# RESOLVING MIDDLEWARE CONFIGURATION CHALLENGES USING MODEL DRIVEN DEVELOPMENT

By

Emre Turkay

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

August, 2005

Nashville, Tennessee

Approved:

Douglas Schmidt

Aniruddha Gokhale

# ACKNOWLEDGEMENTS

I am grateful to Dr. Douglas Schmidt and Dr. Aniruddha Gokhale for their helps and support throughout my studies. I would also like to thank to all of the DOC group people for their respectful ideas, helps in implementing many of them and answering my never ending questions.

I would like to especially thank to my family for their emotional support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

**Emerging trends.** The technology requirements of various domains are increasing, such as distributed platforms are commonly in use, the quality of service requirements of embedded systems are increasing as the hardware underneath grows powerful, customers ask for platform independent software and interoperability, etc. Application developers mostly rely on third party middleware, tools and libraries (i.e., web servers, distributed middleware such as CORBA, etc.) to respond the emerging trends of their target domain. However, the variability in the application domains and their requirements on the Quality of Service issues enforce middleware to be a one size fits all solution. The answer of the middleware developers to this trend is to provide flexible, highly customizable and open source middleware to the application developers.

**Challanges of customization.** The flexibility and the customization of the middleware reflects to the application developers as options to configure for their specific domain and applications and a highly customizable middleware present a huge configuration space for application developers to learn and modify. For example CIAO (Component Integrated ACE ORB), which is an open source CCM (Corba Component Model) implementation, provides hundreds of options at compilation stage and for setting up the run-time behavior of the ORB (Object Request Broker). Obviously, the configuration of such a huge set of parameters require a great deal of knowledge and produce various challenges which we can classify as follows:

1. **Lack of synchronized documentation.** The application developers may not have in depth information about the techniques used in the middleware and

certainly it can not be expected them to know the infrastructure and the implementation details of the middleware. Ideally, the middleware is shipped with all the required information for application developers to configure and customize the middleware. However, the configuration options and the assumptions about them may not be synchronized with the documentation, may be outdated and even the documentation may not exist for some cases.

2. **Accidental complexities of configuration.** Providing documentation about the middleware configuration does not reduce the accidental complexity of configuring middleware. The specific way of configuring various parts of middleware provides potential misuse and it may cause run-time failures or performance decrease. If the middleware used is open source software the configuration process may require writing source files before the compilation process, for example users of CIAO may require to edit various macro and C++ header files manually for customazing the middleware for certain behaviors or platforms.

3. **High load to the middleware developers.** Writing detailed documentation and updating it with the evolving implementation put a high load to the middleware developers. This process requires extensive work and research, and therefore may be omitted by the middleware developers. Furthermore, the MW developer may not have enough information or visualization about the platforms where the system works.

The problems described above are shared among DRE applications, and a way to resolve these challenges is to provide tools to simplify and automate the configuration and customization process. For this purpose we have focused on *model-driven development* (MDD) methodology and developed Option Configuration Modeling Language (OCML) for the use of both middleware developers and application developers. In

this thesis OCML with its various aspects and features and how is it applied to resolve the described challenges is explained.

**Thesis organization.** In Chapter II we analyze the MDD approach to the middleware configuration and describe the modeling aspects of OCML. In Chapter III we describe how do we resolve the accidental complexities of middleware configuration with the OCML Configurator user interface and demonstrate the constraint enforcing mechanism. In Chapter IV we define the Configuration and Customization patterns and how we minimize the load to the middleware developers. In Chapter V we describe the generative aspects of OCML. In Chapter VI we demonstrate the use of OCML with examples and present result analysis.

# CHAPTER II

# MODELING THE CONFIGURATION SPACE

The documentation about the configuration options and the assumptions about them are not synchronized with the software and even the documentation may not exist for most of the cases. Understanding and setting up the huge configuration space of third party tools and libraries used by system developers like distributed middleware is practically impossible without clear and structured documentation. To eliminate this problem we have developed a systematic approach for keeping the track of configuration options and the constraints, which confirms to the assumptions the developer made.

In this section we demonstrate how the model driven development methodology is used in the solutions of the challenge 1 and 2 explained in the Chapter I. Our primary focus is on the CIAO component middleware which is used mostly on DRE applications with critical QoS requirements. Firstly, we explain the configuration model of the CIAO middleware. Then we explain the general principles about the modeling of option configurations and finally show how we use OCML to model a middleware configuration.

**A highly customizable middleware: CIAO (Component Integrated ACE Middleware.)** In this section we are focused on the configuration of ACE (Adaptive Communication Environment) middleware. ACE [8] is a highly customizable host infrastructure middleware providing uniform APIs for common operating system functionality. Most of the operating systems provide similar functionality (i.e., inter process communication, concurrent execution and synchronization, signal handling, file I/O, etc.) in various fashions (like, posix, win32, etc.) ACE aims to unify

the access to this functionality in a C++ programming language environment [15]. An important property of ACE is it is used in TAO [7] (The ACE ORB) and CIAO [6] (Component Integrated ACE ORB) distributed middlewares to access the OS functionality in a unified way. ACE is an open source library which is freely distributed together with TAO and CIAO. Although ACE can be used in any application designed with platform independence in the mind, it is specifically designed to be used in applications with high performance and real time communication requirements. One may configure ACE according to the application needs and also may compile to decrease the footprint and latency by omitting the functionality not required by the specific application.

