Run Manager Module for CORAL Laboratory Management

by

Jeffrey G. Klann

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author .....
Department of Electrical Engineering and Computer Science
August 8, 2002

Certified by .....
Vicky Diadiuk
Principal Research Engineer, MTL
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Run Manager Module for CORAL Laboratory Management

by

Jeffrey G. Klann

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a new module, the Run Manager (RM), for Stanford Nanofabrication Facility's Common Object Representation for Advanced Laboratories (CORAL). CORAL is the lab manager with which MIT's Microsystems Technology Laboratories hopes to replace its outmoded, fifteen-year-old lab manager. RM will be used to collect and store parameterized information about each step in the device fabrication process, which labs will use to automatically generate accounting, maintenance, and lot history reports. RM consists of an XML (Extensible Markup Language)-configurable Common Object Request Broker Architecture (CORBA) application server, database storage, and a Graphical User Interface (GUI) module which will be integrable with the existing CORAL Java client.

Thesis Supervisor: Vicky Diadiuk
Title: Principal Research Engineer, MTL

Acknowledgments

Thank you to Vicky Diadiuk, Thomas Lohman, and Ike Lin at MTL, whose design and requirements-gathering assistance was invaluable in ensuring the success of this project. Also, many thanks to John Shott and Bill Murray at SNF for their additional insights and suggestions.

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 MTL and CAFE

The MIT Microsystems Technology Laboratories (MTL) is an Interdepartmental Laboratory established in 1984 to support education, teaching, and research in areas that could involve microelectronic and microfabrication technology. MTL houses more than a hundred pieces of equipment used by nearly 40 research groups. Over 350 students use the lab each year, and MTL facilities provide critical support for the research done by students, resulting in 30 PhD and 30 SM and MEng degrees awarded in the 1998-1999 academic year [7, 4]. The plethora of lab resources is managed by a technical staff that includes Dr. Vicky Diadiuk, nine research engineers, and three research technicians. This Operations Group is in charge of maintaining the machines and their processes and training students to use them. Scheduling machine reservations and tracking machine maintenance is facilitated by a computer lab manager. MTL currently uses a 15 year-old home-grown lab manager called CAFE (Computer Aided Fabrication Environment) that allows users to make machine reservations and report maintenance problems. Also, a sophisticated billing subsystem is coupled to CAFE and is designed to charge users only for the work they have done and resources consumed, and not simply time at the machine. CAFE is a first generation lab manager, and while it served as an excellent proof-of-concept for a sophisticated computer lab manager, it is lacking in features and functionality. For one, its user

interface is a teletype terminal, which many users find cumbersome and confusing. More importantly, CAFE is technically outmoded. It lacks a distributed, three-tiered architecture and is thus not as scalable or expandable as a second generation lab manager could be. It cannot interlock the machines or validate user identity, and therefore it relies almost entirely on user honesty for billing, usage, and reservation tracking. Finally, CAFE has little functionality to collect data on how the machines are used, though these data are important for billing and scheduling maintenance. Instead, CAFE users are asked to voluntarily add syntactically cryptic comments to a comment field when they operate a machine, describing how they used the machine. This comment-adding process is suboptimal for collecting accurate, complete data and creating a pleasant user experience. It also requires many manual interventions for accounting purposes [4, 9].

Another major failure of CAFE is that it is not flexible enough to allow for changes in processes often called for in research. For example, a user might add wafers (see definition in Appendix A) to their lot as their research progresses, but CAFE doesn't support this change. PFR, or Process Flow Representation, is an extension of CAFE that tries to obviate the need for in-process changes by asking users to design their entire set of process steps ahead of time. However, this pre-planned approach is infeasible for unique runs, which is normally the case in research. Thus, CAFE's inflexibility is actually worsened by PFR, which was therefore abandoned in 1996 [4, 9].

1.2 Collaboration with other schools

CAFE is the result of joint work among a group of fabrication research laboratories who have collaborated in developing computer automation tools since the mid 1980s. The core of CAFE was original developed at the University of California Berkeley (UCB), and it evolved into separate proprietary packages at each school which adopted it: CAFE at MTL and Crystal at the Stanford Nanofabrication Facility at Stanford University (SNF) [4, 9]. SNF, MIT, and UCB are equal in sophistication.

SNF, for example, has over 400 researchers who logged about 50,000 hours last year [18]. It is not surprising, then, that SNF and UCB have also been feeling burdened by their cumbersome CAFE-like lab managers in recent years. Therefore, SNF has launched development of a second-generation lab manager, which is already in use (though not nearly feature-complete) at SNF. UCB has also begun developing a new lab manager, but it will not be complete until 2005, which is not within MTL's time-frame for a lab manager upgrade [4]. Therefore MIT has chosen to collaborate with SNF again to contribute software enhancements and implement CORAL at MIT by the end of 2002. This new lab manager is SNF's Common Object Representation for Advanced Laboratories (CORAL), currently being developed by Bill Murray and John Shott at Stanford and Thomas Lohman and Ike Lin at MIT. The acronym CORAL was selected in part because "while [both the system and ocean coral] appear to be stable and lifeless, each is, in fact, a living and growing organism [16]." CORAL is constantly under development, which is intended both to better serve the SNF community and to make CORAL the lab manager of choice for other nanofabrication research facilities. As a second-generation lab manager, CORAL is designed to be flexible, extensible, and scalable. It is based on a three-tiered client/server/database architecture that uses CORBA application server technology (see definition in Section A) and a powerful client-side Java GUI. It supports interlocking machines and verifying users [16, 4, 9].

Currently, CORAL has four implemented functional abilities: 1) tracking equipment usage, 2) managing equipment reservations, 3) allowing communication of equipment problems for scheduling maintenance, and 4) user management [16]. While these four functions provide the basis of lab management at MIT, CORAL still lacks many of the features MTL requires and desires. CORAL cannot produce statistical reports of machine logs and repair history. CORAL lacks a key feature in CAFE which allowed users to produce a report of which machines operated on their wafers (called a traveller'), which is useful when trying to recall how a correct wafer was produced. These travellers are essentially a lab notebook, which reconstruct a process flow (as in PFR) after the fact, rather than before [4, 9]. Most importantly, CORAL lacks

the ability to collect information on machine use and then bill users based on the tasks performed using the equipment. CORAL's tracking system is based only on time spent at a machine, which is even weaker than CAFE's already cryptic and trust-dependent system. Because users can use a machine for two hours in very different ways, MTL considers CORAL's tracking system insufficient, and it would be a step backward in technology to use the comment-field tracking hack currently used in CAFE [4].

Although CORAL already supersedes CAFE in its flexible design, scalability, user interface, and security features, MTL's needs require further development and design before CORAL can replace CAFE at MTL. As a joint project, everything that is developed at MTL and SNF will be kept synchronized at Stanford and will be useful to Stanford and other universities as CORAL becomes the research lab manager of the new millennium.

1.3 Previous work

CAFE and Crystal were first-generation fabrication lab managers developed jointly in the 1980s between SNF, MTL, and UCB. These products have been in use at these three labs for over 10 years, and other labs (such as MIT's Lincoln Lab) have picked them up more recently. However, these products have become technically outmoded for reasons just described and need to be replaced by a second-generation lab manager. Commercial lab managers, such as PRI Automation's Promis, have been in development for many years, but their focus is on production fabrication, not research fabrication. Consequently, they are designed for one process to be run thousands of times, rather than for thousands of processes to be run one time, as research fabs require. Promis has no concept of reserving machines, because commercial labs staff each machine with a factory employee. Therefore Promis and other commercial lab managers have inflexible (or no) accounting, reservation, and process design and are inappropriate for research purposes [14, 4, 9]. The lack of commercial lab managers useful to research labs was a major motivation in developing CORAL.

