all the necessary information to sucessfully compile the generated source code. Use `QuickStart.mwc` to generate the appropriate workspace and then compile it.

NOTE: The auto-generated source code for this model is also available at the following location:

```
$(CUTS_ROOT)/tutorials/QuickStart
```

2.3 Running a CUTS Emulation

One design goal of CUTS is to (re)use the same infrastructure used in the target environment. The QuickStart example uses the *Deployment And Configuration Engine (DAnCE)*, which is included with CIAO's standard distribution, to deploy the test system. To deploy the example, use the following steps:

1. Copy the contents of the `./descriptors` directory in the tutorial to the directory where you generated the source code.
2. In one window, launch the deployment infrastructure using the following command:

   ```
   %> cutsnode_d -c deployment.config
   ```

3. In another window, launch the test using the following command:

   ```
   %> cutstest -c test.config --time=30
   ```

This will run the test for 30 seconds. While the system is executing, it will generate an execution trace, which will be stored in the test database. This execution trace is what we will analyze in the next, and final, step.

2.4 Analyzing System Execution Traces

CUTS analyzes quality-of-service (QoS) properties via system execution traces with a tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)*. In this quick start tutorial, we generated a system execution trace that can be used to measure the service time of an event. We are now going to use UNITE to analyze the generated system execution trace. To do this, execute the following steps:

1. Copy the contents of the `./analysis` directory in the tutorial to the directory where you generated the source code.

2. Execute the following command:

```
%> cuts-unite -f ../descriptors/00000000-0000-0000-0000-000000000000.cdb
  -c QuickStart.unite
```

After executing UNITE, you should get an average service time for an event based on what is captured in the generated system execution trace.

Part II

CUTS Modeling Toolset

Chapter 3

PICML Modeling Quick Reference

This section provides a quick reference to the modeling environment. In particular, it focuses on key elements in PICML modeling language (`http://www.dre.vanderbilt.edu/cosmic`) so you do not spend much time and effort trying to determine what elements you need to build a CUTS model.

The layout of this section is as follows: each subsection corresponds to a folder type under the root folder in a PICML model. Within each section, we then highlight the important elements, *i.e.*, those that are used most often. This does not necessarily mean you will not use elements covered in the section. Finally, for each element, we explain its purpose so you have a better understanding of its role in the model.

3.1 Predefined Types

Predefined types, *e.g.*, integer, boolean, and string, are defined in the `PredefinedTypes` folder. When you add a new predefined type folder to the model, PICML will automatically populate it will all the predefined types it currently supports.

3.2 Interface Definitions

The `InterfaceDefinitions` folder is used to define the interfaces, complex data types, and components, of the distributed system. This folder can contain only the `File` element type, which represents a *Interface Definition File (IDL)* file. Within the `File` element, the following elements are the most import elements when constructing a model that is useable by CUTS (listed in alphabetical order):

Aggregate

The `Aggregate` is similar to the structure construct (*i.e.*, `struct`) in C and C++, or a tuple in dynamic languages. It contains members elements, which reference their data type (*e.g.*, `String`, `LongInteger`, and `DateTime`). By default, all members of the aggregate are considered public.

The following is a list of the aggregate's child elements that are most important, which are discussed in this section: `Member` and `Key`.

Component

The `Component` element is an abstraction that provides/requires a set of services or sends/-receives events. This is determined by the ports it exposes to the other components. The component can also contain attributes, which are points-of-configuration that whose value is determined at deployment time. In CUTS, the component is where you model application behavior for early integration testing via emulation.

The following is a list of the component's child elements that are most important, which are discussed in this section: `ProvidedRequestPort`, `RequiredRequestPort`, `InEventPort`, `OutEventPort`, and `Attribute`, and `ReadonlyAttribute`.

Event

The `Event` element is similar to an aggregate in PICML in that it is a collection of data types. It can also contain more complex definitions, which are currently ignored in CUTS. The main difference is that an event is used to send data between two components via their `InEventPort` and `OutEventPort` ports.

The following is a list of an event's child elements that are most important, which are discussed in this section: `Member`.

InEventPort

The `InEventPort` element is a child of the `Component` element that represents a port for receiving events from other components. It is a reference element that refers to any `Event` element in the model.

Key

The `Key` element is a child element of an aggregate element. It is primarily used in DDS to define the key value(s) of a data type. If you are using the DDS code generators in CUTS, then you will definitely need to make use of this element. It is the only way to ensure you DDS data type definitions are complete.

Member

The `Member` element is a child element of an aggregate and event. It is a reference element that points to a members data type.

Object

The `Object` element is representative of an interface in a distributed system. It can also be thought of as a remote interface. This means that the object can have methods, which are invokable by other components. Currently, CUTS does not associate any behavior with objects.

OutEventPort

The `OutEventPort` element is a child of the `Component` element that represents a port for sending events to other components. It is a reference element that refers to any `Event` element in the model.

Package

The `Package` element is representative of a namespace is C++ and packages in Java. It is an object that provide scope hierarchy to the definition. The package contains the same elements as a file. This means the important elements discussed in this section also apply to a package element.

ProvidedRequestPort

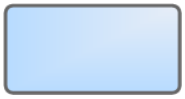
The `ProvidedRequestPort` element is a child of the `Component` element that represents a service that is provide by the parent component. It is a reference element that refers to any `Object` element in the model.

RequiredRequestPort

The `RequiredRequestPort` element is a child of the `Component` element that represents a service that is required by the parent component. It is a reference element that refers to any `Object` element in the model.

3.3 Component Implementations

The `ComponentImplementations` folder is used to define different component assemblies (*i.e.*, set of inter-connected components that communicate with each other). This folder can contain only the `ComponentImplementationContainer` element type, which is just a container with a view for visualizing a contain component assembly. Because of how component assemblies are defined in PICML, the `ComponentImplemenationContainer` has many different elements. Of those elements, the following are most important when constructing a model that is useable by CUTS (listed in alphabetical order):



ComponentAssembly

The `ComponentAssembly` model element is a container class that contains a set of related `ComponentInstance` elements and their connections. Within a component assembly, it is possible to instantiate a component assembly. This is done by, first, create a standard component assembly. In another component assembly, insert the former component assembly as a GME model instance.

ComponentInstance

The `ComponentInstance` model element is used to instantiate an instance of a `Component` implementation, which is defined in another part of the model. When you insert a `ComponentInstance` into the model, PICML's model intelligence helps you select the component instance's correct implementation.

Property

The `Property` element (in the `DeploymentProperties` aspect) is used to define configuration settings for elements in the model. In the component assembly, the property element can connect to an `Attribute`, `ComponentAssembly`, `ComponentInstance`, and `ReadonlyAttribute`.

In the component assembly, the following property names have predefined meanings:

Property Name	Data Type	Description
InstanceIOR	String	Set the output location of IOR
RegisterNaming	String	Register component with naming service
LocalityArguments	String	Set the parameters for the locality manager
CPUAffinity	String	Set the CPU affinity for the deployment
ProcessPriority	String	Set the processor priority for the deployment

Table 3.1. Listing of reserved names for Property elements within the component assembly aspect.

3.4 Targets

The `Targets` folder is used to define different network configurations and topologies of the target execution environment. This folder can contain only the `Domain` element type, which a single execution domain (*i.e.*, set of connected network entities). Within the `Target` element, the following elements are the most import elements when constructing a model that is useable by CUTS (listed in alphabetical order):

Node

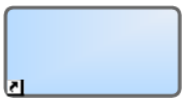
The `Node` element represents a host in the target network. If you want to use a node in a deployment plan, then it must be defined here first.

3.5 Deployment Plans

The `DeploymentPlans` folder is used to different deployment plans of the model system. This folder can contain only the `DeploymentPlan` element type, which is a single deployment of either the entire system or a subset of the system's components. Within the `DeploymentPlan` element, the following elements are the most import elements when constructing a model that is useable by CUTS (listed in alphabetical order):

CollocationGroup

The `CollocationGroup` element represents a container/process that hosts a set of components that share the same execution environment, such as threading model, run-time priorities, and programming language. This element is a GME set. This means you need to switch to set mode in order to add deployed model elements (*i.e.*, components and component assemblies) to a collocation group.



ComponentAssemblyRef

The `ComponentAssemblyRef` element is used to add a component assembly to a deployment plan. This means that all components in the assembly are deployed to the node that the component assembly is deployed. The element is a GME reference and must refer to a `ComponentAssembly` element. Finally, you specify what process the component assembly is deployed to by adding it to a collocation group.

ComponentInstanceRef

The `ComponentInstanceRef` element is used to add a component instance to a deployment plan. The element is a GME reference and must refer to a `ComponentInstance` element in a `ComponentAssembly` model. Finally, you specify what process the component instance is deployed to by adding it to a collocation group.

NodeRef

The `NodeRef` element represents a host that is part of the parent deployment. This element is a GME reference that refers to a `Node` in the `Domain` model. Finally, the node reference connects with a collocation group to show what processes/containers are executing on a node in the target environment.

Property

The `Property` element (in the `DeploymentProperties` aspect) is is used to define configuration settings for elements in the model. In the deployment plan, the property element can connect to a `NodeRef` and a `CollocationGroup`.

In the deployment plan, the following property names have predefined meanings:

Property Name	Data Type	Ex. Value
StringIOR	String	corbaloc:iiop:localhost:30000/PingNode.NodeManager

Table 3.2. Listing of reserved names for Property elements with the deployment plan aspect of the model.

Part III

CUTS Runtime Toolset

Chapter 4

CUTS Node Management Facilities

This chapter discusses the CUTS node manager facilities. These facilities are designed to simplify management of many processes that must execute on many different nodes in a large-scale distributed testing environment. Furthermore, these facilities simplify management of many different execution environments, which is hard to do using traditional shell scripting facilities, such as Bash, C-Shell, and Batch files on Windows.

4.1 Overview

Distributed testing and execution environments, such as network testbeds, consist of many nodes. These nodes are responsible for executing many different processes that must support the overall goals of the execution environment. For example, nodes in the execution environment may host processes that manage software components, processes that synchronize clocks, and processes that collect software/hardware instrumentation data.

When dealing with many processes that must execute on a given node, the traditional approach is to create a script that launches all processes for that given node. For example, in *Unix environments, Bash scripts can be used to launch a set of processes. The advantage of using such scripts is that it provides a *repeatable* technique for ensuring that all process for a given node are launched, and done so in the correct order. Moreover, the script is persistent and can be easily updated to either add or remove processes as needed.

Although such scripts are sufficient when working in a distributed testing environment, its approach adds more complexity to the distributed testing process. First, the scripts are only repeatable after *manually* invoking the script. This means that distributed testers much manually connect to each host in the environment and (re)start the script. Even if it is possible to remotely connect to the host via automation to (re)start the script, it is still *hard* to know what processes are associated with a given script.