The increase in the portability, effiency and predictability results in a huge configuration space. As an example the compile time configuration of ACE middleware is composed of (1) a C++ header file containing platform specific macros and definitions, (2) a Unix style makefile containing platform specific macros and compile flags, (3) a features file containing various flags representing the system capabilities and available libraries and their respective values stating whether they are available or not, (4) various environment variables pointing the location of the source tree and various libraries required by the compilation process. ACE source code comes with templates for each of these configuration files. The configuration files (1) and (2) are generally consistent for a specific operating system and a compiler combination, however when system requirements are out of the general assumptions made at the time of writing the template files (i.e., using a new compiler or an operating system for which there is no template file, or a different standard library other then the ones which come with the compiler) the application developer should change the configurations manually. The configurations (3) and (4) should most of the time be edited by the application developer because it contains flags for the availability of specific

libraries (i.e., ssl, zlib, various GUI toolkits) and information about those libraries (location on the file system and versions.)

**Structured configuration.** The configuration options for various middleware and applications share a common structure. An option can have a value of basic data type such as an integer and a real number, a string, or a logic value. The configuration options can be grouped into hierarchical structures. In the ACE example, we can group the configuration options as given in the previous paragraph (C++ configuration, unix makefile configuration, features, and environment variables.) In addition to the basic types some configuration options may have a complex data structure like C++ struct types, an example for these kind of options is the Property Value defined by the OMG's Property Service [14] (defined as IDL any type) and Component Property fields of OMG's (`http://www.omg.org`) DnC standard. We can also include the description of these options into the structure, i.e. we can have documentation about the options and their possible values in the data type definition.

W3C (World Wide Web Consortium) (`http://www.w3.org` defines a standard way to orginize structured data. XML Schema [20, 1] is a data type definition language in XML format. While the most of the common basic data types are defined in the XML Schema, there are methods to define complex data types like sequences composed of same types and structures composed of heteregenous data types. XML Schema Definition language specifies a standard way to include structured information about the data types in the documentation with "annotation" and "documentation" elements. In XML Schema definition language someone can define basic constraints about the types via restriction mechanism.

**OCML Workflow.** This section describes the novel dual use feature of OCML. OCML has two phases, which are used by both the DRE system developer and the

middleware developer. Figure II.1 depicts the OCML workflow diagram showcasing the dual use of the OCML tool that can be used both by QoS-enabled middleware developers and DRE system developers. The rest of the section describes the dual use feature of the OCML generative programming tool.

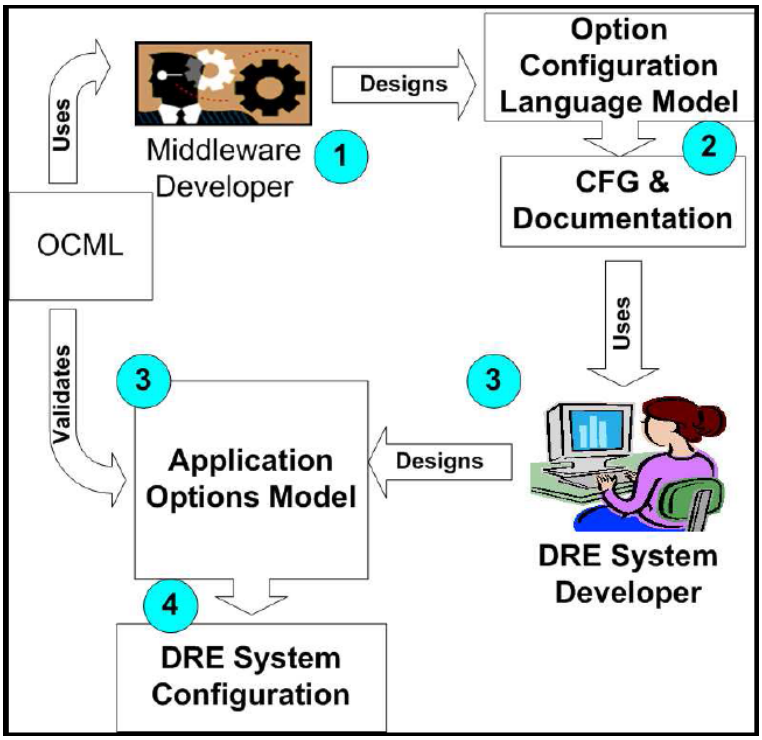


Figure II.1: OCML Workflow

Figure II.1 represents the dual use of the OCML tool. The steps are explained below;

- **Step 1:** Middleware developer uses OCML to model the options, categorize them in a hierarchical order and define the rules governing their dependencies. At this stage the OCML tool is used as a modeling language to create the model specific for a middleware platform.

- **Step 2:** When the middleware specific OCML model is interpreted through the OCML interpreter, HTML documentation and CFG application source code is produced. The middleware configuration layer which consists the steps 1 and

2 in Figure II.1 is hidden in the DRE System Developer, she only uses the generated files.

- **Step 3:** The DRE System Developer uses the generated CFG application to set up the configuration of a specific application. In this step the generated HTML documentation can be used as a reference sheet for the middleware configuration. The design made by the DRE System Developer is checked against the constraints defined in the OCML rule paradigm on-line, which minimizes the risk of choosing wrong set of options.

- **Step 4:** The selected options are exported into files used by the middleware to configure the system.

# CHAPTER III

# ENFORCING CONSTRAINTS AND CONFIGURATOR GUI

Even though a model for the configuration space and the related constraints are defined, the configuration process still has potential for accidental complexities. An application developer or system operator may implement the configuration process wrong and define invalid or not-intended configurations for the application. Providing an environment for configuring applications is useful to resolve this challenge. We have developed an Option Configurator user interface for application developers and system operators to use while configuring their applications and systems. The constraints defined in the configuration model are strictly enforced by this user interface.