Chapter 2

System Requirements

2.1 Motivation summary

MTL has a long wishlist of items it would eventually like added to CORAL, but the one key piece is the ability to collect customized information on tool usage in the lab. MTL uses this information not only to bill users but also to schedule machine maintenance and provide lot history reports to researchers. Because SNF bills its users only by time spent, they have no motivation to design a machine-usage-collector module.

Therefore MTL needs the CORAL Run Manager module (RM), the design of which is the topic of this thesis. RM must minimally provide for CORAL the same data-collection capability as CAFE, but also it should supersede CAFE in that: it must guarantee data-completeness and must be user-friendly (by not requiring users to enter data into a preexisting comment field which is never parsed for errors), and it must be flexible (i.e. easy for a lab administrator to update RM to reflect new tools and new tool capabilities).

The following detailed goals were discussed over a period of weeks at the start of this project, and the system design was chosen with its primary focus on these goals. By meeting these goals, this thesis project contributes a new CORAL module with greater flexibility and extensibility than previous modules, by meshing new technologies (which will be discussed in detail in the next chapter) with the existing CORAL

framework. This contribution demonstrates that careful research and design mixed with CORAL's already flexible structure can significantly extend the capabilities of the lab manager. This thesis also demonstrates the power of thoughtful design, modern project management, and extensive documentation to create quality tools quickly and with positive user reaction.

2.2 Goals

RM has three major goals, each with subgoals. It must be *flexible* (i.e. easy for an administrator to configure support for new lab capabilities), it must be *maintainable* (i.e. easy for a programmer to understand, update, and add capability long after the initial implementation is complete), and it must be *easily integrable* (i.e. easy for a programmer to incorporate the module into the CORAL system in a matter of weeks). Refer to the terminology definitions in Appendix A when studying these goals.

2.2.1 Flexible

- *Customized forms for any tool:* RM must collect run-data from each tool in the lab, providing a unique input form customized for each tool's process steps. Tools can be any object that performs a process, from machines to abstract ideas such as training courses.

- *Separation of process step from tool:* The concept of process step should be decoupled from the tool used to perform those steps. This prevents general process ideas from being unnecessarily tied to specific tools.

- *Inheritance in process steps:* The specification of process steps should be extensible and flexible. Specifically, process step definitions should support derivation from more basic step definitions (i.e. inheritance). This will allow partial-definition-reuse; all machines, for example, have some fields in common.

- *Variety of field types:* Process steps should support a variety of configurable field types. Minimally, text fields, integer numbers, floating-point numbers, and multiple-choice selections must be supported. All fields must have the ability to be optional and provide a default value.

- *Multiple copies of a field:* Every supported field must be configurable such that it can be repeated, with different values, a minimum and maximum number of times. (i.e. a configuration could specify that some process step supports two to five materials).

- *Easy configuration updates:* It must be possible to update RM's configuration on-the-fly, without shutting down all the clients or modifying code on the server, using a provided administration tool. This administration tool should also be extensible to support future configuration-maintenance tasks.

2.2.2 Maintainable

- *Easily enhanced:* RM must be designed with future enhancements in mind, making design decisions to modularize code and easily support augmentation wherever feasible.

- *No redundant changes when enhancing code:* It should be possible to enhance the code (i.e. make significant but non-structural changes, such as adding new field types to input forms) without making redundant changes (i.e. changing a data structure's definition in three different places).

- *Easy to write accounting/reporting scripts that use RM-collected data:* Collected data should be validated and stored in a database. The database should provide enough information on the data's meaning (i.e. datatypes, etc.) that it can be reconstructed by a future accounting or reporting script (such as a traveller or machine-maintenance report).

- *Documentation:* Documentation often is a forgotten piece of software design and implementation [9]. Thorough technical documentation is essential for easy

maintenance. Therefore, accurate JavaDoc (API documentation in-line with the code), installation documentation, and a set of technical overview slides are required.

- *Project management:* The system and its design must be maintainable from the start, in that it must be clear to the MTL team what is occurring at all times. A constant review of requirements and detailed progress reports must always be available, so that the system meets the goals it was intended to meet. Timely code releases should also be available for inspection.

2.2.3 Easily Integrable

- *Follow the CORAL Paradigm:* RM must integrate into the existing CORAL architecture. The constraints of the CORAL architecture are discussed in Section 3.1.

- *Easy client integration:* Additionally, the RM client must be nearly trivial for a programmer to plug into the CORAL client in a few days. Specifically: the package structure should mesh with that of CORAL (so the build can be easily integrated), and the additions to the client should have a simple API that integrates easily with CORAL.

- *Committed/uncommitted record model:* RM's run-data must mesh with CORAL's enabled/disabled tool paradigm. Specifically, while a tool is enabled in CORAL, its usage parameters could be changed by the lab user. Therefore, its run-data in RM must be editable and not yet available to accounting records (as it is not yet complete). This is an uncommitted state. Once a tool is disabled, a lab user cannot affect it, so its run-data must be fixed permanently and available to accounting. This is a committed state.

- *View locks:* Because CORAL may be used by a single user on many client terminals at once, and each client terminal can access all machines in the lab,

it is important that editing RM on one terminal not create corruption on another terminal. This could occur, for example, if a user displays RM data on two terminals, and then edits the data and saves it on only one; in the most straightforward design, the other terminal will not reflect the changes. To prevent this, a design goal was added to only allow a user to edit their run-data for a single tool on one terminal at a time.

- *Approval of SNF:* The software team at SNF must be satisfied with the system, so that they will include RM in their main CORAL CVS tree and it will be distributed with future releases. Beyond meeting the other goals in this section, good communication with SNF and a demo specifically for their team is anticipated.

The following chapter will discuss the detailed design of the Run Manager, touching on how each of the above design goals was met.

Chapter 3

Design

3.1 CORAL Architecture

Before delving into the details of the Run Manager's design, it is important to understand the CORAL technology into which RM will integrate. CORAL is a three-tiered (client/server/database) architecture. The client is a Java Graphical User Interface (GUI) application which provides users access to the various elements of lab management (such as making machine reservations and enabling machines). The client connects via CORBA to a number of Java application servers which each provide a piece of CORAL functionality (i.e. tracking machine reservations, tracking equipment status). This modular server design allows CORAL to be somewhat easily extended, because the server applications are essentially independent. The servers store data in a relational database; currently both Postgres and Oracle are supported.

To integrate with CORAL, the Run Manager must provide client code that hooks into CORAL's existing Java GUI, and it must provide a complementary CORBA server application that supports both Postgres and Oracle.

3.2 Overview

As with CORAL, the Run Manager is based on a three-tiered (client/sever/database) architecture. Additionally, RM uses the Extensible Markup Language (XML) to allow

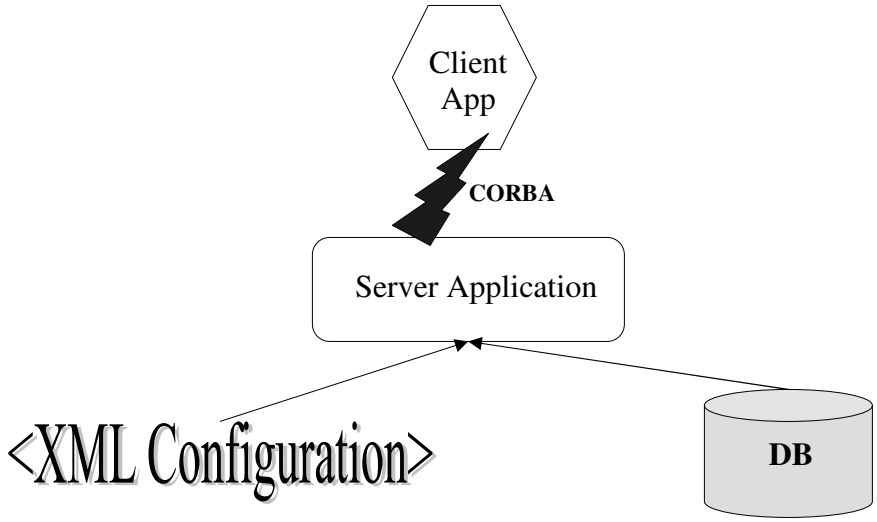

Figure 3-1: High-level overview of the Run Manager.

administrators to define processes, tools, and the connections between them. The data flow of the Run Manager is as follows:

1. An administrator defines the **lab configuration** in XML, and he notifies the Run Manager server of this via a Run Manager administration tool (i.e. the Admin Tool in Section 3.6).