Secondly, such scripts do no easily support injection/removal of processes on-the-fly. It is assumed in the script used to configure the host. At that point, all processes injected/removed from the host are assumed to be part of all the processes on that machine. Lastly, it is *hard* to switch between different execution environment where an execution environment is set of environment variables and a set of processes.

To address the limitations of traditional approaches to executing and managing a distributed testing environment, CUTS provides node management facilities. The facilities provide methods for managing many different execution environments, which can easily be swapped between. The facilities also provide tools for remotely controlling nodes in the execution environment.

The remainder of the chapter therefore presents the basics for using the CUTS node management facilities in a production-like environment.In particular, the following information is covered in this chapter:

- **CUTS Node Daemon** - The CUTS Node Daemon for managing many different execution environments for a node. Details of the CUTS Node Daemon are presented in Section 4.2.
- **CUTS Node Daemon Client** - The CUTS Node Daemon Client is responsible remotely controlling the CUTS Node Daemon. Details of the CUTS Node Daemon Client are provided in Section 4.3.

4.2 The Node Daemon

The CUTS Node Daemon is responsible for managing different execution environments for a node. There is typically one CUTS Node Daemon executing on a single node. There can be cases where more than one CUTS Node Daemon executes on the same machine, but this defeats the purpose of the daemon. Regardless of the situation, what is discussed in the remainder of this chapter can be used to execute one or more CUTS Node Daemon processes of a single host.

4.2.1 Configuring the Node Daemon

The CUTS Node Daemon is designed to manage different execution environments. Before it can doe this, however, it must be properly configured. The configuration for the CUTS Node Daemon is similar to a shell script, except that it is specified in XML format. Listing 4.1 illustrates an example configuration for the CUTS Node Daemon.

```
1 <?xml version="1.0" encoding="utf-8" standalone="no" ?>
2 <cuts:node xmlns="http://cuts.cs.iupui.edu"
3            xmlns:cuts="http://cuts.cs.iupui.edu"
4            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5            xsi:schemaLocation="http://cuts.cs.iupui.edu cutsnode.xsd">
```

```
6
7   <environment id="default" inherit="true" active="true">
8     <!-- environment variables for this environment -->
9     <variables>
10      <variable name="NameServiceIOR"
11                value="corbaloc:iiop:localhost:60003/NameService" />
12    </variables>
13
14    <startup>
15      <!-- NamingService -->
16      <process id="dance.naming.service" waitforcompletion="false">
17        <executable>${TAO_ROOT}/orbsvcs/Naming_Service/Naming_Service</executable>
18        <arguments>-m 0 -ORBEndpoint iiop://localhost:60003 -o ns.ior</arguments>
19      </process>
20
21      <!-- List of DAnCE NodeManager executions -->
22      <process id="dance.nodemanager.pingnode" waitforcompletion="false">
23        <executable>${DANCE_ROOT}/bin/dance_node_manager</executable>
24        <arguments>-ORBEndpoint iiop://localhost:30000
25          -s ${DANCE_ROOT}/bin/dance_locality_manager
26          -n PingNode=PingNode.ior -t 30
27          --instance-nc corbaloc:rir:/NameService</arguments>
28        <workingdirectory>../lib</workingdirectory>
29      </process>
30
31      <!-- The one and only execution manager -->
32      <process id="dance.executionmanager" waitforcompletion="false">
33        <executable>${DANCE_ROOT}/bin/dance_execution_manager</executable>
34        <arguments>-eEM.ior --node-map PingPong.nodemap</arguments>
35      </process>
36    </startup>
37  </environment>
38 </cuts:node>
```

Listing 4.1. An example configuration file for the CUTS Node Daemon.

Without going into the full details of the configuration in Listing 4.1, the CUTS Node Configuration contains three critical sections:

- **Environment** – The environment section, denoted by the `<environment>` tag, defines an execution environment. There can be more than one execution environment in a configuration As shown in Listing 4.1, there is only one execution environment named `default`.

The remaining details of the execution environment is discussed in Section 4.2.4.

- **Variables** – The variables section, denoted by the `<variable>` tag, defines environment variables for the parent execution environment. All processes in the execution environment can use these environment variables via the ${} tag (see line 17). As shown in Listing 4.1, the `default` execution environment contains one environment variable named `NameServiceIOR`. This environment variable is not explicitly used in the configuration. Instead, the processes in this configuration expect for the variable to be defined as part of the default environment variables. The remaining defaults of the variables section is discussed in Section 4.2.4.

- **Startup/Shutdown Processes** – The startup/shutdown section, denoted by `<startup>` and `<shutdown>`, respectively, defines the processes to execute when an execution environment starts up or shuts down. As shown in Listing 4.1, the configuration defines only a set of startup processes. The remaining details of the processes section is discussed in Section 4.2.4.

4.2.2 Running the Node Daemon

Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS Node Daemon is installed at the following location:

```
%> $CUTS_ROOT/bin/cutsnode_d
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cutsnode_d --help
```

Assuming the configuration discussed above is defined in a file named *cutsnode.config*, you run the CUTS Node Daemon with the configuration via the following command:

```
%> $CUTS_ROOT/bin/cutsnode_d -c cutsnode.config
```

4.2.3 Configuring the Node Daemon Server

When you run the CUTS Node Daemon, the server is configured with its default values. In many cases, you may want to change the server's configuration. For example, you may want to bind the server to localhost instead of its hostname, or you may want to change the port on which it is listening.

You can configure the server using the `--server-args` command-line option. This should be a quoted command-line option because there can many different arguments passed to the server. The following table lists the different command-line options:

Name	Required	Description
--register-with-iortable	No	Name used to expose server via the IORTable
--register-with-ns	No	Name used to register server with naming service

In addition to these options, you can passed any option supported by the ORB, such as `-ORBEndpoint`. The following command-line illustrates how to change to listening address of the CUTS Node Daemon to `node.hydra.iupui.edu:20000` and expose it via the IORTable as `CUTS/NodeDaemon`.

```
%> $CUTS_ROOT/bin/cutsnode
  --server-args="-ORBEndpoint iiop://node.hydra.iupui.edu:20000
  --register-with-iortable=CUTS/NodeDaemon"
```

4.2.4 Node Daemon Schema Definition

The section discusses the details of the CUTS Node Daemon configuration based on its underlying XML Schema Definition.

`<cuts:node>`

This is the main tag for the configuration document. Its XML namespace must have the `http://cuts.cs.iupui.edu` definition. If it does not have this definition, then the configuration will not load.

`<environment>`

This tag defines the different execution environments. As previously stated, there can be more than one execution environment in a configuration file. The following is a list of attributes supported by this tag.

Name	Type	Default Value	Required	Description
id	String		Yes	Unique id for the execution environment
inherit	Boolean	false	No	Inherit the system's environment variables
active	Boolean	false	No	This is the active execution environment

The following is a list of child tags: `<variables>`, `<startup>`, `<shutdown>`.

`<variables>`

This is the container tag for the environment variables in execution environment. This tag does not contain any attributes. The following is a list of child tags: `<variable>`.

<variable>

This tag is used to defined environment variables for an execution environment. The following is a list of attributes supported by this tag.

Name	Type	Default Value	Required	Description
name	String		Yes	Name of the environment variable
value	String		Yes	Value of the environment variable

This tag does not contain any child tags.

<startup>

This tag is a container for a list of processes that are to executed with an environment starts up. The order in which the processes are listed is their startup order. The following is a list of child tags: <process>.

<shutdown>

This tag is a container for a list of processes that are to executed with an environment is shutdown. The order in which the processes are listed is their shutdown order. The following is a list of child tags: <process>.

<process>

This tag defines an executable process for the execution environment. The following is a list of attributes supported by this tag.

Name	Type	Default Value	Required	Description
id	String		Yes	Unique id for the process
delay	Float	0.0	No	Seconds to wait before executing process
waitforcompletion	Boolean	false	No	Wait for process to complete before progressing

The following is a list of child tags: <executable>, <arguments>, <workingdirectory>.

<executable>

This tag defines the complete/relative path of the executable process. Environment variables can be used within this tag. This tag does not contain any child tags.

<arguments>

This tag defines the command-line for the executable process. Environment variables can be used within this tag. This tag does not contain any child tags.

<workingdirectory>

This tag defines the working directory for the executable process (*i.e.*, the directory where executable process will be spawned). Environment variables can be used within this tag. This tag does not contain any child tags.

4.3 The Node Daemon Client

The purpose of the CUTS Node Daemon Client is to remotely control different CUTS Node Daemon application. This prevents users from having to manually login to different nodes when there is need to alter the execution environment.

4.3.1 Running the Node Daemon Client

Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS Node Daemon Client is installed at the following location:

```
%> \$CUTS\_ROOT/bin/cutsnode
```

To see a complete list of command-line options, use the following command:

```
%> \$CUTS\_ROOT/bin/cutsnode --help
```

Assuming the CUTS Node Daemon is running at the address `iiop://node.hydra.iupui.edu:20000` and registered with the IORTable as `CUTS/NodeDaemon` as discussed above, then you can connect and control it via the CUTS Node Daemon Client using the following command:

```
%> $CUTS_ROOT/bin/cutsnode --reset -ORBInitRef
  NodeDaemon=corbaloc:iiop:node.hydra.iupui.edu:20000/CUTS/NodeDaemon
```

In this example, the `-ORBInitRef` is a `corbaloc:`. This, however, need not always be the case. It can be any StringIOR support by *The ACE ORB (TAO)*, such as a IOR file or naming service location.

Chapter 5

CUTS Testing Facilities

Chapter 6

CUTS Logging Facilities

This chapter discusses the CUTS logging facilities. These facilities are designed to simplify collection of system execution traces in a distributed environment. Moreover, the facilities simplify partitioning system execution traces with different tests. This therefore makes it possible to execute multiple tests in the same environment while preserving data integrity. Finally, the logging facilities discussed in this chatper are extensible via intercepters, and are highly configurable so they adapt to different application domains.

6.1 Overview

Distributed systems consist of many software components executing on many hosts that are connected via a network. When validating distributed system function and quality-of-service (QoS) properties, *e.g.*, end-to-end response time, scalability, and throughput, it is necessary to collect data about the systems behavior in the target environment. Such behaviors could be the events executed of the execution lifetime of the system, the state of the system at different points in time, or data points needed to calculate the end-to-end response of an event.