The Configurator user interface allows the application developer to enter values for a specific configuration space defined in an XML schema file and corresponding constraint definitions. The values entered by the application developer are kept in an in-memory representation of the schema values. The stored values are then exported into an XML formatted string confirming to the given XML schema file. OCML Configurator is implemented as a library so that an external application can invoke the user interface, feed the interface with some default values and get the updated value, if necessary transform it to a different format via an XSLT [3] process or via code.

The XSD (XML Schema Definition) provides a way to define basic constraints on the data types like specifying minimum and maximum values for numeric types, patterns and length constraints for textual types, however there is no way of defining complex constraints involving more than one type and capturing dependencies on each other. OMG defines an Object Constraint Language [13] (OCL) for complex

constraint definition. OCL is tightly coupled with UML [12] to define the constraints of the objects defined in a UML model. OCL provides detailed control over the constraints on UML Objects and therefore the dependencies between different data types can be handled by OCL however OCL is tightly coupled with the UML and UML is a language specific for *objects* while the XSD is used to define data only (i.e. there are no ways for specifying objects with methods in XSD.)

In OCML we are using a simplified OCL like constraint definition language for handling the inter type dependencies. The defined constraints are executed by a constraint engine. There are two main roles of the constraint engine (1) validates the data given by the user against the predefined constraints (2) if there is a constraint violation tries to validate the constraint by modifying the data. In the execution process the defined constraints are compiled into a first order logic expressions together with the every data entry by the user. The generated expressions are in the format of prolog expressions, and are fed into a prolog interpreter. Finally the changed values in the constraint validation process are queried by the engine and the memory representation of the values are updated accordingly.

As shown in the Figure III.1 there are two actors interacting in a configuration scenario Configurator User Interface and the Constraint Engine. We can demonstrate the interaction process in six steps, which are labeled with numbers in the figure.

1. The XML Schema file representing the structure of the configuration space is parsed by the Configurator. The user interface is created dynamically on the fly according to the XML Schema file. Each element of the data structure is mapped to an appropriate visual data entry element (i.e. numbers to number entry boxes.)

2. The Constraint Engine parses the file containing the constraints and creates the logic structures which are used to validate the data.

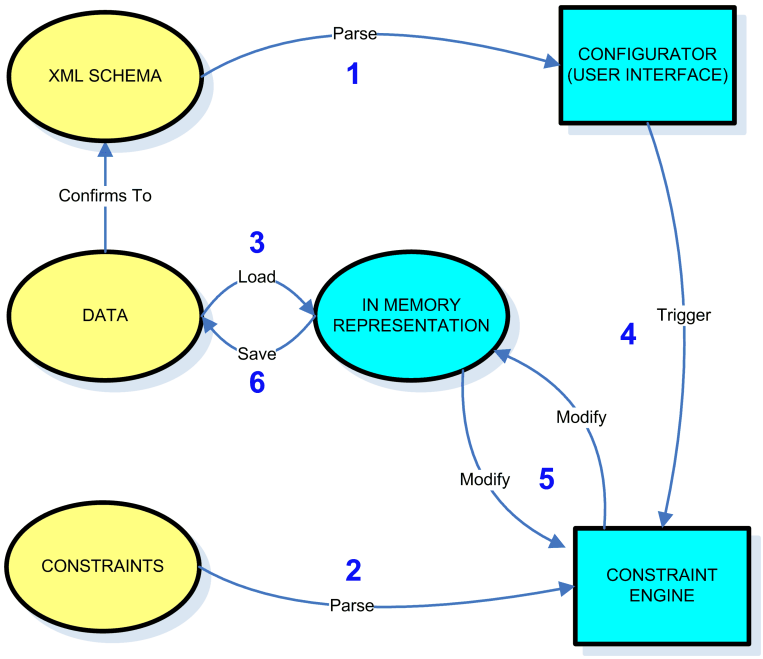


Figure III.1: OCML Internal Interaction

3. The XML data, which confirms to the XML scema, is loaded into the memory. This data contains the previously stored values of a specific configuration.

4. After the parsing and the initiation of the OCML, the user entry is expected. Each action taken by the user, by changing the values in the user interface triggers the Constraint Engine.

5. Whenever triggered the Constraint Engine checks for the newly entered or modified data and checks for any constraint mismatch. If the data entered by the user does not confirm to the constraints, the engine first tries to find a possible solution by changing the effecting values, if finds a solution applies. If there is no such solution the change is rolled back and the user gets a warning stating that the value change request cannot be done by giving detailed information about the constraint.

6. Finally, the changed values are stored back into the XML file.

The OCML constraint engine is integrated with the OCML configurator user interface. The user interaction is filtered through the constraint engine, such as every entry of the user triggers the constraint engine and the engine checks for the given value if it satisfies the direct or indirect dependencies. The in memory representation of the value space is always correct in terms of the given constraints therefore the exported values are correct by construction.

OCML Configurator is a dialog window generated on fly according to the given XML Schema file. The basic XSD types integer, string, and boolean values are represented with basic GUI value entry widgets numeric and text entry fields and check boxes. The enumerated fields are represented with radio boxes. The XSD complex sequence fields are displayed in a tree-like format to reflect the structural relationship between the elements of the same complex type.

# CHAPTER IV

# DEFINING HIGH LEVEL CONFIGURATION MODELS

Configuring the dependent tools and libraries of an application require an extensive knowledge about the way that specific library or tool works and the way it should be configured, however the system operator or an application developer may not have such a detailed knowledge about the dependent systems and configurations. To eliminate this challenge we provide a functionality to define high level configuration options in the configuration space, the higher level configuration maps to the low level configuration options and provide an intuitive way for configuring applications. The constraint definitions are used to define this mapping.

In this chapter we explain the *Configuration Patterns* and how they are related with the *software design patterns*. Then we briefly explain the *Aspect Oriented Domain Specific Modeling Languages (AODSMLs)* and how we have used *Aspect Oriented Programming* methodologies to apply configuration patterns.