2. The **Server** loads the lab configuration into memory.

3. On request from the Run Manager client (Client), the Server sends a relevant portion of the configuration to the client via **a remote method call**.

4. The **Client** generates a Graphical User Interface (GUI) to collect data from the user.

5. The Client translates user-entered data to a database-compatible structure and sends it back to the Server, again via **a remote method call**.

6. The Server stores the data in the **database** with a level of completeness such that it can be reconstructed independently of the configuration by future accounting scripts.

The high-level interactions between the Client, Server, database, and XML configuration can be seen in the overview in Figure 3-1. This architecture provides an easy integration path with CORAL. The Server can be reused as a CORAL server (as discussed in Section 3.4.4), and the Client is built on a reusable portion of code which can be plugged in to the existing CORAL client application (as discussed in Section 3.5). This CORAL-integrable design is essential to meeting the *easy integration* goals discussed in Section 2.2.3.

The rest of this chapter will discuss the components of RM's data-flow in detail: *XML configuration/database design, Server design, Client design, Admin Tool design,* and *project management*. Throughout this chapter, refer to the XML Schema (in Appendix E) for data-structure definitions, the included figures for Java class layout, and the definitions (in Appendix A) for some terminology explanations.

3.3 Configuration and Database Design

The foundation of RM is its configuration-definition language and its run-data storage paradigm. Most of the *flexibility* goals in Section 2.2.1 are possible because of an extensible XML language used for defining lab tools, process steps, and connections between them. This language is supplemented by a database storage format that is not tied to any specific field names or even field types, so that modifying the configuration (and often, adding to its capabilities) will not change the database structure. Finally, the XML configuration and the database tables are mapped to Java with a powerful object mapping tool called Castor. This tool provides translation between Java objects, XML tags, and SQL (defined in Appendix A) tables. Castor will be discussed in more detail in Section 3.3.3.

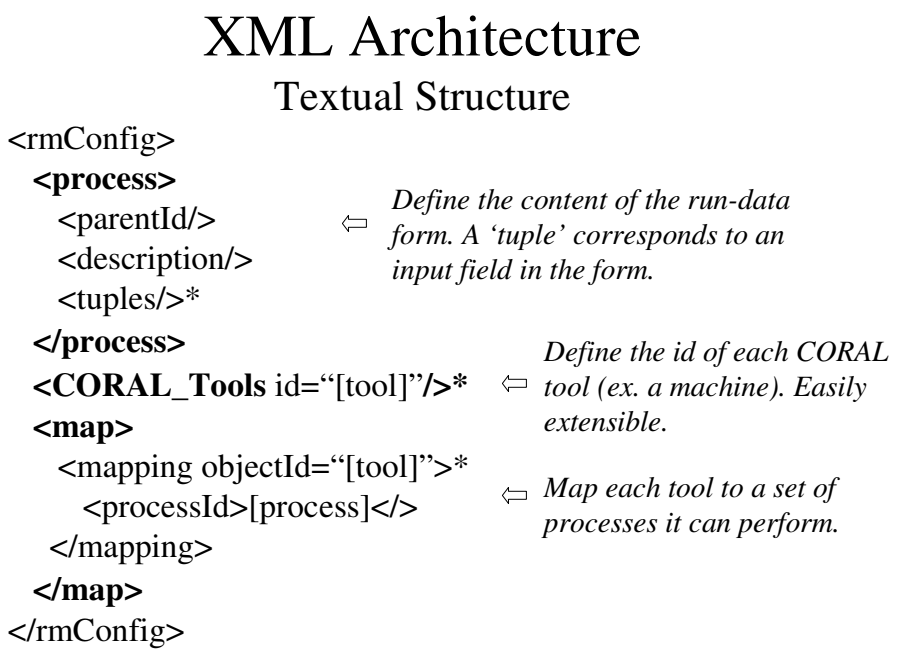

Figure 3-2: XML Architecture: Configuration file overview.

3.3.1 XML Configuration Language

RM has a *configuration* which defines what data it collects. This configuration is written in XML and consists of a text file stored in the database, which can be updated by the Admin Tool (see Section 3.6). The configuration's language is XML, constrained by an XML Schema (listed in Appendix E) which defines an `rm` namespace. XML with XML Schema was chosen on recommendation from Stanford [15, 11], because XML with XML Schema allows quick definition of human-readable, customized languages and is quickly becoming the standard for configuration languages [20, 15, 11]. The Schema defines three sections: *a process library, tool references,* and *tool mappings.*

- The *process library* is an object hierarchy of process step definitions not tied to a specific tool. Each object in the hierarchy may derive from multiple other objects, and each object defines the unique (non-derived) part of the input form for its process step, using the input-field tags provided by the Schema.[1] When a

[1]The current implementation supports input-fields of strings, integers, multiple-choice pulldowns,

process step is displayed by the client, the content of all its ancestor objects are also displayed. This library structure achieves the goal of supporting inheritance in process step-definitions.

- *Tool references* are simple objects which begin with a tag describing the tool's type (i.e. machines are described by the `<machine>` tag). They usually contain the id of a tool (for linking the tool back to CORAL) and a tool name (for display in parts of the RM client). Each tool type can extend this structure with additional fields. Because they are so trivial, these *tool references* could have been integrated directly with the *tool mappings*. Instead, they were defined separately, because SNF seemed interested in eventually extending the Schema to use the configuration to store additional tool-related information.

- *Tool mappings* are several-to-several mappings of tools to the process steps they can perform in the process library. Each tool is allowed to collect run-data for any of the processes for which a mapping exists. This way, each tool can perform multiple process steps, and the same process step can be associated with several tools. This meets the goal of separating process step from tool, allowing the construction of complex, extensible process hierarchies separate from tool definitions.

These three pieces comprise an XML Configuration Language that is the key element for the goal of providing customized input forms for each process. The language allows the complete *specification* of those input forms. Additionally, using XML Schema is an important design element in having easily-updatable configurations (only the human-readable XML file need change) and simply supporting software enhancements (adding new object types to the XML Schema is not a complex task, as it is written in human-readable XML itself).

Figure 3-2 shows the overall structure of a configuration file, and Figure 3-3 shows a UML diagram of the associated Java data-structures. Also, see Appendix D for a

and floating-point numbers; this meets the minimum requirements of the *variety-of-field-types* goal (see Section 2.2.1), though it can be easily extended.