When dealing with data collection in a distributed environment, such as those that host distributed systems, data is collected and analyzed either *offline* or *online*. In offline collection and analysis, data from each host is written to local persistent storage, such as a file, while the system is executing in its target environment. After the system is shutdown, the collected data in local storage on each host is combined and analyzed. The advantage of this approach is network traffic is kept to a minimum since collected data is not transmitted over the network until after the system is shutdown. The disadvantage of this approach is collected data is not processed until the system is shutdown. This can pose a problem for systems with a long execution lifetime, *e.g.*, ultra-large-scale systems that must run 24x7, or when trying to monitor/analyze a system in real-time using collected data.

In online collection and analysis, data is collected and transmitted via network to a host outside the execution environment. The advantage of this approach is that it allows analysis of metrics data an environment that does not use the systems resources. This also helps not to skew the results while the system is running. The disadvantage of online collection and analysis is the difficulty of devising a strategy for efficiently collecting data in a distributed environment and submitting it to a central location without negatively impacting the excuting systems QoS—especially if the system generates heavy network traffic.

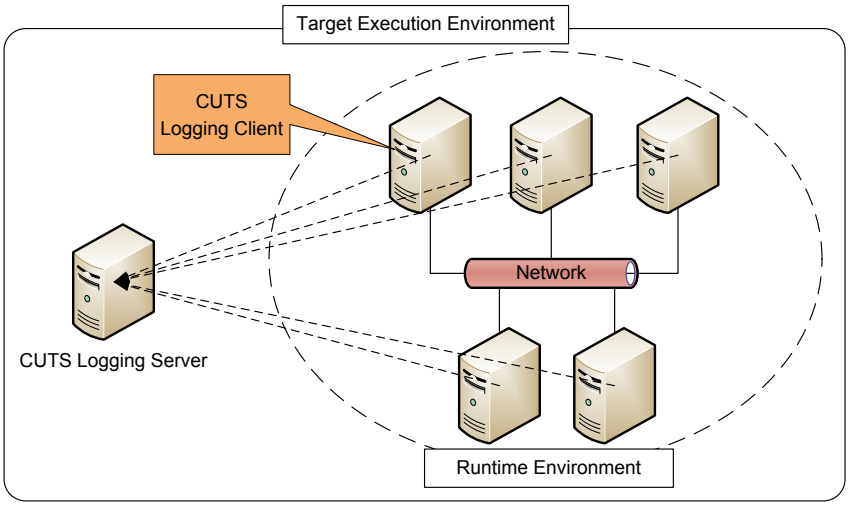


Figure 6.1. Overview of the CUTS logging facilities.

Because online collection and analysis is a more promising approach to data collection and analysis, CUTS logging facilities implement online data collection. As illustrated in Figure 6.1, there is a target execution environment and a runtime environment. The runtime environment is where the system is executing and generating data for collection. Each host in the runtime environment has a logging client (see Section 6.3). The logging client is responsible for collecting data from individual software/hardware components on its host. After collecting data on the host, the logging client periodically submits collected data to a logging server (see Section 6.2), which is executing outside of the runtime environment.

The remainder of the chapter therefore presents the basics for using the CUTS logging facilities in a production-like environment. In particular, the following information is covered in this chapter:

- **CUTS Logging Server** - The CUTS Logging Server is responsible for collecting data from individual CUTS Logging Clients executing in the runtime environment. Details of the CUTS Logging Server are presented in Section 6.2.

- **CUTS Logging Client** - The CUTS Logging Client is responsible for collecting data from individual client loggers and submitting it to the CUTS Logging Server. Details of the CUTS Logging Client are provided in Section 6.3.
- **Client Logger** - The client logger is responsible for collecting actual data from the runtime environment and submitting it to the CUTS Logging Client executing on its host. Details of the client logger are provided in Section 6.4.

6.2 The Logging Server

The logging server is responsible for collecting data from individual logging clients in the runtime environment (see Section 6.3). As shown in Figure 6.1, there is typically one instance of a logging server running in the target execution environment. There can also be cases when more than one instance of a logging server is desired, such as using multiple logging servers to balance the workload due to the amount of data collected by individual logging clients in the runtime environment. This user's manual, however, assumes that only a single instance of a logging server is used in the runtime environment.

6.2.1 Running the Logging Server

Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS Logging Server is installed at the following location:

```
%> $CUTS_ROOT/bin/cuts-logging-server
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-logging-server --help
```

6.2.2 Configuring the Logging Server

There are only a handful of command-line options for configuring the logging server. The most important of the available command-line options is `--register-with-iortable=NAME`, where `NAME` is an user-defined name, such as `LoggingServer`. When this option is combined with the `-ORBEndpoint ENDPOINT` CORBA command-line option, an external reference to the logging server is exposed. This is what allows the CUTS Logging Client (see Section 6.3) to connect to the logging server.

The following is an example of exposing the logging server named `LoggingServer` on port 20000 for logging clients to connect:

```
%> $CUTS_ROOT/bin/cuts-logging-server -ORBEndpoint \
   iiop://`hostname`:20000 --register-with-iortable=LoggingServer
```

As shown in the example above, the logging server is listening using port 20000 on its host [1], and has the name `LoggingServer`. The logging client can therefore connect to the logging server using the following CORBA reference:

```
corbaloc:iiop:`hostname`:20000/LoggingServer
```

The next section therefore discusses how to configure the logging client to connect to the correct logging server in the runtime environment.

6.3 The Logging Client

The logging client is responsible for collecting data from individual client loggers in the runtime environment (see Section 6.4). As shown in Figure 6.1, each host in the runtime environment typically runs one instance of a logging client. Similar to the logging server, there can also be cases when more than one instance of a logging client is desired, such as using multiple logging clients to balance the workload due to the amount of data collected by individual client loggers in the runtime environment. This user's manual, however, assumes that only a single instance of a logging client running on each host in the runtime environment.

6.3.1 Running the Logging Client

Similar to the CUTS Logging Server, the CUTS Logging Client is installed at the following location:

```
%> $CUTS_ROOT/bin/cuts-logging-client
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-logging-client --help
```

6.3.2 Configuring the Logging Client

The CUTS Logging Client is configured similar to how the CUTS Logging Server is configured (see Section 6.2.2). For example, the `-ORBEndpoint ENDPOINT` defines the CORBA endpoint for the logging client. The logging client's endpoint is used by the client loggers (see Section 6.4) that need to connect to the CUTS Logging Client executing on its local host.

[1] `hostname` is used because it is assumed that Unix command-line substitution is available on the host in the example. If such a feature is not available, it is possible to hardcode the hostname in place of `hostname`, *e.g.*, host.foo.bar.com

The addition to configuration parameters similar to the CUTS Logging Server, to the logging client's configuration requires the location of the CUTS Logging Server. In Section 6.2.2, the logging server was given an endpoint that is used by the logging client when connecting to it. To specify the location of the CUTS logging server, use the `-ORBInitRef LoggingServer=LOCATION` CORBA command-line option, where `LOCATION` is the location of the logging server [2].

The following is an example of configuring the logging client to connect to the logging server discussed in Section 6.2.2 and listens to data from client loggers on port 20000:

```
%> $CUTS_ROOT/bin/cuts-logging-client -ORBEndpoint \
   iiop://`hostname`:20000 --register-with-iortable=LoggingClient \
   -ORBInitRef LoggingServer=corbaloc:iiop:server.foo.bar.com:20000/LoggingServer
```

The next section discusses how to use client loggers to submit data for collection and analysis to the logging clients executing on each host in the runtime environment.

6.4 The Client Loggers

The client loggers are responsible for collecting log messages from an application and submitting them to the logging client on its host machine. Since client loggers interact directly with the application, they are language dependent. Currently, CUTS supports client loggers for C++ and Java applications. In addition, CUTS support the following logging frameworks: ACE Logging Facilities and log4j. The remainder of the section dicusses how directly integrate client loggers into an application and indirectly intergrate them using an pre-existing logging framework.

6.4.1 Direct Integration

Direct integration is the simplest method for integrating CUTS logging facilities into an existing application. It is also the most instrusive approach for integrating the logging facilities into an existing application. This is because you have to augment existing code to include the client logger(s). Only then can the application use to client logger to collect messages and submit them to the logging client executing on the localhost.

The following code illustrates integrating the client logger using the C++ version:

```
1 #include "cuts/utils/logging/client/logger/Client_Logger.h"
2
3 int main (int argc, char * argv []) {
4   try {
5     // connect client logger to logging client.
6     CUTS_Client_Logger logger;
```

[2] The location of the `LoggingServer` can be a `corbaloc`, `corbaname`, or `IOR`.

```
7       if (-1 == logger.connect (argv[1]))
8         return 1;
9
10      // Send a single, yet simple message.
11      logger.log (LM_DEBUG, "this is a simple message");
12
13      // Disconnect from the logging client.
14      logger.disconnect ();
15      return 0;
16    }
17    catch (const ::CORBA::Exception & ex) {  }
18    catch (...) {  }
19
20    return 1;
21  }
```

Listing 6.1. Example illustrating integration of the C++ client logger.

Likewise, this code illustrates the same concept using the Java version of the client logger:

```
1  import CUTS.client.logging.Logger;
2
3  public class TestApplication {
4    public static void main (String [] args) {
5      String loggingClientLoc = /* location of logging client */
6      Logger clientLogger = new Logger ();
7
8      // connect to the logging client
9      clientLogger.connect (loggingClientLoc);
10
11     // send a log message
12     clientLogger.log ("this is a simple message");
13
14     // disconnect from the logging client
15     clientLogger.disconnect ();
16   }
17 }
```

Listing 6.2. Example illustrating integration of the Java client logger.

In both the C++ and Java example, the `connect()` method takes as its input parameter the location of the logging client. This location can be a CORBA IOR, corbaname, or corbaloc (*i.e.*, anything that is a valid parameter for the `string_to_object()` method implemented on the

ORB). After a connection is made to the logging client, the client logger submits messages to the logging client using the `log()` method. When the client logger is done (or the application is ready to exit), it disconnects from the logging client using the `disconnect()` method. This is a critical step because it ensures the logging client does not get into an inconsistent state when trying to keep track of the number of active client loggers.

6.4.2 Indirect Integration

Indirect integration is a non-intrusive method for integrating the CUTS logging facilities with an existing application. This approach only work if the application is already using a logging framework, such as ACE Logging Facilities or log4j. Indirect integration works by intercepting log messages sent to the original logging framework and submitting them to the CUTS logging facilities (*i.e.*, the logging client).