**Configuration patterns.** Software design patterns assist developers to document and reuse the solutions for the generic development problems. The software factories [4] concept is highly influenced with the software design patterns which intends to industrialize the software development process and bring an approach like product line architectures.

The different QoS requirements of applications in different domains have common configuration patterns. It is necessary to define recurring configuration patterns for defining a high level set of configuration for a specific application domain. Furthermore the libraries and middleware for a specific domain share a certain set

of terminology. The people using the provided software is expected to master this terminology therefore the configuration parameters of an application should be representable according to this terminology. A high level configuration set defined in a familiar terminology to the application developer and the systems operator, which also represents the system requirements can be accepted as an intuitive way of configuration.

**Aspect Oriented Domain Specific Modeling Languages (AODSMLs)** AODSMLs are modeling languages based on aspect oriented programming methodology. They extend the conventional domain specific models by specifying pointcuts in the modeling level, instead of the programming level which is done by conventional aspect oriented programming languages, like AspectJ [19]. We use aspect oriented programming methodology to define and document the DRE system concerns, (i.e. predictability, interoperability, scalability, etc.)

In a middleware configuration parameter space, in which a DRE system configuration parameters are defined as vectors of specific coordinates, DRE systems with similar QoS requirements can be grouped together. In these kind of systems the configuration parameters are generally shared and they show only small differences (i.e., very few varying configuration parameters.) The groups defined by the relative small distance of the configuration parameters can be mapped onto the DRE system concerns. By defining such a mapping between the configuration parameter space and the DRE system concerns we can provide an intuitive higher level configuration methodology for DRE middleware configuration process. The weaving of the concerns results with configuration files required by a specific middleware (such as XML configuration files for a CORBA Component application.)

# CHAPTER V

# CODE GENERATION

Generating the code and configuration by using the configuration space is a good idea, to eliminate the accidental complexities and minimizing the development time. The code generation add-ons are integrated with the Configurator GUI and automatically generate the configuration settings in the form which the tools and libraries are expecting.

OCML Configurator as explained in the section III is a graphical user interface in which the middleware configuration is done by the application developer. The user may save the current configuration in a file for further use, however one of the main advantages of using the OCML Configurator user interface comes from its expandability. While the Configurator user interface is a generic data entry form, the middleware developers can extend it to generate the configuration data in the form which the specific middleware expects. For example, the CIAO middleware expects a service configuration file which is loaded by the ORB at the initialization time to configure itself. The service configuration file can be in two different forms, one is a text file and the other is a XML file.

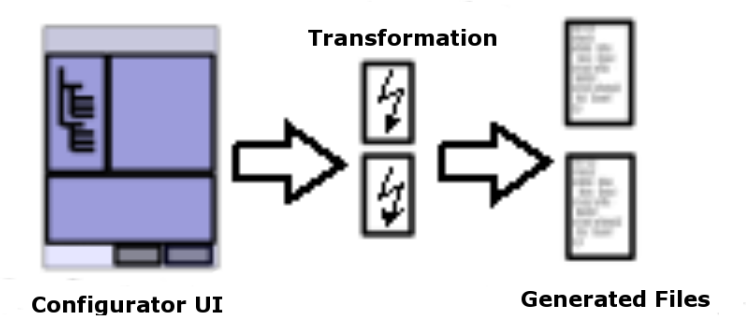


Figure V.1: Extending Configurator with add-ons

As shown in the Figure V.1 there are two ways to extend the OCML Configurator.

(1) Writing an XSLT script which accepts an XML file, which confirms to the XML Schema presented to the OCML configurator (which is used to generate the user interface) and generates a XML or textual output in the form which is expected by the middleware. (2) Writing a C++ code, which traverses the given XML file and generates an output similar to the XSLT script described above.

**Configuration of CIAO component middleware.** CIAO (Component Integrated ACE ORB) is an open source distributed and highly flexible middleware. To provide flexibility for the application developers CIAO provides various methods for configuration which can be grouped in two categories (1) compile time configuration and (2) run-time configuration. The CIAO source code can be freely downloaded from the internet `http://deuce.doc.wustl.edu/Download.html`. Although the precompiled binaries for various target platforms are available, downloading the source code and compiling it has various advantages, like decreasing the foot-print by eliminating the parts not required by the application from the build step, optimizing for a specific host and operating system platform, etc. CIAO is generally used in the embedded systems, therefore footprint and performance are crucial for the application developers.

It is also possible to configure CIAO at run-time, the service configuration framework [16] provided by the ACE host infrastructure middleware (the very basic layer on which the CIAO is built.) The run-time configuration settings are written in a specifically formatted file, the CIAO ORB parses this file at initialization time. Through various hooks in the middleware, the initialization process selects various strategies and sets various parameters according to the given configuration file.