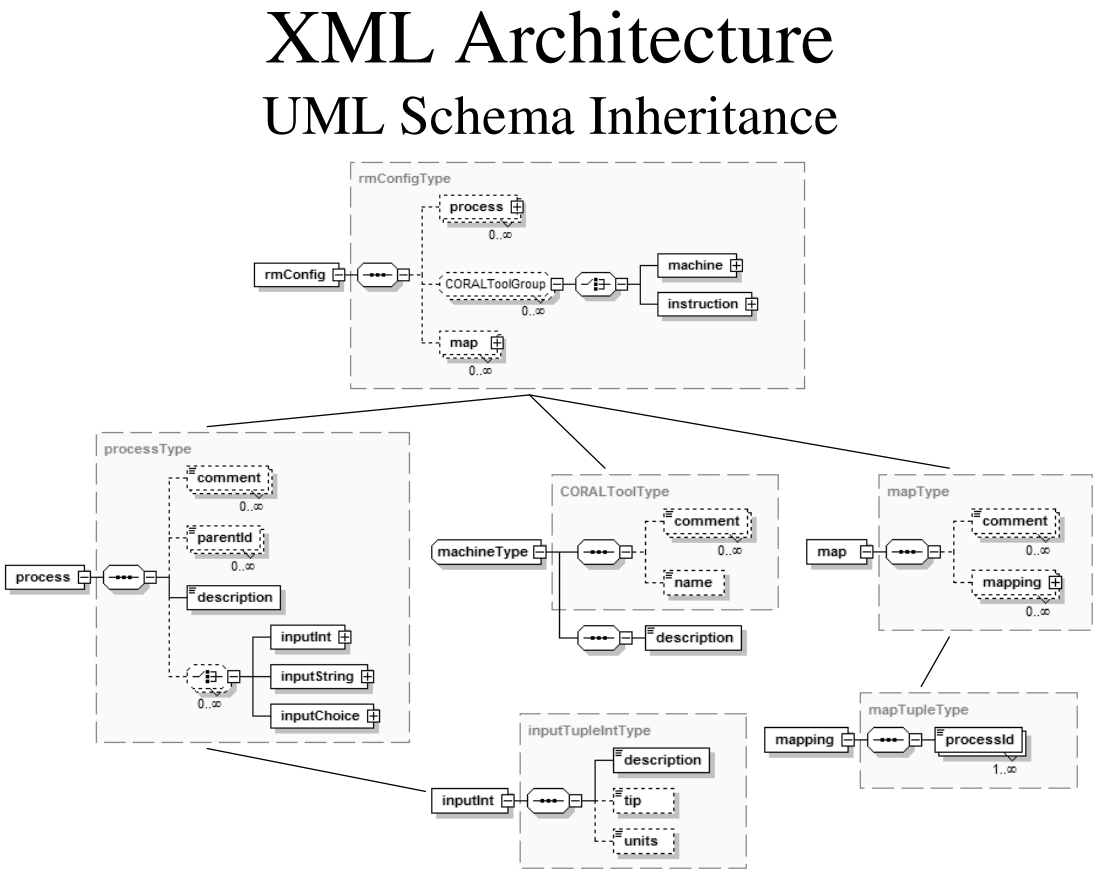

Figure 3-3: XML Architecture: UML layout of configuration file Java data structures (Castor-converted from XML Schema).

sample configuration and Appendix E for the entire XML Schema.

3.3.2 Run-data Storage Format

To meet the goals in the previous chapter, RM needs to store run-data in a persistent store accessible to future accounting scripts. To fit with CORAL's current design, the Oracle and Postgres databases must be supported.[2] However, the database structure may not be tied to a specific configuration, or the *flexibility* goals would not be achieved. Ideally, the database structure must be flexible enough to not only seamlessly support configuration modifications but also to support new input fields without modification (to meet the *easily-enhanced* goal in Section 2.2).

RM's design achieves this flexible structure by carefully defining two tables to store data in a configuration-independent format: `rundata` and `rundata_tuples`. Whenever a piece of run-data is saved, a single `rundata` entry is stored in the database. This entry contains common information such as: *what process was performed?, which tool was used?, what user performed the process?, when was the process performed?.* Also, every `rundata` entry is assigned a unique Transaction Id (tid), which is never reused. The content of the input form is stored in the other table, `rundata_tuples`. An entry is stored in this table for each input-field in the input form, and each entry is associated with the aforementioned tid, to link it back to its `rundata` entry. The `rundata_tuples` entries contain information such as the name of the field and its input type (i.e. string, integer, etc), and they also contain several generic data-storage fields, which are used variously by the different input types. With this structure, run-data size is effectively unlimited and not tied to a specific input type.[3]

See Appendix B for a detailed specification of the database.

[2]Thankfully, as far as the features RM uses, these two databases are essentially the same. See Appendix B to view the differences in data-types between the two, and see Section 3.4.1 for a discussion on Oracle-specific changes needed to make one feature work.

[3]Currently, only an integer and string field are defined, but this is not necessarily a limitation; string fields can be used to store virtually any type of simple Java type. The only loss is some speed in conversion time, but blazing execution speed is not a goal.

3.3.3 Castor

Castor is a Java data-binding framework which is "basically the shortest path between Java objects, XML documents, SQL tables and LDAP directories [5]." RM uses two Castor packages: Castor XML and Castor Java Data Objects (JDO). Castor XML translates XML Schemas into a package of Java data-structures, and then it unmarshalls instances of those Schemas into instances of the Java data-structures. Castor JDO supports user-defined mappings between Java data-structures and SQL tables, which can be used to load and save Java objects to a relational database.

These two tools are used to almost completely eliminate code redundancy and organize code more cleanly (contributing significantly to the *easily-enhanced and* no-redundancy goals from Section 2.2). All data-structures are defined only in the XML Schema[4], including both the structures needed to write an RM configuration and those used to store run-data. Therefore, modifying data-structures requires only modifying the Schema and rebuilding. Without Castor XML, the data-structures would need to be defined in the XML Schema (for verification) and equivalent structures would need to be defined in Java (for processing). Even the CORBA Interface Definition Language (IDL) takes advantage of Castor, by using XML strings to transmit complex data across the wire, which is unmarshalled to Java on either end of the remote connection. Therefore not even the IDL need change when the configuration structure does. In fact, changing the structure of the configuration or the database often requires changes only to the XML Schema and a Castor JDO mapping file, with a few modification to the Java code which manipulates the generated data-structures.

Castor JDO and XML, together with RM's database and XML Schema design, provide a framework for the rest of the client and server, which only needs to manipulate and display the Java data-structures generated by Castor. Castor handles all the data-storage details. For example, RM configuration updates can occur almost automatically; a single method call unmarshalls the updated configuration from XML into Castor's data-structures. Similarly, when run-data is saved, RM builds up

[4]This isn't quite true. As mentioned in the evaluation chapter (see Section 5.2.1), Castor JDO requires an additional definition file, describing how Java structures map to SQL statements.

a `RmRunData` object (which is defined in the Schema and describes the content of the run-data) and uses Castor JDO to map this field-for-field into the database.

Castor is not a perfect out-of-box solution (see Section 5), but its advanced Java-data-binding ability provides a redundancy-eliminating framework for the client and server which is simple to build upon.

3.4 Server Design

The Server uses the Castor XML data-structure framework to provide information to the client and store information in the database. It fits within the CORAL server paradigm, in that it:

- Is CORBA-based.
- Shares tables in a single relational database with other CORBA servers.
- Runs on a central server and is accessed (again, via CORBA) by client applications running remotely.
- Is one of several server applications which run concurrently to provide various functional components to client applications.

The Server is a Java application which uses CORBA to export the following functionality:

- Get global information, such as the configuration's version string.
- Inspect the configuration (i.e. available tools, processes supported by a given tool, process information).
- Store and retrieve run-data.
- Additional administration (i.e. reset the server; set and release view locks, as discussed in Section 3.5).

The entire IDL specification is available in Appendix F. Using the XML Schema design above, Castor obviates many of the most challenging aspects of server design; the Java code is mostly a front end for Castor's tools that also performs some data-structure manipulation.

As the Java code is discussed in the following subsections, refer to Figure 3-4 to study the class relationships.

3.4.1 The `RmHandlers`

The Server code is built on the `RmHandler` interface. An `RmHandler` is nothing more than a class which has methods to provide descriptive information on itself, but it acts as a base class for objects which provide server functionality. The Server provides the following handler objects:

`RmConfigHandler` Uses Castor XML to load a new configuration, validate it, and provide the configuration-inspection capabilities to the remote interface.

`RmRunDataHandler` Uses Castor JDO to map `RmRunData` objects to and from the database, providing run-data storage and retrieval capabilities to the remote interface. (This task is not entirely trivial, as it requires some burdensome details: commitment-checks[5]; transaction-id propagation when saving[6]; and care to leave placeholders when deleting the last element, so transaction ids will not be reused.)