Unlike direct integration, system developers do not need to update their existing code base to leverage indirect integration. Instead, they must update the existing logging frameworks configuration files to "install" the necessary inteceptors. The remainder of this section therefore presents details on using indirect integration with supported logging frameworks in CUTS.

log4j

Language. Java
How to integrate. Update `log4j.properties` (or similar file)
Illustrative example.

```
# define the loggers
log4j.rootCategory=ALL, Console, LoggingClient

# console appender
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

# CUTS appender
log4j.appender.LoggingClient=cuts.log4j.LoggingClientAppender
log4j.appender.LoggingClient.LoggerClient=corbaloc:iiop:localhost:20000/LoggingClient
```

ACE Logging Facilities

Language. C++
How to integrate. Update `svc.conf` (or similar file)
Illustrative example.

```
dynamic CUTS_ACE_Log_Interceptor Service_Object * \
 CUTS_ACE_Log_Interceptor:_make_CUTS_ACE_Log_Interceptor() active \
 ''--client=corbaloc:iiop:localhost:20000/LoggingClient''
```

Part IV

CUTS Analysis Toolset

Chapter 7

Understanding Non-functional Intentions via Testing and Experimentation (UNITE)

This chapter discusses the a tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)*. The purpose of UNITE is to support analysis of distributed systems quality-of-service (QoS) metrics via system execution traces. This is usually a *challenging* problem when dealing with data is collected from many different software components that execute on many different hardware components. UNITE therefore provides the necessary functionality to analysis such metrics so system testers does not have to concern themselves with the accidental and inherit complexities associated with this domain.

7.1 Overview

UNITE is a method and a tool for analyzing system execution traces and validating QoS properties, *e.g.*, performance, scalability, and reliability, whose analysis typically varies across different application domains. UNITE's primary purpose is to support validation for distributed software systems, but it can be used to analyze QoS properties of any software system that generates a system execution trace.

To provide a little more background, a *system execution trace* is a collection of log messages. These log messages are related to an execution of the system during a user-defined time period, or the entire lifetime of the system. Listing 7.1 shows an example system execution trace, which is a sequential listing of log messages for different events that occur throughout the system's execution lifetime. The system execution trace in Listing 7.1 resembles many execution traces that are generated by logging frameworks, such as log4j.

```
activating LoginComponent
...
```

```
LoginComponent recv request 6 at 1234945638
validating username and password for request 6
username and password is valid
granting access at 1234945652 to request 6
...
deactivating the LoginComponent
```

Listing 7.1. An example system execution trace stored in a flat file.

Using the system execution trace above, it is possible to evaluate the authentication time for a user. Likewise, if we many different occurrences of the messages illustrated in Listing 7.1, then we evaluate different statistics related to authentication time, such as average and worst authentication time. Finally, we can view the data trend (*i.e.*, how a metric evolves with respect to time) associated with authentication time. The main challenge, however, is ensuring different data points are correlated with their correct execution trace. Otherwise, there is great chance of producing incorrect results.

With this in mind, this chapter discusses how to use UNITE for evaluating user-defined QoS properties via system execution traces. In particular, this chapter first introduces concepts related to instrumentation and UNITE. This chapter then discusses how to use UNITE. Finally, this chapter

7.2 System Execution Trace Concepts

Table 7.1 shows another sample system execution trace. This time, however, the system execution trace is stored in a central database for offline QoS analysis. You will also notice that this system execution trace contains more information, such as time of day, hostname of origin, and message severity. Although all column in this table are important, we are going to focus on the `Message` column when discussing the following concepts related to system execution traces. This is because data in the `Message` column is of most to the user using UNITE. The other columns are used by UNITE, but they are used internally.

ID	Time of Day	Hostname	Severity	Message
10	2011-02-25 05:15:55.34	s.cs.iupui.edu	INFO	Config: sent event 5 at 120394455
11	2011-02-25 05:15:55.35	s.cs.iupui.edu	INFO	Config: sent event 6 at 120394465
12	2011-02-25 05:15:55.38	r.cs.iupui.edu	INFO	Effector: received event 5 at 120394476
13	2011-02-25 05:15:55.46	s.cs.iupui.edu	INFO	Config: sent event 7 at 120394480
14	2011-02-25 05:15:55.37	r.cs.iupui.edu	INFO	Effector: received event 6 at 120394488
15	2011-02-25 05:15:55.52	r.cs.iupui.edu	INFO	Effector: received event 7 at 120394502

Table 7.1. An example system execution trace stored in a central database.

- **Log Format** – Each log message in the system execution trace is constructed from a well defined abstraction called a *log format*. More specifically, log formats are high-level constructs that capture both constant and variable portions of individual, but similar log messages in a system execution trace. For example, Listing 7.2 highlights two different log formats which capture the log messages from the system execution trace in Table 7.1. In this example, `LF1` represents log messages 10, 11, and 13 in Table 7.1; whereas, `LF2` represents log messages 12, 14, and 15 in Table 7.1.

```
LF1: {STRING cmp_id}:sent event {INT event_id} at {INT sent}
LF2: {STRING cmp_id}:received event {INT event_id} at {INT recv}
```

Listing 7.2. Example log formats that represent different log messages in a system execution trace.

The variable portions of the log format is specified inside placeholders {}. The information captured in the variable portions of a log format represent metrics that are very useful in QoS analysis. In the above example, the the time variables `LF1.sent` and `LF2.recv` can be used to determine the latency of different events. Table 7.2 shows the data types currently supported by UNITE.

Each log format consists of two parts: *data type* and *variable name*. The data types gives UNITE an hint as to what kind of data it should be expecting for that placeholder. It is possible to consider all data as a string, but this can complicate the identification process when different data types are string together (*e.g.*, Host5 which is {`STRING hostName`}{`INT hostId`}). The variable name is used when constructing the dataflow graph and the expression for evaluating the user-defined QoS property, which are both discussed later. Finally, Table 7.2 lists the different data types currently supported by UNITE.

Type	Description
INT	Integer data type
LONG	Long data type
STRING	String data type (There cannot be spaces)
FLOAT	Floating-point data type

Table 7.2. List of data types currently supported in UNITE along with their description.

- **Log Format Relation** – The other main concept in UNITE is Log Format Relations. Log Format Relations define a cause-effect relationship among two log formats. For example in the above example `LF1` represents a sent event in the distributed system. And `LF2` represents

a receive event in the system. The sent event always happens before the receive event. There for the log message corresponds to a particular sent event occurs before the corresponding receive event. We say `LF1` is the cause log format and `LF2` is the effect log format. In order to relate the two log messages distributed system testators can use the *event_id* variable as follows.

```
LF1.event_id = LF2.event_id
```

- **Dataflow Graph** – The dataflow graph
- **QoS Analysis Expression** – After specifying Log formats and Log format relations distributed system testers have to specify a QoS analysis equation to get the results. For example in the above example in order to get the average response time following equation can be used.

```
AVG(LF2.recv - LF1.sent)
```

The next section 7.3 shows how to specify above mentioned details using the configuration files in UNITE.

7.3 Defining a Datagraph File

UNITE uses two xml configuration files called, *datagraph file* and *unite file* for the configurations. The *datagraph file* is used to specify the details of log formats and relations. The *unite file* contains a reference to the *datagraph file* and is used for the QoS analysis.

Listing 7.3 shows a sample *datagraph file*.

```
1  <?xml version="1.0" encoding="utf-8" standalone="no" ?>
2  <cuts:datagraph xmlns="http://cuts.cs.iupui.edu"
3                  xmlns:cuts="http://cuts.cs.iupui.edu"
4                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5                  xsi:schemaLocation="http://cuts.cs.iupui.edu cutsnode.xsd">
6   <logformats>
7    <logformat id="LF1">
8     <value>{STRING cmp_id}:sent event {INT event_id} at {INT sent}</value>
9      <relations>
10      <relation effectref="LF2">
11       <causality cause="cmp_id" effect="cmp_id"/>
12        <causality cause="event_id" effect="event_id"/>
13         </relation>
14       </relations>
15    </logformat>
```

```
16     <logformat id="LF2">
17       <value>{STRING cmp_id}:received event {INT event_id} at {INT recv}</value>
18     </logformat>
19   </logformats>
20 </cuts:datagraph>
```

Listing 7.3. An example configuration file for UNITE datagraph.

7.3.1 Datagraph Schema Definition

This section discusses the details of the UNITE datagraph configuration based on its underlying XML Schema Definition.

`<cuts:datagraph>`

This is the main tag for the datagraph configuration document. Its XML namespace must have the `http://cuts.cs.iupui.edu` definition. If it does not have this definition, then UNITE will not execute further.

`<name>`

This tag contains the name of the datagraph. This tag does not have any child tags.

`<adapter>`

This tag contains the location of an external adapter module required for for UNITE to correctly analyze the QoS properties. This tag is not mandatory. It only needs when the system execution trace does not contain required properties for the QoS analysis. See Chapter 8 for more details.

`<logformats>`

This tag is a container for set of log formats that will be used in QoS analysis process. It may contain any number of logformat elements. Log formats can be specified in any order .In general, cause log format is specified first. Then its effect log format is specified. The following is a list of child tags: `<logformat>`.

`<logformat>`

This tag contains all the details about a particular log format. It has the following attributes.
The following is a list of child tags: `<value>`, `<relations>`.

Name	Type	Default Value	Required	Description
id	String		Yes	Name of the log format

<value>

This tag contains the actual value of the log format. Value of a log format represent set of log messages in the system execution trace which have constant and variable parts as described in section (ref) This tag does not have any child tags.

<relations>

This tag contains set of relations, in which the parent log format is involved in as the cause log format. The following is a list of child tags: `<relation>`.

<relation>

This tag contains all the details about a particular relation. Such as the effect log format and cause-effect variables. There can be more than one cause-effect variable pairs. It has the following attributes.

Name	Type	Default Value	Required	Description
effectref	String		Yes	The id of the effect log format in this relation

The following is a list of child tags: `<causality>`.

<causality>

This tag represent the relationship between a variable in the cause log format and a variable in the effect log format. It has the following attributes.

Name	Type	Default Value	Required	Description
cause	String		Yes	The name of the cause variable
effect	String		Yes	The name of the effect variable

Therefore the elements mentioned above represent the datagraph for a partiucar system execution trace. Depending on the QoS properties being analyzed a particular system execution trace may have more than one datagraphs. These different datagraphs may have different log formats and cause-effect relations.