### Demonstrating Weaving Capabilities of OCML in CIAO

In this section we show the generated code from the OCML model interpreters. The code snippet illustrated corresponds the following configuration parameters for the ACE+TAO open-source C++ based CORBA middleware. Table V.1 illustrates the input configuration settings to the OCML paradigm. Below we illustrate the

| Notation | Option Name | Option Settings |
|---|---|---|
| o1 | ORBProfileLock | {null} |
| o2 | ORBClientConnectionHandler | {RW } |
| o3 | ORBTransportMuxStrategy | {EXCLUSIVE } |
| o4 | ORBConnectionCacheLock | {null } |
| o5 | ORBPOALock | {null } |
| o6 | ORBAllowReactivtionofSystemids | {0 } |
| o7 | ORBReactorMaskSignals | {0 } |
| o8 | ORBInputCDRAllocator | {null } |
| o9 | ORBConnectionCacheLock | {null } |

Table V.1: A Sample Configuration Space for ACE+TAO

XML configuration file generated for the input configuration options

```
<?xml version='1.0'?>
<ACE_Svc_Conf>
<static id="Advanced_Resource_Factory"
  params="-ORBReactorMaskSignals 0
          -ORBInputCDRAllocator null
          -ORBReactorType select_st
          -ORBConnectionCacheLock null
"/>
<static id="Server_Strategy_Factory"
  params="-ORBPOALock null
          -ORBAllowReactivationOfSystemids 0
"/>
```

```
<static id="Client_Strategy_Factory"
  params="-ORBTransportMuxStrategy EXCLUSIVE
          -ORBProfileLock null
          -ORBClientConnectionHandler RW
"/>
</ACE_Svc_Conf>
```

The generated configuration files is organized hierarchically into three different factories for configuring the middleware at the server (`Server_Strategy_Factory`) client (`Client_Strategy_Factory`) and client and server (`Advanced_Resource_Factory`). The options themselves are set using the Service Configurator [9] pattern.

# CHAPTER VI

# APPLICATION SCENARIOS TO SHOWCASE C&C PATTERNS

This section describes how our work on C&C patterns can be applied across diverse domains, including the distributed real-time and embedded systems domain and enterprise warehouse management domain. We first provide an overview of the two representative applications in these domains. We then illustrate the configuration challenges of components in these applications . Finally, we show how C&C patterns can be used to address the configuration challenges of components present in these domains.

### Boeing Avionics Scenario

In Boeing's *Basic_SP* scenario [17], a GPS device sends out periodic position updates to a GUI display that displays these updates to a Pilot. The desired data request and the display frequencies are fixed at 40 Hz. The BoldStroke architecture uses a *push event/pull data* publisher/subscriber communication paradigm. The component interaction for the navigation display example is depicted in Figure VI.1. The scenario shown in Figure VI.1 begins with the GPS being invoked by the TAO

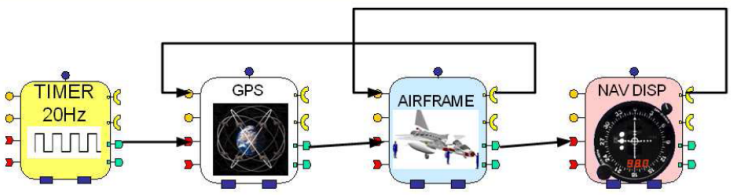


Figure VI.1: Navigation Display Collaboration Example

Real-time Event Service [5] (shown as a `Timer` component). On receiving a pulse event from the `Timer`, the `GPS` generates its data and issues a data available event.

The `Airframe` component retrieves the data from the `GPS` component, updates its state and issues a data available event. Finally, the `Nav_Display` component in turn retrieves the data from the `Airframe` and updates its state and displays it.

**QoS requirements for Avionics Components** Given this scenario, we would like to minimize the latency between the *Airframe* and *Nav_Display* components. To achieve, this goal, it is necessary to examine the Qos attributes/concerns of the two components and map them to corresponding configurations. We assume the application is configured appropriately. We see that the display receives updates only from the Airframe component and does not send messages back to the Airframe component. It thus plays only the role of a client. Similarly, the Airframe component communicates with both the the GPS and the Display playing the role of a peer. However, this component does not concurrently process requests as the events are sequential.

### Robot Assembly Scenario

In this manufacturing assembly line based scenario, a pallet (controlled by the `Palette_Manager` component) containing digital watches moves to a robot station (controlled by the `Robot_Manager` component) where its time is set using the current time provided by a periodic clock (controlled by a *Watch_Manager*. The management for the watch setting facility located at a remote site, can send production work orders and receive response to orders, ongoing work status, inventory, and other messages. These instructions are sent to the WM component using `Management_Work_Instructions` (MWI) component. The WM component interacts with a human operator who using the `Human_Machine_Interface component` (HMI) accepts/rejects the watch. When the watch is accepted, the WM component uses the RobotManager to set the time.

However, when a watch is rejected, the RobotManager removes the watch from the assembly line. Figure VI.2 illustrates this assembly of components.

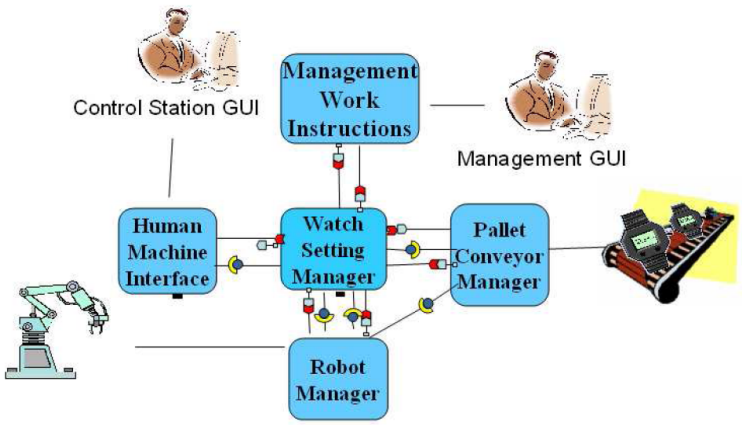


Figure VI.2: Robot Assembly scenario

**QoS requirements for Robot Assembly components** As with the avionics scenario, in this case, the round-trip latency between the Watch Manager and the Human Operator needs to be minimized to improve the efficiency of the product line. To achieve this goal, the application developers need to configure the underlying middleware and the individual components to minimize latency. Similar to the Display component, the HMI component only plays the role of a client as its only source is the Watch Manager component. The Watch Manager however, is at the heart of the scenario interacting with all other entities. However, it (like the Airframe component) does not process requests concurrently, because the decision to accept/reject a watch is made by the Human Operator sequentially. Hence its characteristics are similar to that of the Airframe component.