`RmGlobalContainerHandler` A third database table, `rm_globals`, is used to store character large objects (CLOBs) in the database, and `RmGlobalContainerHandler` uses Castor JDO to map the Java structure `RmGlobalContainer`[7] to and from this SQL table (see definition in Appendix A). `RmGlobalContainerHandler`'s

[5]Data can be flagged as committed or uncommitted (as required in Section 2.2.3), and `RmRunDataHandler` must not allow committed data to be updated arbitrarily. The database structure (see Appendix B) explains the committed flag's implementation.

[6]The transaction id must be copied from the `rundata` entry to each `rundata_tuples` entry (see Section 3.3.2 for more information).

[7]Like other structures, this is defined in the XML Schema.

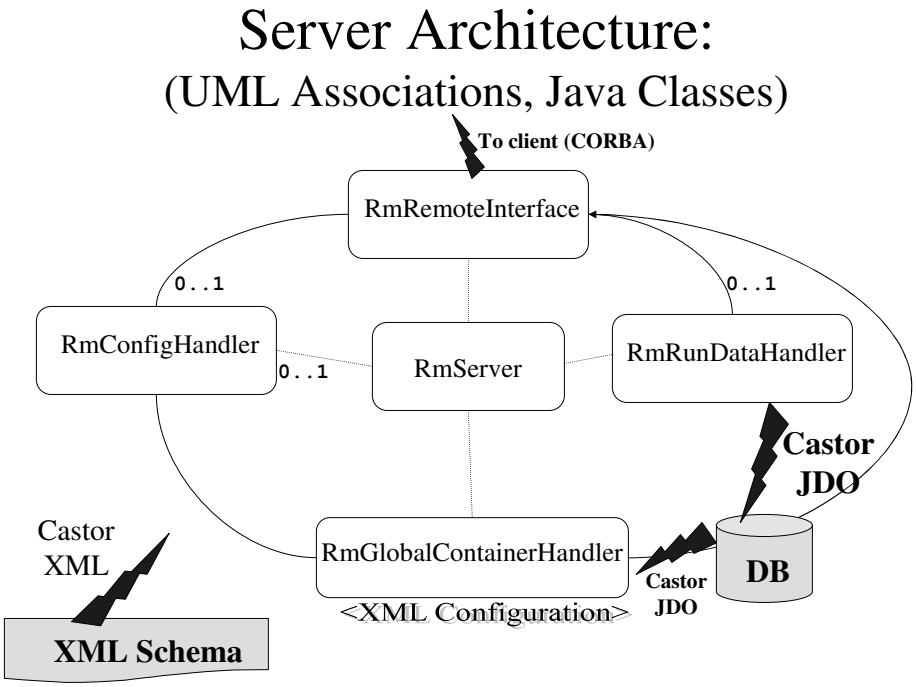

Figure 3-4: Server Architecture UML Overview.

functionality is used primarily to store and load the configuration in the database. This handler exports functionality to the remote interface to load and save containers. `RmGlobalContainerHandler` is also not trivial, because Oracle only allows 4-kilobyte CLOBs, many configurations will be greater than 4-kilobytes, and Oracle support is necessary to meet with the *CORAL-Paradigm* goal stated in Section 2.2.3. Therefore, this Handler can optionally splice CLOBs into arbitrarily large chunks before saving and can load both spliced- and non-spliced-records.

3.4.2 `RmServer`

A main class, `RmServer,` loads the configuration, instantiates these `RmHandlers,` and passes the Handlers to an `RmRemoteInterfaceImpl` (which implements the IDL in Appendix F), which it then registers with CORBA. `RmServer` then simply waits until it receives a signal from `RmRemoteInterfaceImpl` to either shutdown or re-

initialize itself (via the inter-thread communication capabilities of `Object.notify()`). This way, a remote application (such as the Admin Tool described in Section 3.6) can command the server to reset, which implicitly loads any configuration updates. `RmGlobalContainerHandler` additionally provides the remote interface to load and save configurations remotely, opening the door for the Admin Tool and satisfying the server part of the goal of easy configuration updates (defined in Section 2.2.1).

As an added measure of protection, `RmServer` has a last-known-good configuration functionality. That is, if `RmConfigHandler` cannot validate the main configuration, `RmServer` tries to load the last successfully validated configuration. Once a configuration is validated, it becomes the last-known-good configuration. This small addition provides significant protection against administrators bringing down the Server by installing an invalid configuration.[8]

3.4.3 `RmRemoteInterfaceImpl`

The final piece of the server, `RmRemoteInterfaceImpl`, as previously mentioned, implements the IDL remote interface. It is initialized with an array of `RmHandlers` and uses Java's reflection technology (described in Appendix A) to determine the `RmHandler`'s capabilities and to automatically pass calls through to them. Whenever a remote method is called, `RmRemoteInterfaceImpl` searches through the public method lists of all available `RmHandlers` (again, using reflection), and calls the first method it finds with the same signature. If no method exists, a CORBA exception is thrown back to the client. Also, `RmRemoteInterfaceImpl` automatically converts between `String`s of XML and Castor-generated Java objects. This means that if a method has a `String` in its signature, `RmRemoteInterfaceImpl` automatically searches for a Handler method with a Castor-generated Java object at that point in the signature, and it unmarshals that `String` to Java. Similarly, return types that are Castor-generated Java objects are automatically converted to XML

[8]Note that an XML Schema-aware programmer's editor, such as jEdit, can provide significant configuration validation before it is sent to the server [17]. However, jEdit is unable to check for information that cannot be specified in the Schema, for example that every tool mapping references a valid tool id. Therefore, the added protection of a last-known-good configuration backup is beneficial.

`Strings`. Lastly, `RmRemoteInterfaceImpl` catches `null` returns before they are returned through CORBA (which generates a general CORBA exception that has several meanings) and throws an RM `NullReturnException` through CORBA, which is defined in RM's IDL (listed in Appendix F). This way, the client can determine whether a `null` was returned (which could be a legal return value in some cases) or whether an actual communication failure occurred. These several pieces of functionality together make `RmRemoteInterfaceImpl` a powerful tool to wrap CORBA around the complex functionality in the various `RmHandler` objects.

3.4.4 Conclusion

Together, the Server functions as a standalone CORBA application server that is approximately compatible with CORAL. Very few modifications are anticipated to integrate this module with CORAL. Possible changes that will need to be made include:

- Changing the way the Server registers with the CORBA nameserver in order to be compatible with the http-based protocol that was used when the CORAL core was developed.

- Modifying the Server to store a copy of some of its data in existing CORAL tables. (Most likely this will entail copying the transaction id of inputted run-data to an entry in the existing CORAL accounting tables.)

- Possibly partially using the CORAL Persistence Engine to store some data.[9]

All of these changes should not take more than a week of developer time to someone who is familiar with the code layout, thanks to the modular Handler-oriented design of the Server.

[9]The Persistence Engine is SNF's alternative to Castor JDO, written before Castor JDO was developed. It will probably not be necessary to support it, as there is no hard requirement that all CORAL modules must use the Persistence Engine.

3.5 Client Design

The Client is divided into two halves: the graphical components and the internal components. Central to the graphical components is `RmDemo`, a large class (with several inner classes) which defines all aspects of the Graphical User Interface (GUI) which will not be directly reused in CORAL. These include a login window (which does not authenticate users, but merely passes a user id to the RM GUI), a Tool List window which allows the user to enable and disable tools (mimicking the CORAL Object Hierarchy), and a Configuration Management window which puts a graphical front-end on `RmTool's` configuration-update capabilities, as described in the Admin Tool Design section. The innards of `RmDemo` will not be discussed in this thesis, as its design is incidental to the project, providing only essential wrapping to test the Run Manager as a stand-alone system. It is worth noting, however, that `RmDemo` was able to take advantage of some technology developed for other parts of RM; specifically, `RmDemo` is able to store its tool-state (enabled/disabled) directly into the database by remote calls into `RmGlobalContainerHandler.` This ensures consistent enabled/disabled information at all clients.