7.4 Defining a UNITE Configuration File

In general the location of the *datagraph file* is specified in the *unite file*. Listing 7.4 shows a sample *unite file*.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <cuts:test xmlns="http://cuts.cs.iupui.edu"
3              xmlns:cuts="http://cuts.cs.iupui.edu"
4              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5              xsi:schemaLocation="http://cuts.cs.iupui.edu cuts-unite.xsd">
6
7    <name>example</name>
8    <datagraph location="example.datagraph" />
9    <evaluation>LF2.recv - LF1.sent</evaluation>
10   <aggregation>AVG</aggregation>
11   <grouping>
12          <groupitem name="LF1_task" />
13   </grouping>
14   <services>
15   <service id="gnuplot">
16    <location>CUTS_Gnuplot_Presentation_Service</location>
17    <classname>CUTS_Gnuplot_Presentation_Service</classname>
18    <params>-o gnuplot --disable-group-title</params>
19   </service>
20  </services>
21 </cuts:test>
```

Listing 7.4. An example configuration file for UNITE.

7.4.1 UNITE Schema Definition

This section discusses the details of the UNITE configuration based on its underlying XML Schema Definition.

`<cuts:test>`

This is the main tag for the unite configuration document. Its XML namespace must have the `http://cuts.cs.iupui.edu` definition. If it does not have this definition, then UNITE will not execute further.

<name>

This tag contains the name of the unite file. This tag does not have any child tags.

<datagraph>

This tag specify the location of the associated datagraph for the QoS analysis. This tag has the following attributes.

Name	Type	Default Value	Required	Description
location	String		Yes	The location of the datagraph configuration file.

This tag does not have any child tags.

<evaluation>

This tag contains the equation for QoS evaluation. The text inside this tag should be written in a certain way. The variables of this equation should be specified with the log format it belongs to. The dot operator after the log format specified the actual variable. This tag does not have any child tags.

<aggregation>

This tag specifies a function used to convert a dataset to a single value. Some examples of aggregation functions are AVERAGE, MIN, MAX and SUM. This tag does not have any child tags.

<grouping>

For a given aggregation function this tag tell how to classify datasets that are independent of each other. This is the container tag for such grouping items The following is a list of child tags: `<groupitem>`.

<groupitem>

This tag represent an actual grouping item. The *name* attribute specify the grouping item. This attribute consists of a particular log format id and a variable name. Their should be a _ between the log format and the variable. This tag has the following attributes.

Name	Type	Default Value	Required	Description
name	String		Yes	The name of the grouping item.

Name	Type	Default Value	Required	Description
name	String		Yes	The name of the grouping item.

<services>

UNITE has a facility to show the results of QoS analysis using different presentation methods such as graphs. There are many different presentation softwares available such as Microsoft Excel, Gnuplot. This tag contains the set of such services the testers want present the analyzed results. This tag has the following child tags:<service>

<service>

This tag represent a presentation service. In general these presentation services are developed externally to UNITE and plugged in via a common interface. Therefore child tags of this tag gives the details required to load these services during the runtime. This tag has the following attributes. The following is a list of child tags: <location>, <classname>, <params>.

<location>

This tag specify the location of a presentation service module. UNITE will search for this location when it wants to load the presentation service. This tag does not have any child tags.

<classname>

This tag specify the name of the class of the presentation service. This tag does not have any child tags.

<params>

This tag specify the additional parameters required to invoke the presentation service. This tag does not have any child tags.

7.5 Invoking UNITE

Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS UNITE tool is installed at the following location:

```
%> $CUTS_ROOT/bin/cuts-unite
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-unite --help
```

A typical invocation of UNITE tool requires the data file which contains the system execution trace and the UNITE configuration file discussed above as arguments. The datagraph file location is specified inside the UNITE configuration file so it doesn't need to be specified as an argument. Assuming the configuration discussed above is defined in a file named *example.unite*, and the system execution trace is resides in a file called *example.cdb* you can run the CUTS UNITE tool with the configuration via the following command:

```
%> $CUTS_ROOT/bin/cuts-unite -f example.cdb -c example.unite
```

Chapter 8

System Execution Trace Adaptation Framework (SETAF)

This chapter discusses a tool and a method called *System Execution Trace Adaptation Framework (SETAF)* in the CUTS analysis tool set. This tool is used as an extension to UNITE when analyzing QoS properties from system execution traces.

8.1 Overview

Chapter 7 discussed the method of UNITE when doing the QoS analysis using system execution traces. In some situations UNITE cannot be used alone to analyze system execution traces. This is because UNITE assumes that every different instance of a log format is unique. Which means it assumes that at least one log format variable other than the variable represent time have different values among different instances. This causes non-unique relations among different instances of log formats.

```
Started doing task A at 12.00
Finished doing task A at 12.01
Started doing task A at 12.02
Finished doing task A at 12.03
```

Listing 8.1. Portion of a system execution trace that does not contain unique relations .

For example, Listing 8.1 illustrates an example system execution trace where the dataflow graph will not have unique relations between the log format. This is because it is *hard* to know start/finish messages are associated with one another without human intervention. Moreover, when an example similar to the one present in Listing 8.1 is analyzed by UNITE, it will yield incorrect results because it is *hard* to determine correct causality between similar log messages.

With proper planning early in the software lifecycle, it is possible to ensure generated system execution traces have unique relations to facilitate proper analysis. This, however, is not alway possible—especially when analyzing system execution traces generated by third-party systems and their components. Although such system execution traces may not contain unique relations, the existing relations can be exploited (or adapted) to enforce a unique relation. For example, in Listing 8.1, although the relation is not unique, it can be adapted to be a unique relation by adding an id to each log message. This will ensure that UNITE analyzes the dataflow model and evaluates the expression correctly.

Adding this kind of new ids means we have to add new log format variables to the identified log formats. Further new log format relations need to be identified for these newly added variables. Even though the ordinary log format variables get values from the system execution trace, it does not contain the values for these externally added log format variables. Further the variables need to be added, the relations need to be enforced and what sort of values they get depend on the system execution trace being analyzed. Therefore UNITE provides a common interface for these task, but the implementation module should take care of how to adapt the dataflow model. Distributed system testers need to find out the adaptation patterns and need to provide these external adapters.

Distributed system testers do not need to develop these modules using a language like C++. Instead SETAF provides a simple language to specify them as a adaptation specification. SETAF has a code generation tool to generate the source code for these adapters. After compiling these adapter modules, they need to be specified in the datagraph file so that UNITE can load them during the run time. The Figure 8.1 showcase the overall workflow of SETAF.

The rest of the chapter describes the content of the UNITE Adapter Specification in Section 8.2. In Section 8.3 it describes the procedure of code generation of adapter modules in SETAF. Finally Section 8.4 shows how to use SETAF with UNITE to do the QoS analysis.

8.2 UNITE Apater Specification

This section explains the content of Unite Adapter Specification. Listing 8.2 shows an example of UNITE Adapter specification.

```
1 Variables:
2  int id_;
3
4 Init:
5  id_ = 0;
6
7 Reset:
8  id_ = 0;
9
```

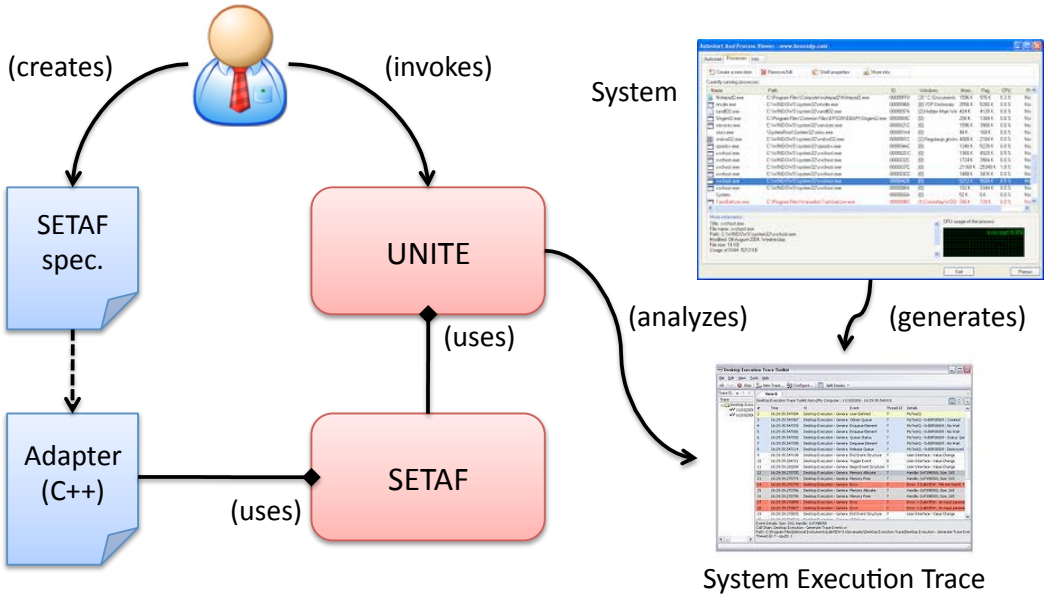


Figure 8.1. Conceptual overview of SETAF's workflow.

```
DataPoints:
 int LF1.uid;
 int LF2.uid;
 string LF2.cmp_name;
 string LF3.cmp_name;

Relations:
 LF1.uid->LF2.uid;
 LF2.cmp_name->LF3.cmp_name;

On LF1:
 id_++;
 int32_var * ivar = dynamic_cast <int32_var *> (vars ["uid"]);
 ivar->value (this->id_);

On LF2:
 vars ["cmp_name"]->value ("foo");

 id_++;
 int32_var * ivar = dynamic_cast <int32_var *> (vars ["uid"]);
```

```
30  ivar->value (this->id_);
31
32 On LF3:
33  vars ["cmp_name"]->value ("foo");
34
35  id_++;
36  int32_var * ivar = dynamic_cast <int32_var *> (vars ["uid"]);
37  ivar->value (this->id_);
```

Listing 8.2. An example UNITE Adapter Specification.

As shown in the listing the UNITE Adapter Specification has five sections (`Variables, Init, Reset, Datapoints, Relations, Adaption code`). Following describes what data need to be specified in these different sections. Please refer to the example provided above when reading the section below.

- **Variables** - The adaptation module keeps the state of the adaption using the variables specified in this section. These variables will be converted to private variables in the generated C++ code. Let's call them state variables. These state variables are used to populate values for the newly added log format variables based on the state.

- **Init** - The initial state of the adapter is specified in this section. The declared variables in the above *Variables* section are initialized in this section.

- **Reset** - Sometimes the variables representing the adapter state may need to be reset. For example same state variable may be used to populate values for different log format variables. In this case sometimes the state variables need to reset when switching from one log format to the other. This section is used for that.

- **DataPoints** - This section specifies the log format variables need to be added for the adaption. Each entry has a type and an identifier. The identifier has two parts, which are separated by a "." . Left side of the separator contains the log format id and the right side contains the name of the newly added log format variable. For an example `LF1.uid` means a new variable named `uid` is added to the `LF1` log format. The type part specified the type of the log format variable. These log format variables can have following types.

- **Relations** - This section specifies the relations which need be enforced related to the the log format variables added in the `DataPoints` section. Each relation entry has two parts which are separated from a $\rightarrow$. The left side of the $\rightarrow$ specifies the cause log format variable. And the right side of $\rightarrow$ specifies the effect log format variable.

Table 8.1. UNITE data types

Type	Description
INT	Integer data type
LONG	Long data type
STRING	String data type
FLOAT	Floating-point data type

- **Adaption code** - The adaption code is where the domain-specific logic resides for the adaption pattern. The adaption code is segmented based on the log formats that must undergo adaption. Each segment dictates how to update variables in the dataflow graph, as well as its own private variables. For string variables the name of the variable and the value of the variable are sufficient. For other types of variables a dynamic cast need to be done as specified in the above example.

8.3 SETAF code generation

The above descrIbed UNITE adapter specification will resides in a file which has *.uas* extension. SETAF has a tool called *cuts-setaf* to generate the adaptation source code using the adapter specification described above. Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS SETAF tool is installed at the following location:

```
%> $CUTS_ROOT/bin/cuts-setaf
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-setaf --help
```

For the code generation process two arguments need to be provided. The location of the UNITE adapter specification file and the directory location, to output the generated source code. As an example following invocation will generate the source code using the *example.uas* file and the generated code will be output to the current directory.