## Hypotheses

In this study, we explore the following high level hypotheses:

- The modeling tools enable capturing the QoS requirement, *i.e.*, end-to-end latency and the configuration space for the scenarios.
- The `Nav_Display` and `Human_Machine_Interface` components having similar QoS requirements/concerns, should have the underlying middleware configured the same way to satisfy the QoS requirements. The same for the `Airframe` and `Watch_Manager` components.

## CCV Patterns Identification Process

We applied the following step-wise process to document CCV patterns.

**Choosing Subject Application** We used ACE v5.4.2 + TAO v1.4.2 + CIAO v0.4.2 for this study. CIAO is a QoS-enabled implementation of CCM being developed at Washington University, St. Louis and Vanderbilt University to extend TAO [11] to support components, which simplifies the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system.

**Modeling Component Interactions** Use CoSMIC modeling environment to build the application scenario. To conduct the QA task, this phase involved modeling the artifacts, *i.e.*, elements involved in the scenario. Figure VI.3 illustrates how the Robot Assembly component interactions were modeled in CoSMIC. The models were then checked for constraint violations and XML meta-data needed for deployment was generated from the models.

**Determining configuration space** Select a set of middleware C&C options using the OCML tool that are expected to provide these QoS guarantees. The CIAO middleware provide over 500 configuration options, however not all of these correspond

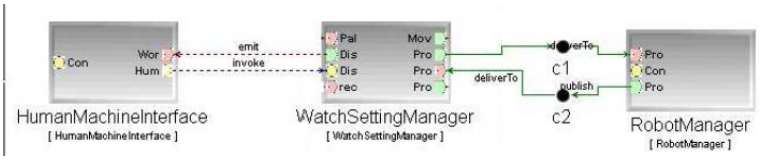


Figure VI.3: Depicting component interaction

to the QoS requirements for the components in our study. For example, the Navigation Display and HMI components do not need to consider server side options as they only act as clients. Thus the first step involved narrowing down the configuration space via examination. Table VI.1 shows the relevant configuration options for the aforementioned components. Further examination of this reduced configuration

| Notation | Option Name | Option Settings |
|---|---|---|
| A | ORBReactorMaskSignals | {**0**, 1} |
| B | ORBInputCDRAllocator | { **null**, thread} |
| C | ORBReactorType | { **select_st**, mt} |
| D | ORBProfileLock | {thread, **null**} |
| E | ORBObjectLock | {thread, **null**} |
| F | ORBConnectionCacheLock | {**null**, thread} |
| G | ORBClientConnectionHandler | {RW, ST} |
| H | ORBTransportMuxStrategy | {EXCLUSIVE, MUXED} |
| I | ORBConnectionPurging Strategy | {LF, reactive} |
| J | ORBConnectStrategy | {LF, reactive} |

Table VI.1: Configuration Space for Display based Components

space, reveals that some of the configurations settings can be a priori *i.e.* without experimentation. For example, both components interact with only one source and do not need synchronization or locking. These option settings (options o1 - o5) can be directly determined (shown in bold) as shown in Table VI.1. For the remaining configurations, where both options are suitable, the suitable configurations are determined experimentally. For the `Watch_Manager` and `Airframe` components the configuration space was determined in a similar manner. Table VI.2 illustrates the configuration space. Note that apart from the (K-L) options shown in the table, options (A-E)

| Notation | Option Name | Option Settings |
|---|---|---|
| K | ORBReactivationOfSystemIds | {0, 1} |
| L | ORBPOALockType | {thread, null } |

Table VI.2: Configuration Space for WatchManager and Airframe Components

shown in Table VI.1 are also relevant to these components. After determining the relevant configurations, we use our OCML tool to generate the configuration files for all four components.

**Navigating Configuration Space** For the configuration space chosen, run the generated benchmarking tests to evaluate QoS. The Skoll framework is leveraged to schedule the individual experiments and navigate the configuration space. Figure VI.4 shows how Skoll builds a configuration model to schedule experiments on host machines. For each such constrained C&C set in step 4, run the generated benchmarking tests to evaluate the QoS.

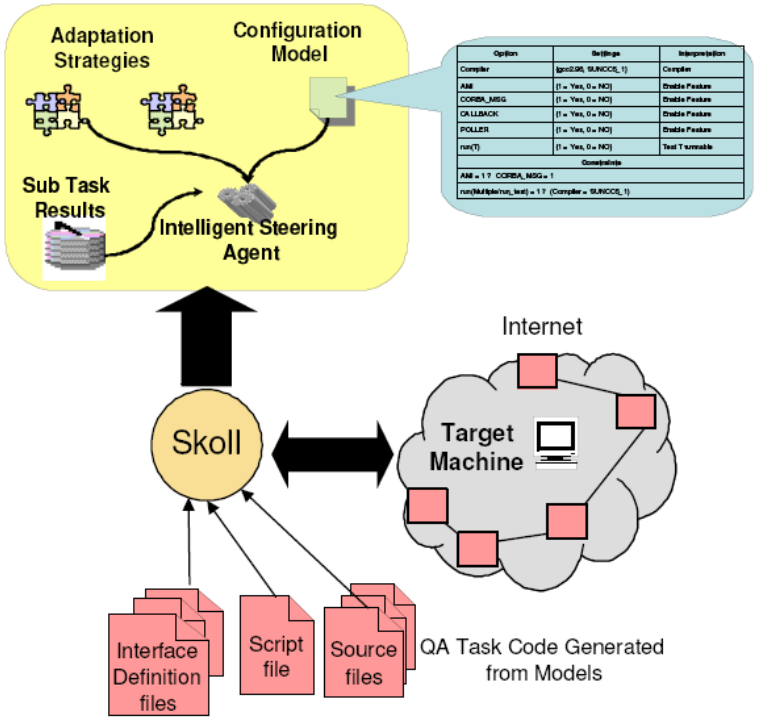


Figure VI.4: Integrating MDD tools with Skoll

**Identify Reusable CCV artifacts** Repeat the above process for both the scenarios and analyze results from the configuration space to identify clusters of configurations that influence QoS the most. If these patterns are also shown on other platforms/domains then the configuration can be factored out as a reusable artifact.