As the Client's code structure is discussed in the following subsections, refer to to Figure 3-5 to study the class relationships. Also, Figure 3-6 shows a sample UML instance diagram that can help one understand how the classes described below are instantiated to collect run-data.

3.5.1 Graphical components

`RmGui` is the heart of the reusable graphical components which have been designed to be integrated into CORAL. `RmGui` is a Swing JFrame which provides the skeleton GUI for collecting run-data and acts as both a process-selector and a run-data collection form. `RmGui` is designed to be easily integrable with CORAL. Interaction between a client container (i.e. the `RmDemo` or, eventually, the CORAL client) and the RM GUI consists mainly of initializing an instance of `RmGui` with a user id, tool name, and a copy of the CORBA Run Manager remote object (from the server). The

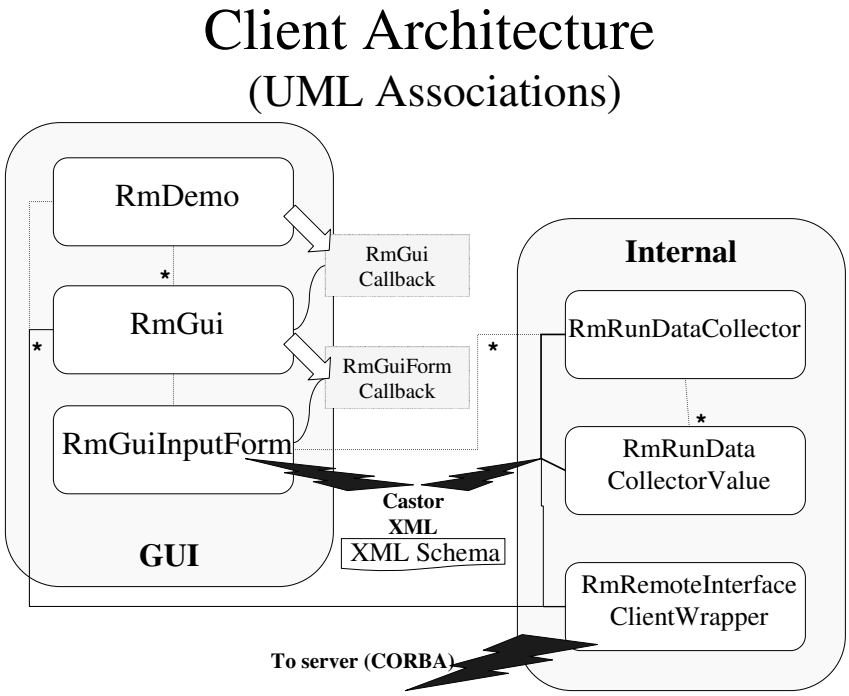

Figure 3-5: Client Architecture UML Overview. Note that inheritance from Swing classes is not included in this diagram.

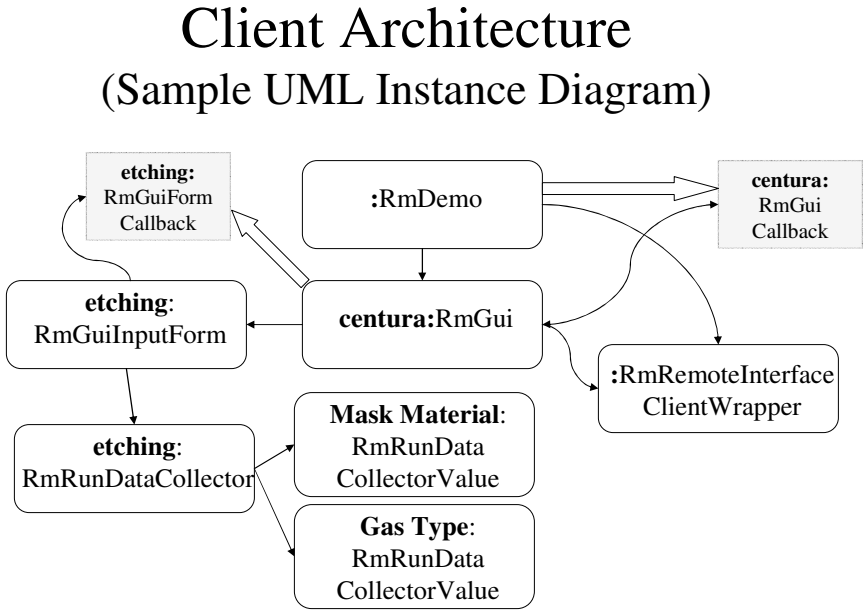

Figure 3-6: Client Architecture UML Instance Diagram. Shows an example instance structure when a simple process step is being edited in the run-data editor.

only additional necessary work is to register the client container as a callback with `RmGui` and to respond to the events in the `RmGuiCallback` interface, which include notifications of fatal errors, successful save, and successful commit. `RmGui` also provides some optional public methods which can change its state as the client container desires. These include the ability to force the current run-data collection to abort and to transition from *collecting data* to *finalizing data* (see next paragraph). Because of this simple interaction model with a client container, the goal of easy integration is also met on the client side.

`RmGui` is essentially a simple state machine. It can be in one of four states at any given time: *process selection, collecting data, finalizing data,* and *finished*. *Process selection* occurs just after initialization, when the `RmGui` knows what tool is being used, but not what process the tool is performing; this step is skipped if only one process is available. *Collecting data* occurs next, displaying an input form and a Save button, and it is used to save a draft of collected data to the server. *Finalizing data* occurs only on explicit call from the client container, as previously described; it should occur just before a machine is disabled. This state looks approximately the same as *collecting data*, except that saving the data commits it so that it cannot be changed. This committed/uncommitted model achieves the client-side aspect of the *committing-model* goal (see Section 2.2.3). Collecting data can occur many times, but *finalizing data* can occur only once, after which the record is marked as committed in the database, RM will refuse to edit it, and it can be used by accounting scripts.

`RmGui` also achieves the *view-lock goal*, described in Section 2.2.3. The Server provides remote methods to set and release view-locks on a piece of run-data. When a user is *collecting or finalizing* data, `RmGui` tries to set the viewlock on their run-data. If it fails, that user is editing data on another terminal, and `RmGui` refuses to open.

The only other graphical element of the RM GUI is `RmGuiInputForm,` which fills in the most complex graphical piece. This class is a Java `JPanel` which displays the actual run-data input form and error messages. Therefore `RmGuiInputForm` gets process information from the server and displays it in a custom form on screen. It

also informs `RmGui` via a callback whenever its state changes.

Note that only display logic is included in `RmGuiInputForm`; the logic to convert a process description into an `RmRunData` object (which can be saved via the server; see Section 3.4) is handled in separate, internal client components (which are discussed below). In software development terms, this design separates the *model* from the *view*, which makes it easy to write a new user interface on top of the internal client components. In fact, for early testing, the author wrote a simple text-based user interface to test the internal components. Additionally, this *model-view* design helps the system meet the *easily-enhanced* goal from Section 2.2, because the *model-view* code separates internal complexity from display logic, making the code easier to understand. It also helps meet the *quick-integration goal*, in that rewriting parts of the GUI, should that be necessary, would not necessitate throwing out any of the internal client logic.

Details on how the GUI is used can be found in the User Manual in the next chapter. Pictures of the RM GUI can also be found there. The remainder of this section will be dedicated to discussing the internal client components.