```
%> $CUTS_ROOT/bin/cuts-setaf -f example.uas -o .
```

This execution will output four files - *source file, header file, project file and workspace file*. As an example the above invocation will generate *example.h, example.cpp, example.mpc and example.mwc* files. Depending on the user preferred compiler the workspace for compiling the code can be created using Makefile, Project, and Workspace Creator (MPC). As an example following shows how to create workspace for gcc and Visual Studio 2008 compilers. For gcc:

```
%> mwc.pl -type gnuace example.mwc
```

For Visual Studio 2008:

```
%> mwc.pl -type vc9 example.mwc
```

In Linux like systems this will generate a dynamic link library with *.so* extension. Therefore in this case it will be example.so. In windows like systems this will create a dynamic link library with *.dll* extension. Therefore in this case it will be example.dll.

8.4 QoS analysis with UNITE

Section 8.4 described the process of generating the adapter module (the dynamic link library). The location and the name of this library is specified in the `<adapter>` tag of the datagraph file. See Section 7.3 for further details of datagraph file. UNITE will then load the adapter module from this location and use its functionality when required.

Chapter 9

Dataflow Model Auto Construction (DMAC) Tool

Chapter 7 discussed about the QoS analysis of distributed systems using UNITE. One of the major difficulty when using UNITE tool is coming up with the dataflow model for a particular system execution trace. UNITE assumes that the distributed system testers are going to specify the dataflow model for the QoS analysis. *Dataflow Model Auto Construction (DMAC)* is a tool and a process for auto constructing the dataflow model for a particular system execution trace based on the frequently occurring log formats.

9.1 Overview

For a particular system execution trace many log formats can be defined. Even though there are many log formats , few of them are useful for the QoS analysis. Finding useful log formats and relations for QoS analysis requires examining the system execution trace and a good understanding about the system being analyzed. For system execution traces which are larger in size, this is a very difficult process. Further this can result in identifying less important log formats as important log formats, missing important log formats and deciding variable parts of a log formats as static parts.

One heuristic used in constructing the dataflow model is finding out frequent log formats and relations between them. When a log format is frequent there is a very high chance that it is included in the dataflow model. Because such a log format represent an event that occurred frequently. Finding the QoS properties related to this kind of events, gives a very good overview about the system performance through out its execution period.

Figure 9.1 shows the overall process of auto constructing the dataflow model in DMAC. Section 9.2 describes the different steps in this process.

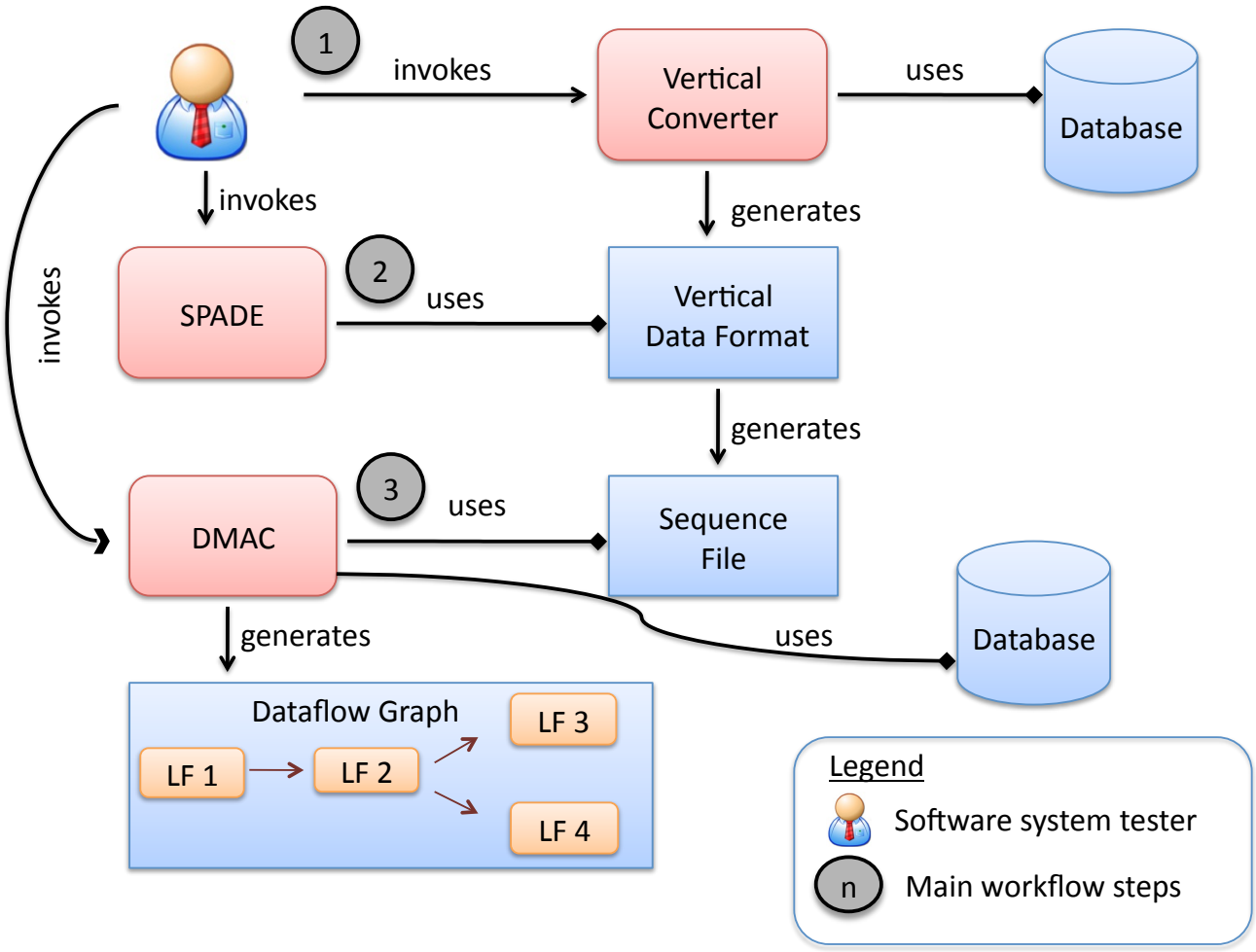


Figure 9.1. Dataflow model auto-generation process.

9.2 Process of auto constructing the dataflow model

This section describes different steps involved in auto constructing the dataflow model using DMAC.

9.2.1 Finding frequently occurring word sequences

As the first step frequent-sequence mining need to carried out to find out the frequently occurring word sequences in the system execution trace.

In data mining, frequent-sequence mining is the process of identifying frequently occurring sequences in a collection of time-stamped event-set such as purchased items in a transaction database. For example a *customer-sequence* consists of a sequence of all transactions related to a particular customer. Customer-sequences are normally sorted on increasing transaction time. Finally, a customer-sequence supports a sequence s if s is a sub-sequence of that particular customer-sequence.

To identify frequently occurring sequences, users have to supply a minimum number of customer-sequences. This is called *minimum support* denoted by *min-sup*. For a given database frequent-sequence mining locates sub-sequences that occur more than *min-sup* customer-sequences.

In the context of system execution traces, we consider a customer-sequence an execution trace that consists of words. By words, we mean the items that can be separated by a delimiter (*e.g.*, space and new-line).

DMAC uses SPADE (`http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software`) sequence mining tool to find out the frequently occurring word sequences.

Table 9.1. Sample system execution trace.

SID	message
1	A sent message 1 at 10.00
2	A received message 1 at 10.01

SPADE requires data to be in a format called *vertical id-list*. Table 9.1 contains two sample messages with a `Sequence Id (SID)` and the actual message. The corresponding vertical id-list for the system execution trace in Table 9.1 is shown in the Table 9.2. In addition to the `SID`, Table 9.2 contains `Event ID (EID)`, which serves as a timestamp within the sequence, and a `Size` field, which denotes how many items are in the current transaction. In the context of a system execution trace, the `EID` is a monotonically increasing value and the `Size` is always one. Finally, a word is considered as an item set and there is always just one item in any transaction.

DMAC has a tool called *cuts-dmac-vertical* to convert the system execution trace into *vertical id-list*. Assuming the CUTS runtime architecture has been built and installed correctly, the CUTS DMAC vertical converter tool is installed at the following location:

Table 9.2. Vertical data format for the system execution trace.

SID	EID	Size	Items
1	1	1	A
1	2	1	sent
1	3	1	message
1	4	1	1
1	5	1	at
1	6	1	10.01
2	1	1	A
2	2	1	received
2	3	1	message
2	4	1	1
2	5	1	at
2	6	1	10.01

```
%> $CUTS_ROOT/bin/cuts-dmac-vertical
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-dmac-vertical --help
```

Giving the database file as the input *cuts-dmac-vertical* will convert the system execution trace into a vertical id-list ,which can be redirect into a file ending with `.data` extension as follows.

```
%> $CUTS_ROOT/bin/cuts-dmac-vertical -f example.cdb > example.data
```

The next step is finding the frequently occurring word sequence using the *SPADE* data mining tool. DMAC uses the version of *SPADE* comes with the *Data Mining Template Library (DMTL)* (http://sourceforge.net/projects/dmtl/) data mining tool. DMTL can be downloaded from the following location http://sourceforge.net/projects/dmtl/ . After extracting the DMTL distribution change, to the directory named *test* inside the distribution and use the following command to to find out the command line options.

```
%> $DMTL_INSTALL_DIR/test/sequence_test -h
```

To find out the frequent-sequences which have a equal or higher occurrence than the provided *min-sup* value, SPADE can be executed providing the *min-sup* value and the vertical-id list of the system execution trace as follows.