**Use representative Deployment Scenario** To empirical evaluate the configuration and identify the recurring settings, we used the testbed as shown in Table VI.3. The individual computers themselves then were connected via Local Area Network (LAN). This configuration simulates a deployment scenario where these components will be deployed on different nodes or embedded within processor boards.

| | **DOC** | **ACE** | **Tango** |
|---|---|---|---|
| CPU Type | AMD Athlon | AMD Athlon | Intel Xeon |
| CPU Speed (GHz) | 2 | 2 | 1.9 |
| Memory (GB) | 1 | 1 | 2 |
| Cache (KB) | 512 | 512 | 2048 |
| Compiler (gcc) | 3.2.2 | 3.3 | 3.3.2 |
| OS (Linux) | Red Hat 9 | Red Hat 8 | Debian |
| Kernel | 2.4.20 | 2.4.20 | 2.4.23 |
| Avionics | Trigger<br>Nav_Display | Airframe | GPS |
| RobotAssmbly | Watch_Manager | Palette_Manager<br>Robot_Manager<br>MWI | HMI |

Table VI.3: Testbed and Deployment Summary

## Result Analysis

In this section we analyze the results obtained from our experimentation process and showcase reusable CCV artifacts that apply to the components in both domains. The experimental process described earlier showed how our tools help in capturing

HMI Component

| Setting | Latency ($\mu$secs) |
|---|---|
| (G1, H1, I2, J2) | 64.70 |
| (G1, H1, I1, J2) | 65.10 |
| (G1, H1, I2, J1) | 65.40 |
| (G1, H1, I1, J1) | 65.60 |
| (G1, H2, I2, J2) | 65.80 |
| (G1, H2, I1, J1) | 66.50 |
| (G1, H2, I1, J2) | 68.11 |
| (G1, H2, I2, J1) | 68.19 |
| (G2, H1, I1, J2) | 68.30 |
| (.....) | |

Scenario 1

Nav_Display Component

| Setting | Latency ($\mu$secs) |
|---|---|
| (G1, H1, I2, J2) | 504 |
| (G1, H1, I2, J1) | 528 |
| (G1, H1, I1, J2) | 529 |
| (G1, H1, I1, J1) | 532 |
| (G1, H2, I1, J1) | 536 |
| (G1, H2, I2, J1) | 548 |
| (G1, H2, I1, J2) | 552 |
| (G1, H2, I2, J1) | 562 |
| (G2, H1, I2, J2) | 568 |
| (.....) | |

Scenario 2

Table VI.4: Latency QoS distribution for the HMI, Nav_Display Components

Airframe Component

| Setting | Latency ($\mu$secs) |
|---|---|
| (K0, L0) | 452 |
| (K1, L0) | 459 |
| (K0, L1) | 462 |
| (K1, L1) | 467 |

Scenario 3

Watch Manager Component

| Setting | Latency ($\mu$secs) |
|---|---|
| (K0, L0) | 55.5 |
| (K1, L0) | 56.8 |
| (K0, L1) | 59.6 |
| (K1, L1) | 60.2 |

Scenario 4

Table VI.5: Latency QoS distribution for Airframe Watch_Manager Components

the QoS and configuration concerns present in both the Avionics and RobotAssembly domains thus addressing first hypothesis. Table VI.4 tabulates the latency distributions for the client-side display based components. We use the notation *A1*, *B2*, etc. to identify the individual options within each category. For example, the `ORBConnectStrategy` value of `LF` is denoted as *J1*. The top 8 configurations (out of a possible 16) are shown in increasing order of latency values. A closer look at the values reveals a clear pattern of configuration options and its effect on QoS (end-to-end) latency. For example, the options G1 has the greatest effect on performance, *i.e.*, changing its value to G2 increases latency by $\sim 4\mu$secs for the RobotAssembly and by $\sim 50\mu$secs in the second case. After G, the option H influences latency the most, *i.e.*, changing its value from H1 to H2 worsens latency by $\sim 2\mu$secs in the first case and by $\sim 30\mu$secs in the second case. Table VI.5 shows the results for the Airframe and Watch Manager components. We see that the settings for L have the greatest influence on latency, *i.e.*, changing its value along increases latency by $\sim 8\mu$secs in the Avionics case and by $\sim 4\mu$secs in the RobotAssembly case.

The categorization of the latency values, enables us to use clustering analysis to create distinct sets of tuple spaces, *i.e.*, $(x_i, val(x_i)...x_n, val(x_n)$, where $x_i$ denotes a configuration and $val(x_i)$ its settings. All the elements in the set being closely related. The sets themselves can be visualized as being separated by hamming distances. Where moving from a configuration in one set to another results in an improvement/degradation in the QoS measures. As shown in the Table VI.4, the top 4 configurations for both the components are put in a set and the next four configurations form another cluster. Analyzing the the cluster also reveals a pattern, we see that the topmost configuration produces the best end-to-end latency values for both cases. This configuration identified and codified empirically can be reused (along with

the configurations chosen by examination) to directly generate the most appropriate middleware configuration.