3.5.2 Internal components

`RmRunDataCollector` is the model in the model-view design. It is the bridge between an RM `Process` object (which describes a process-step and is retrieved from the XML configuration through the server; see `Process` in the Schema listing in Appendix E) and an `RmRunData` object (which is stored in the database). This class creates `RmRunDataCollectorValue` objects for each input field in the process-step. These objects provide inspection and validation methods on inputted data. Together, `RmRunDataCollector` and its collection of `RmRunDataCollectorValue` objects can validate inputted data and output an `RmRunData` object to be stored in the database. In `RmGuiInputForm`, validation errors (faults) are displayed as red messages below the input field, and they are caused

when an invalid value is entered in a field according to the configuration.[10] Recall that the goals in Section 2.2.1 included the requirement that fields optionally allow multiple copies (see Section 2.2.1). Only one `RmRunDataCollectorValue` exists for all copies of a field, and public methods exist to add and remove copies from that `RmRunDataCollectorValue`. `RmGuiInputForm` takes advantage of this to provide add/remove buttons next to fields when appropriate.

`RmRemoteInterfaceClientWrapper` wraps around an `RmRemoteInterface` remote object retrieved through CORAL, and it helps achieve the no-redundancy goal by providing much of the same functionality as `RmRemoteInterfaceImpl`, which was described in the Server design section (3.4). Specifically: the class automatically marshals/unmarshalls XML strings to and from Castor-generated Java objects before sending across the wire, and it converts the RM CORBA Exceptions (i.e. `NullReturnException` and `ServerErrorException`) back to an appropriate non-remote form (i.e. a `null` return value and a local exception). A `RmRemoteInterfaceClientWrapper` is passed to `RmGui` on initialization, which uses it for all its Server communication.

3.5.3 Conclusion

By providing a Client which interprets and displays the language defined in the XML Schema (see Section 3.3.1), the Java-side of the flexibility goals (see Section 2.2.1), which the XML Schema language defines, are implicitly met. Also, because `RmDemo` is so easily severable from `RmGui` (and because of `RmGui`'s simple API), the *easy integration* goals in Section 2.2.3 are also met.

Note that all of these classes depend on the Java objects which Castor generated from the XML Schema (see the top of this chapter), because these are used to examine subsets of the configuration which are sent across the wire (i.e. process-step descriptions).

[10] Common faults include a number which was not within minimum/maximum bounds or a non-optional field which has no value. See the input-type definitions in the Schema listing in Appendix E for a fuller understanding of the available restrictions.

3.6 Admin Tool Design

Another goal of RM is to provide functionality to update the configuration painlessly (see Section 2.2.1). As discussed before, the Server supports remote method calls to update the configuration and reset the server (see Section 3.4), which provides the framework to support this goal. The other necessary piece is a client tool that puts this functionality in an adminstrator's toolbox.

Therefore an Admin Tool is provided which can update the configuration in the database from a local file and also signal a Server reset (causing it to load the new configuration). Because simplifying future enhancements is also a goal (see Section 2.2.2), the Admin Tool was designed to be easily extensible to provide new functionality.

The Admin Tool consists of two classes: `ToolBase` and `RmTool`. `ToolBase` defines a Java Reflection-based framework for writing command line tools (see Appendix A for a brief explanation of Reflection), and `RmTool` uses `ToolBase` to provide Run Manager administration abilities.

`ToolBase` defines a set of rules for writing method signatures, which, when followed, allow `ToolBase` to determine the capabilities of any administration tool, provide descriptions of each ability to the command prompt, and execute any number of these capabilities in one line, by parsing command-line options and executing each capability in order.

Specifically, to create a derived class (a *tool*) using the `ToolBase` framework, these rules must be followed:

1. A description of the tool must be provided in a global `String descr`.

2. Every tool capability should consist of a method of the form `public boolean tool_[name](String *)` (that is, a method with any number of String inputs). The method must return true on success or false if the program should terminate without processing any more command-line options and instead `System.exit(1)`.

3. Every method should have a corresponding global `String descr_[name]` which describes it.

4. Each method *may* have some corresponding `String descr_[name]_[arg#]`, which describes the purpose of an argument, where arg# is numbered 1 to n.

`RmTool` uses `ToolBase` to implement the Admin Tool and provide the following command-line options:

`debug` : Turns on debugging information.

`nameserver` *iiop://-name-:port-* : Sets the CORBA nameserver used to find the remote object.

`getxml` *<xml keyname> <local filename>* : Retrieves xml from the database with the given key.

`putxml` *<xml keyname> <local filename>* : Stores xml with the given key into the database.

`cycleserver` : Tells the server to reset itself.

Updating the configuration on the server is as simple as running `RmTool` with the command line options:

```
-nameserver iiop://[name]:[port] -putxml instance [filename] -cycleserver
```

Here `filename` is the name of the new configuration file.[11] Alternately, `RmDemo` provides a graphical front end to select a file and execute exactly this set of command line parameters, to simplify updates for those not savvy with the commmand line.

`RmTool` can be easily updated to provide new capabilities. For example, MTL developers are discussing storing the configuration in multiple containers (i.e. separating each process step definition into its own container), so that individual pieces

[11] "Instance" on the command line above is the key name used for the configuration in the database, which is defined among other constants in a utility class.

could be updated independently. Writing a new method `tool_putpartialxml()` to store only a part of a given XML file in the database and adding a few Strings to describe the method would automatically add this capability to the host of command line options.

3.7 Project Management Design

Technical design aside, project management design was the most critical element in determining the success of this thesis. Because this thesis was defined in an isolated environment (i.e. only one developer/designer), developing methods for good communication was essential, so that the author's progress was always available to the MTL Operations team. Additionally, because this thesis project was designed for integration into a larger system and did not begin with a detailed pre-defined specification, the MTL team needed the ability to review requirements and code on a regular basis. Finally, because it was essential that SNF be satisfied enough with RM to integrate it into their CVS tree (see Section 2.2.3), communication with them was also important. For all these reasons, the following project-management tools were implemented:

Initial specification: Before any code was written, over a month was spent in discussions with Dr. Diadiuk (this thesis project's advisor) and developers at SNF and (primarily) MTL to determine the system's requirements and to create a high-level specification. During this time, many important goals and design-decisions were made that saved the author from many *try, delete, repeat* cycles during development. For example, many of the *flexibility* goals were decided in this initial time period (see Section 2.2.1), including the decisions to use XML, XML Schema, and a process-step hierarchy.

Weekly status reports / status meetings: Early every week, the author updated the MTL team on the project's progress via email. This was followed by a weekly design meeting with other developers, where design questions could be resolved.

It is the author's opinion that the quick-turnaround time on design issues yielded a higher-quality product than would have been possible with completely independent design. For example, some database-related issues were discussed and solved at the weekly meetings. At one meeting, the team suggested adding fully-reconstructable database records to the list of goals (see Section 2.2.2).[12]

Quick code cycle / weekly testing: A new code release was made available nearly every week, together with a test application that put the newest release through its paces. Although the other developers seldom looked at the intermediate code releases, writing new test applications every week also ensured a high-quality system, as many obscure errors were discovered early that might not even have been apparent with integration testing. For example, the client model accidentally allowed some illegal input values that could not be entered in the client view, so the error in the model would not have been caught without separate testing.

Online research and task lists: A Swiki (defined in Appendix A) was used to provide more immediate and detailed progress feedback than the weekly reports. The author posted his research, design decisions, and updated a task list several times a week. Others on the MTL Operations team could log in with read-only access and post comments. Although the design decisions were not often reviewed online, the task list became a fundamental centerpiece to the thesis design. It listed all the top-level technical tasks (including design research) needed to complete the project, their completion percentage, and the number of hours spent. A copy of the task list can be found in Appendix C. Not only did the task list help the author track time spent so as to improve his time-estimation skills, it also assured Dr. Diadiuk that progress was being made and that the project would be completed on-time. This was especially helpful to Dr. Diadiuk, as she is not familiar with all of the detailed tasks necessary in

[12]This was chosen as opposed to an earlier, poorer alternative which allowed the configuration to change or data to be reconstructed but not both.

software design [4]. After the initial project was completed at the beginning of June, a "Laundry List" was added to the task list, which was a more free-form bug-tracking list. By posting it publicly on the Swiki, the MTL team felt more able to discuss solutions and add additional bugs.