```
%> $DMTL_ROOT/test/sequence_test -i example.data -s 100 -p > example-sequence
```

The `-i` option specifies the vertical-id list created in the previous step, `-s` option specifies the *min-sup* value and *-p* option specifies the output to be printed to the standard input. In the above

case the output is redirected to a file called *example-sequence*. This file is fed to DMAC to generate the dataflow model in the next step.

9.2.2 Auto generating the dataflow model

Having find out the frequent-sequences of a particular system execution trace the next step is to generate the dataflow model which has log formats and cause-effect relations. This step is done using a tool called *cuts-dmac*.

Assuming the CUTS runtime architecture has been built and installed correctly, the *cuts-dmac* tool is installed at the following location:

```
%> $CUTS_ROOT/bin/cuts-dmac
```

To see a complete list of command-line options, use the following command:

```
%> $CUTS_ROOT/bin/cuts-dmac-vertical --help
```

Using the system execution trace and the frequent-sequence file (eg: example-sequence) generated in the previous step, following is how to generate the dataflow model for the given system execution trace.

```
%> $CUTS_ROOT/bin/cuts-dmac -i example-sequence -f example.cdb -n example
```

This will output the possible log formats and relations which will be useful for QoS analysis. Further it will create a datagraph file using the argument provided for `-n`. For the abobve invocation it will create a datagraph file named *example.datagraph*. This datgraph contains all the possible log formats and the cause-effect relations among them based on the frequent-sequence mining. Distributed system testers can prune this datagraph file further to find out the QoS properties depending on the scenario been analyzed. The above invocation also output the coverage (as a percentage) of each log format in the system execution trace. This is to assist distributed system testers to take decisions on pruning the generated datagraph file.

Part V

Appendix

Appendix A

Building and Installing Third-Party Libraries

The following appendix contains best practices for building and installing third-party libraries used by CUTS projects. It is not a requirement to install third-party libraries per the instructions in this appendix. If you are experiencing compilation problems with CUTS due to third-party libraries, however, the instructions in this appendix may be of assistance. To make help explain the installation process of the thirdparty libraries, it will be assumed that all third-party libraries will be installed at the following location: `/opt/thirdparty`[1] [2].

A.1 The Makefile, Project, Workspace Creator (MPC)

The *Makefile, Project, Workspace Creator (MPC)* is a command-line utility for generating workspaces. It is required to build different third-party libraries. and CUTS. The artifacts for MPC can be downloaded from the following location: `https://svn.dre.vanderbilt.edu/DOC/MPC/trunk`.

A.1.1 System Configuration

MPC does not need to be built since its a PERL application. Instead, we must properly set environment variables to use MPC correctly. Please therefore define the following environment variables:

[1] This location is representative of a common location for installing third-party libraries, and may be different on your system.

[2] If you are installing CUTS from source on Windows, it is recommended that you place all environment variables in a BATCH file (`.bat`) so you do not pollute the system environment.

```
export MPC_ROOT=location of MPC
export PATH=$PATH:MPC_ROOT
```

A.1.2 Installation

There are no separate installation procedures for MPC.

A.2 Boost

The Boost libraries are required when building and installing CUTS. All artifacts for Boost, such as prebuilt versions and source files, can be downloaded from the following location: http://www.boost.org. Currently XSC seems to only work with Boost 1.43.0.

A.2.1 System Configuration

To build CUTS correctly, the Boost environment variables must be defined. Please define the following required environment variables:

```
%> export BOOST_ROOT=/opt/thirdparty/boost
%> export BOOST_VERSION=boost-1_34_1
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$BOOST_ROOT/lib
%> export PATH=$PATH:$BOOST_ROOT/bin
```

In some cases, Boost libraries will have an optional configuration appended to their filename, such as `-mt-d`. To use a specific configuration of Boost, the `BOOST_CFG` environment variable must be defined. The following is an example of defining the Boost configuration environment variable:

```
%> export BOOST_CFG=-mt-d
```

The configuration of Boost is now complete.

A.2.2 Installation

Non-Windows

If you are installing Boost on a non-Windows system, *e.g.*, Linux, Soloris, or MacOS X, then you can download the source code directly from its main website (http://www.boost.org), build it, and install it using standard procedures.

As a quick reference, you can quickly build Boost using the following sequence of commands:

```
%> cd $BOOST_ROOT
%> ./bootstrap.sh --prefix=$BOOST_ROOT
%> ./bjam --without-python
%> ./bjam --install
```

This will build Boost using the preferred method, and i Finally, please be patient when compiling Boost from sources because it does take at least 30 minutes.

Windows

Installing Boost from sources on Windows is not a trivial process. Moreover, the prebuilt version of Boost available for download from the main website does not contain the correct configuration or directory layout. It is therefore recommended that you download and use the prebuilt version of Boost available for download from the CUTS website at the following location: http://www.dre.vanderbilt.edu/CUTS/downloads/thirdparty/boost. Please download the correct version of Boost that matches your version of Microsoft Visual C++ and extract it to a common location on disk.

A.3 Xerces-C

Xerces-C 3.x is required when building and installing CUTS and other third-party libraries. All artifacts for Xerces-C, such as prebuilt versions and source files, can be downloaded from the following location: http://xerces.apache.org/xerces-c.

A.3.1 System Configuration

To build CUTS correctly, the Xerces-C environment variables must be defined. Please define the following required environment variables:

```
%> export XERCESCROOT=/opt/thirdparty/xerces-c
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$XERCESCROOT/lib
```

The configuration of Xerces-C is now complete. Once all the other third-party libraries are installed you will be able to build CUTS (see Chapter 1).

A.3.2 Installation

Non-Windows

If you are installing Xerces-C on a non-Windows system, *e.g.*, Linux, Soloris, or MacOS X, then you can download the source code directly from its main website (http://xerces.apache.org/xerces-c), build it, and install it using standard procedures.

Windows

Installing Xerces-C from sources on Windows is not a trivial process. Moreover, the prebuilt version of Xerces-C available for download from the main website does not contain the correct configuration or directory layout. It is therefore recommended that you download and use the prebuilt version of Xerces-C available for download from the CUTS website at the following location: http://www.dre.vanderbilt.edu/CUTS/downloads/thirdparty/xerces-c. Please download the correct version of Xerces-C that matches your version of Microsoft Visual C++ and extract it to a common location on disk.

A.4 SQLite

SQLite is required when building and installing several projects in CUTS. You can download the source code for SQLite from the following location: http://www.sqlite.org

A.4.1 System Configuration

To build CUTS correctly, the SQLite environment variables must be defined. Please define the following required environment variables:

```
%> export SQLITE_ROOT=/opt/thirdparty/SQLite
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SQLITE_ROOT/lib
%> export PATH=$PATH:$SQLITE_ROOT/bin
```

The configuration of SQLite is now complete. Once all the other third-party libraries are installed you will be able to build CUTS (see Chapter 1).

A.4.2 Installation

Non-Windows

If you are installing SQLite on a non-Windows system, *e.g.*, Linux, Soloris, or MacOS X, then you can download the source code directly from its main website (http://www.sqlite.org), build it, and install it using standard procedures.

Windows

Installing SQLite from sources on Windows is not a trivial process. It is therefore recommended that you download and use the prebuilt version of SQLite available for download from the CUTS website at the following location: http://www.dre.vanderbilt.edu/CUTS/downloads/thirdparty/sqlite. Please download the correct version of SQLite that matches your version of Microsoft Visual C++ and extract it to a common location on disk.

A.5 XML Schema Compiler (XSC)

XSC is required when building and installing CUTS. All artifacts for XSC can be downloaded from the following location: svn://svn.dre.vanderbilt.edu/XSC/trunk. The remainder of this section discusses how to build and install XSC.

A.5.1 System Configuration

You must configure you system installing XSC. Once you have downloaded XSC from the website above please define the following environment variables:

```
%> svn co svn://svn.dre.vanderbilt.edu/XSC/trunk XSC
%> export XSC_ROOT=/opt/thirdparty/XSC
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$XSC_ROOT/lib
%> export PATH=$PATH:$XSC_ROOT/bin
```

A.5.2 Installation

After you have configured your system for XSC, you are ready to build and install it. Please use the following steps to build and install XSC:

```
%> cd $XSC_ROOT
%> $ACE_ROOT/bin/mwc.pl -type [build tool] -features xsc=1,xerces3=1 XSC.mwc
```

This will generate the workspace for building and installing XSC. The name of the workspace depends on the build tool specified when generating the workspace. Finally, using your build tool of choice, you can build and install XSC.

A.6 Perl-Compatible Regular Expressions (PCRE)

PCRE is required when building and installing several projects in CUTS. You can download the source code for PCRE from the following location: http://www.pcre.org

A.6.1 System Configuration

To build CUTS correctly, the PCRE environment variables must be defined. Please define the following required environment variables:

```
%> export PCRE_ROOT=/opt/thirdparty/pcre
%> export PCRE_VERSION= version of PCRE, blank if not applicable
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PCRE_ROOT/lib
```

The configuration of PCRE is now complete. Once all the other third-party libraries are installed you will be able to build CUTS (see Chapter 1).

A.6.2 Installation

Non-Windows

If you are installing PCRE on a non-Windows system, *e.g.*, Linux, Soloris, or MacOS X, then you can download the source code directly from its main website (http://www.pcre.org), build it, and install it using standard procedures.

Windows

Installing PCRE from sources on Windows is not a trivial process. It is therefore recommended that you download and use the prebuilt version of PCRE available for download from the CUTS website at the following location: http://www.dre.vanderbilt.edu/CUTS/downloads/thirdparty/pcre. Please download the correct version of PCRE that matches your version of Microsoft Visual C++ and extract it to a common location on disk.

A.7 DOC Group Middleware

DOC Group Middleware is required when building and installing CUTS. Its source code can be downloaded from the following location: http://www.dre.vanderbilt.edu/CIAO.

A.7.1 System Configuration

Unlike Boost, you must manually configure your system before installing the DOC middleware. Once you have downloaded the tarball from the website above[3], please define the following environment variables:

```
%> export ACE_ROOT=/opt/thirdparty/Middleware/ACE
%> export TAO_ROOT=/opt/thirdparty/Middleware/TAO
%> export CIAO_ROOT=/opt/thirdparty/Middleware/CIAO
%> export ADBC_ROOT=/opt/thirdparty/Middleware/ADBC
%> export DANCE_ROOT=/opt/thirdparty/Middleware/DAnCE