The codification of the configurations, also enables them to be reused as a *CCV pattern*. A design pattern presents a solution to a common software problem within a particular context [2]. A CCV pattern is similar to a design pattern in that it represents a recurring solution to a *configuration and customization* problem arising within a particular context, *e.g.*, for similar concerns across multiple domains. Our results showed that (1) same configuration satisfies the QoS requirement for two components having similar operational context across multiple domains, and (2) The configurations were the same independent of the underlying platform (hardware, OS and compiler)[1]. CCV patterns help in modularizing the cross-cutting configuration concerns aiding their reuse across several application domains, thereby minimizing unnecessary effort expended in rediscovering these patterns for each application domain. These configurations can also be fed-back into the modeling tools to directly generate the most suitable configuration given the Qos requirement/concern. For example, the OCML tool can be used to generate the most appropriate configuration given set of requirements.

[1]The Nav_Display and HMI components were placed on DOC Tango Machines in our experiment, each an entirely different platform as shown in Table VI.3.

# CHAPTER VII

# RELATED WORK

Techniques for middleware configuration. A number of ORBs (such as VisiBroker, ORBacus, omniORB, and CIAO) provide mechanisms for configuring and customizing the middleware. For example, CIAO uses the ACE Service Configurator framework, which can be used to statically and dynamically configure components into middleware and applications. Likewise, Java provides an API for configuring applications based on XML property files. Key/value pairs for specific options are stored in an XML-formatted files and read by applications using XML parsers or a provided API.

The Microsoft .Net platform provides a similar approach to the Java XML property files named .Net configuration files. The System.Configuration API can be used to read the configuration. Using this API, .Net provides access to three different information: (1) machine configuration files, which control machine-wide assembly binding and remoting channels, (2) application configuration files, which control application-specific configurations, such as assembly binding policies and remoting objects, and (3) security configuration files, which contain security information for applications. Editing text configuration files formatted with XML is common for .Net, Java, and various CORBA and CCM implementations. OCML enhances the configuration of various middleware platforms by providing an MDM methodology. From these OCML models, documentation about the configuration of the middleware and a GUI interface for middleware configuration can be generated automatically.

# CHAPTER VIII

# CONCLUDING REMARKS

The OCML MDD tool provides a metaprogrammable interface to capture the configuration concerns of QoS-enabled middleware. The OCML metamodel provides extensible, platform-independent building blocks for representing configuration options of diverse middleware implementations. The generative capabilities of OCML enable both syntactically and semantically correct configurations. OCML also generates documentation for the option space for QoS-enabled middleware, similar to Javadoc [18] and Doxygen via annotations within the models. OCML can be applied to both compile- and run-time options, whereas other approaches (such as Java's code-level meta-data annotations [10]) are only applicable to compile-time options. OCML also enables *separation of concerns*, where middleware developers model the middleware options and their constraints and application developers use the Configurator GUI produced from the OCML models to generate configurations suitable for their domain. Application of Configuration and Customization Patterns is a novel approach to middleware configuration domain. The main advantage of this approach is to bring easy of use to the application developers.

OCML provides a framework for modeling the option configurations, in this work we have mainly focused on the configuration model of CIAO middleware. For every other tool or middleware the middleware developers need to design a model specific for thaat middleware configuration and develop XSLT scripts or code to make appropriate transformation from the OCML generated XML format to the configuration format expected by the application.

OCML is a tool to make easy the configuration process, it does not guarantee end-to-end QoS. However, using OCML with the benchmarking tools is the recommended way for ensuring QoS requirements.

# BIBLIOGRAPHY


[1] Biron, P. V., and et al., A. M. XML Schema Part 2: Datatypes. W3C Recommendation, 2001.

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[3] Clark, J. XSL Transformations (XSLT). W3C Recommendation, 1999.

[4] Greenfield, J., Short, K., Cook, S., and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[5] Harrison, T. H., Levine, D. L., and Schmidt, D. C. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97* (Atlanta, GA, Oct. 1997), ACM, pp. 184–199.

[6] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[7] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[8] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/, Vanderbilt University.

[9] Jain, P., and Schmidt, D. C. Service Configurator: A Pattern for Dynamic Configuration of Services. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems* (June 1997), USENIX.

[10] Java Specification Request (JSR) 175. A Meta Data Facility for the Java Programming Language. http://www.jcp.org/en/jsr/detail?id=175, Java Community Process.

[11] Krishna, A. S., Schmidt, D. C., Klefstad, R., and Corsaro, A. Real-time CORBA Middleware. In *Middleware for Communications*, Q. Mahmoud, Ed. Wiley and Sons, New York, 2003.

[12] Object Management Group. *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 ed., Sept. 2001.

[13] Object Management Group. *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*, OMG Document ptc/03-10-14 ed., Oct. 2003.

[14] OMG. *Property Service Specification*, 1.0 ed. Object Management Group, July 1996.

[15] Schmidt, D. C. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* $6^{th}$ *USENIX C++ Technical Conference* (Cambridge, Massachusetts, Apr. 1994), USENIX Association.

[16] Schmidt, D. C., and Suda, T. The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons. In *Proceedings of the Second International Workshop on Configurable Distributed Systems* (Pittsburgh, PA, Mar. 1994), IEEE, pp. 190–201.

[17] Sharp, D. C., and Roll, W. C. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003* (May 2003).

[18] Sun Micro Systems. The Javadoc tool. http://java.sun.com/j2se/javadoc/, Sun Developer Network Community.

[19] The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.

[20] Thompson, H. S., Beech, D., Maloney, M., and et al., N. M. XML Schema Part 1: Structures. W3C Recommendation, 2001.