%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/lib:$ADBC_ROOT/lib
%> export PATH=$PATH:$ACE_ROOT/bin:$CIAO_ROOT/bin:$DANCE_ROOT/bin
```

After defining the environment variables above, the next step is configuring the middleware for the target platform. This is accomplished by defining the `$ACE_ROOT/ace/config.h` header

[3] It is recommended you download either the latest version or snapshot of trunk from the SVN repo. Otherwise, you will not have access to ADBC.

file, which will include the correct configuration for your platform. The included configuration file has the form `config-*`, where * is the platform of choice. The following is an example `config.h` file for building DOC middleware on Linux platforms:

```
// -*- C++ -*-

#ifndef _ACE_CONFIG_H_
#define _ACE_CONFIG_H_

#include ''config-linux.h''

#endif // !defined _ACE_CONFIG_H_
```

Non-Windows

Configuring the DOC middleware on non-Windows systems requires one more step. The configuration for the build engine also needs to be defined. Please use the following step to define the build engine's configuration:

```
%> cd $ACE_ROOT/include/makeinclude
%> ln -s platform_*.GNU platform_macros.GNU
```

where * is the architecture for your platform of choice.

A.7.2 Installation

After you have configured your system, you are ready to build and install ACE and TAO. Please use the following steps to build and install the DOC middleware:

```
%> cd $CIAO_ROOT
%> $ACE_ROOT/bin/mwc.pl -type [build tool] -features xerces3=1 CIAO_TAO_DAnCE.mwc
%> build the workspace
```

A.7.3 ACE DataBase Connector (ADBC) Framework

The ACE DataBase Connector (ADBC) Framework is a set of C++ wrappers that provide a common interface for using different database drivers. To install ADBC, first copy the file `default.features.tmpl` to `default.features`. Open the new file and then enable/disable the necessary features based on what wrappers you want to build.[4] Finally, generate the workspace and build ADBC using the following commands:

[4] It is assumed that you have installed the development version for the database drivers that you plan to build.

```
%> cd $ADBC_ROOT
%> $ACE_ROOT/bin/mwc.pl -type [build tool] ADBC.mwc
%> build the workspace
```

A.8 RTI-DDS

RTI-DDS is required if you are building CHAOS. You can obtain RTI-DDS from the following location: http://www.rti.com

A.8.1 System Configuration

In order to use RTI-DDS with CUTS, the environment must be configured a certain way. Otherwise, there is great chance that CUTS emulation code that uses RTI-DDS will not compile correctly. Please define the following environment variables:

```
%> export NDDSHOME=/opt/thirdparty/RTI-DDS
%> export NDDSARCHITECTURE= target arch (see \$NDDSHOME/lib)
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NDDSHOME/lib/$NDDSARCHITECTURE
```

A.8.2 Installation

RTI-DDS comes prebuilt. There are no installation procedures once you unzip the obtained tarball.

A.9 OpenSplice DDS

OpenSplice DDS is required if you are building CHAOS. You can obtain OpenSplice DDS from the following location: http://www.opensplice.org. Please make sure to download the binary version.

A.9.1 System Configuration

In order to use OpenSplice DDS with CUTS, the environment must be configured a certain way. Otherwise, there is great chance that CUTS emulation code that uses OpenSplice DDS will not compile correctly. Please define the following environment variables:

```
%> export OSPL_TARGET=target arch
%> export OSPL_HOME=/opt/thirdparty/OpenSplice/$OSPL_TARGET
%> export OSPL_TEMP_PATH=$OPSL_HOME/etc/idlpp
%> export OSPL_URI="file://$OPSL_HOME/etc/config/ospl.xml"
```

```
%> export CLASSPATH=$OSPL_HOME/jar/dcpssaj.jar;$CLASSPATH
%> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSPL_HOME/lib
```

A.9.2 Installation

OpenSplice DDS is built on top of The ACE ORB (TAO), which is also the case for CUTS. In most cases, however, the version of TAO in either OpenSplice DDS and CUTS is not the same version. Likewise, OpenSplice DDS and CUTS may not be built in such a way that they support different versions in the same process space, which is possible given the correct configuration. Because of this, it is best to rebuild only the custom library of OpenSplice DDS so that both OpenSplice DDS and CUTS can operate in the same address space.

To rebuild the custom library in OpenSplice DDS, first determine the version of TAO installed on your machine. The version number is found in the following file: `$TAO_ROOT/VERSION`. Once you have identified the version number, create the following directories in OpenSplice DDS install directory:

```
%> cp $OSPL_HOME/etc/idlpp/CCPP/[current version DDS_ACE_TAO] \\
  $OSPL_HOME/etc/idlpp/CCPP/DDS_ACE_TAO_x_y_z
%> cp $OSPL_HOME/include/dcps/C++/CCPP/[current version DDS_ACE_TAO] \\
  $OSPL_HOME/include/dcps/C++/CCPP/DDS_ACE_TAO_x_y_z
```

where `x_y_z` is the correct version of TAO installed on your machine. Please make sure to use underscores (_) and not dots (`.`) in the version number. Finally, define the following environment variable so the OpenSplice DDS architecture knows what version of the ORB to use:

```
%> export SPLICE_ORB=DDS_ACE_TAO_x_y_z
```

where `x_y_z` is the correct version of TAO installed on your machine as done above.

The final step in the process is rebuilding the custom library is actually building it. This can be accomplished using the provide MPC project file in CUTS and executing the following steps:

```
%> cd $CUTS_ROOT/contrib/OpenSplice
%> $ACE_ROOT/bin/mwc.pl -type [build type]
%> #build the generated solution
```

The build process will install the custom library in `$OSPL_HOME/lib`.

Building Against TAO 1.7.8 (or Later)

There were several changes to TAO 1.7.8 (or later) that break building OpenSplice against the latest version of TAO. These changes also have not been reflected in OpenSplice. To ensure that the custom library in OpenSplice is able to build against the TAO 1.7.8 (or later), please update

`$OSPL_HOME/include/dcps/C++/CCPP/DDS_ACE_TAO_x_y_z/orb_abstraction.h` with the following content:

```
#include "tao/LocalObject.h"
#include "tao/Version.h"

namespace DDS
{
#if TAO_MAJOR_VERSION <= 1 && TAO_MINOR_VERSION <= 7 && TAO_BETA_VERSION < 8
  #define LOCAL_REFCOUNTED_OBJECT TAO_Local_RefCounted_Object
#else
  #define LOCAL_REFCOUNTED_OBJECT ::CORBA::LocalObject
#endif
  #define LOCAL_REFCOUNTED_VALUEBASE ::CORBA::DefaultValueRefCountBase

#if TAO_MAJOR_VERSION <= 1 && TAO_MINOR_VERSION <= 7 && TAO_BETA_VERSION < 8
  #define THROW_ORB_EXCEPTIONS  ACE_THROW_SPEC ((CORBA::SystemException))
  #define THROW_ORB_AND_USER_EXCEPTIONS(...) \
    ACE_THROW_SPEC ((CORBA::SystemException, __VA_ARGS__))
#else
  #define THROW_ORB_EXCEPTIONS
  #define THROW_ORB_AND_USER_EXCEPTIONS(...)
#endif

  // These definitions seem to be OK.
  #define THROW_VALUETYPE_ORB_EXCEPTIONS
  #define THROW_VALUETYPE_ORB_AND_USER_EXCEPTIONS(...)
}
```

You should now be able to build the custom library against TAO 1.7.8 (or later) without any build errors.

Appendix B

Autobuilds of CUTS

This section contains detailed instructions for hosting an autobuild of CUTS. The autobuild is designed to both build and test core functionality of CUTS so errors can be located in a timely manner. Moreover, hosting an autobuild is designed to be an easy process. Please use these instructions as a starting point. If you need to change any part of the process, please feel free to do so. We ask, however, that you please do not commit your changes back to the repository.

B.1 Hosting a Windows Build

Windows-based builds of CUTS are conducted using CruiseControl (`cruisecontrol.sourceforge.net`), which is a Java-based continuous integration environment. The scripts for hosting a Windows build are already defined for CUTS. You can find them at the following location in the CUTS directory structure:

```
%> $CUTS_ROOT/integration
```

Once CruiseControl is installed in your environment, you can include CUTS in the build cycle in two main steps. First, you must include the CUTS project in CruiseControl's main configuration file (*i.e.*, `config.xml`). The following line is an example of including the Visual Studio 2005 build for CUTS as part of the build cycle.

```
<include.projects
  file="E:/proj/vc8/CUTS/integration/cruisecontrol/cuts-vc8.config.xml" />
```

Configuring the build environment. After including the desired projects in CruiseControl's main configuration file, the next (and final) step is to configure the environment. This is done by first defining the directory structure for the build environment. Each build type (*e.g.*, Visual Studio 2005 and Visual Studio 2008) must have its own sandbox in the target build environment. The sandbox is where all projects for that build type must should reside when building CUTS

via CruiseControl. The following listing shows the directory structure for hosting both a Visual Studio 2005 and Visual Studio 2008 build on the same machine where the sandboxes are located in `E:/proj`.

```
/vc80
  /CUTS
  /ACE_TAO_CIAO
  /XSC
  ...
/vc90
  /CUTS
  /ACE_TAO_CIAO
  /XSC
  ...
```

Once you have established the sandbox locations, you must inform the build environment of its location. This can be done by setting one or more of the following environment variables. Using

Table B.1. Environment variables that capture the location of a sandbox.

Variable Name	Description
`VC80SANDBOX`	Defines the sandbox for Visual Studio 2005 builds
`VC90SANDBOX`	Defines the sandbox for Visual Studio 2008 builds

the example directory structure above, the Visual Studio 2005 would be defined as follows:

```
set VC80SANDBOX=E:/proj/vc8
```

Notice, this environment variable must be defined in a persistent location, such as the user/system environment variables.

Lastly, you must define the appropriate configuration environment variable. This is necessary because each build environment may install the required third-party libraries (see Appendex A) in different locations. The configuration environment variable(s) therefore points to a batch (`.bat`) file that defines all the environment variables need to build CUTS, such as all the environment variables defined in Appendex A. The following table is the list of configuration environment variables based on the build type:

Table B.2. Environment variables that define the build configuration.

Variable Name	Description
`VC80BUILDCFG`	Defines build configuration for Visual Studio 2005 builds
`VC90BUILDCFG`	Defines build configuration for Visual Studio 2008 builds

For example, if Visual Studio 2005 environment variables are defined in `E:/proj/vc9/setenv.bat`, then `VC80BUILDCFG` would be defined as follows:

```
set VC80BUILDCFG=E:/proj/vc8/setenv.bat
```

Notice, this environment variable must be defined in a persistent location, such as the user/system environment variables.

Congratulations, you are now ready to build Windows-based version of CUTS using CruiseControl!