Major Design Reviews / Demos: Twice during the project's design, demonstrations were presented as an opportunity for design review. The first demonstration was for the MTL team on June 5^{th}, just after completion of the initial release of the finished project. During this demonstration, several design changes were recommended, which helped ensure the final project would mesh with MTL's needs. Most significantly, viewlocks were recommended (see Sections 2.2.3 and 3.5 for more information), as was the ability to enable multiple machines per client computer (which is supported in CORAL, but was not supported at the time in RM). The second demonstration was for the SNF team on June 26^{th}, just after the design changes from the first demonstration had been implemented. The demonstration did not result in many suggestions for change, but it did present an attitude of openness from SNF which certainly contributed to their positive reception of RM.

Heavy documentation: As design completed, a focus was placed on documentation (see Section 2.2.2). Several weeks were spent adding JavaDoc API documentation to the code where it was lacking and redesigning portions of the API so that it was easier to understand.[13] Besides adding some ninety pages of JavaDoc, which thoroughly describes how a programmer can *use* RM's technology, a detailed technical slide presentation was made, describing RM's design. The slides use the Unified Modeling Language (UML) to signify class interactions and include detailed notes. These slides were provided to SNF for the June 26^{th} demo, and they will certainly be used by the MTL team as they integrate (and, someday, refactor) parts of RM's code. Additionally, many of the UML

[13]The most significant redesigns were the interactions between the reusable `RmGui` and the temporary `RmDemo` class, as it was critical for these to be simple for future integration with CORAL to eventually succeed. (See Section 2.2.3.)

slides are reused in this document.

Together, these project-management tools helped to ensure a high-quality product by ensuring good communication between RM's author, other developers, and potential users of RM.

Chapter 4

User Manual

The following sections describe the two use cases of the Run Manager: a user entering data and an adminstrator updating the configuration. These use cases are based on RM running within `RmDemo`, but the steps will be similar when integrated with CORAL (and all steps involving the Run Manager window will be equivalent). A picture of the `RmDemo` tool window and the Run Manager main window (in data-entry mode) are provided in Figures 4-1 and 4-2.

4.1 Using the Run Manager

This section describes the use of RM as a normal (non-administrator) user.

1. **Log in:** When you start up RM, you will see a login window. Enter any username you like, though preferably one unique to yourself. It is used to store your run-data in the database.

2. **CORAL Tool Window.** This is a 'fake' tool enabling window that mimics the behavior of how the Run Manager will integrate with CORAL. It displays the list of tool names from the server configuration file. When you 'Enable' a tool, the run-data collector window (also called the Run Manager window) will appear and the tool will become highlighted. You can enable as many tools as you like, unless they are already enabled at another terminal (in which case,

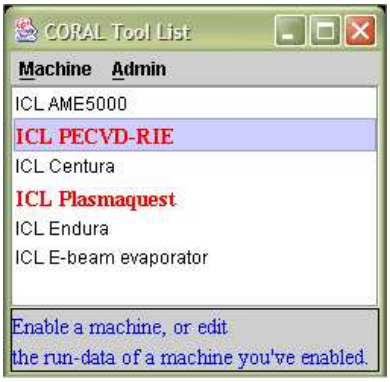

Figure 4-1: The CORAL Tool window.

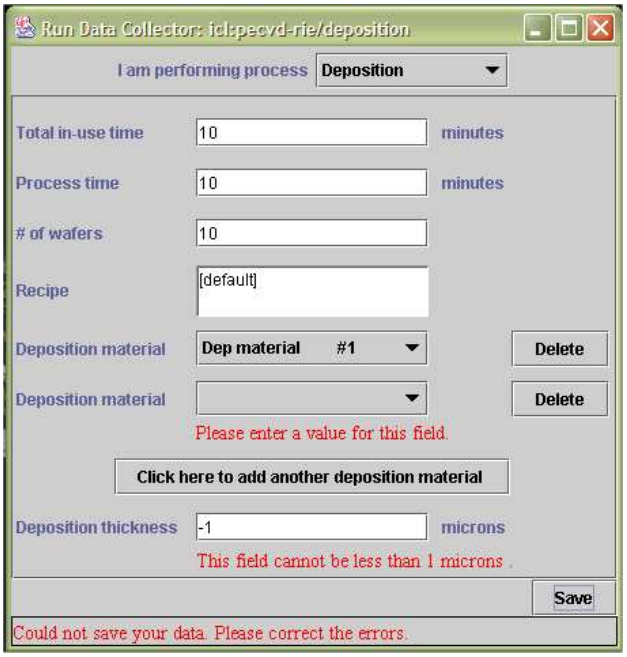

Figure 4-2: The main Run Manager window.

they will already be highlighted). If you previously enabled a tool, have not yet disabled it, and its run-data window is not currently visible, you may choose "Edit Run-Data" (see step 5).

3. **Run Manager Window: Choose a Process.** If you enabled a tool with multiple processes, pick the process you will perform now. If the machine supports only one process, this step is performed automatically for you.

4. **Run Manager Window: Enter Data.** Enter data into the provided fields. Note that some fields can be duplicated or deleted, because you might want to enter multiple answers; notice the appropriate "Add a new..." and "Delete" buttons. When you are finished, click "Save". If your data has errors, it will display them and ask you to correct them. Otherwise, the data will be copied to the server and the window will close.

5. **Run Manager Window: Edit Data.** You may edit data you have entered as long as a tool is still enabled. To do this, select a tool you've enabled and for which you've saved some initial data. Choose "Edit Run Data" from the CORAL Tool Window and the Run Manager Window will reappear. You may continue to update your data and save it until you are finished with the machine. If you click save, the window will close again, and you may repeat this editing step by choosing "Edit Run-Data" in step 2. (You can also edit your run-data on a different terminal via the same process, provided you are logged in with the same user id.)

6. **CORAL Object Window: Disable Machine.** When you are finished with a tool, return to the CORAL Tool Window and "Disable" the tool you enabled earlier. The window will ask you to confirm your run data and click "Commit", which has temporarily replaced the "Save" button in the Run Manager window. When you have done this, the system will commit your data on the server and close the Run Manager window. Repeat this process for all machines you have enabled. You will no longer be able to change your run-data after this step, so

ensure that it is correct.

Premature Exit. At any point, you may exit the application by choosing either "Logout" from the Admin menu or closing the CORAL Object Window. When you log in again (with the same user name), your data will be restored as you left it, and you may continue from the Edit Data step.

View locks. Note that many users may be logged in at once, and a single user may be logged in on several terminals. However, MTL Operations have decided that any user will be allowed to edit a single piece of run-data on only one terminal at a time (as described in Section 2.2.3). Try to edit run-data on two terminals and notice the protection against this.[1]

4.2 Administering the Run Manager

The Run Manager demo provides a partial front-end to `RmTool` (described in Section 3.6), so a lab administrator can graphically update the server's configuration.

1. **Log in:** The same as step 1 above, except your login name will also be used to lock the configuration manager.

2. **CORAL Tool Window:** The same as step 2 above, except choose "Modify configuration" from the Admin menu.

3. **Lock warning:** If someone else seems to be using the Configuration Manager, a warning will appear; you would not want to overwrite what they are working on. If you're sure no one else is using the Configuration Manager, override the lock. Otherwise try again later.

[1]Note that you may save your run-data on one terminal, causing the run-data collector to disappear, and then you may edit it (via step 5) on another terminal. The limit is one terminal at a time.

4. **Configuration Manager: Retrieve Configuration.** A small window with three buttons will appear.[2] First, you will want to retrieve the configuration file from the server. Click on the "Get..." button and pick a location to save the configuration file.

5. **Edit the configuration:** Leave the configuration manager window open, and use your favorite XML editor to change the configuration.[3] jEdit is the recommended editor, because it is free and because it can validate against the XML Schema language specification in order to catch errors more easily [17]. Add some new fields to a process, for example.

6. **Configuration Manager: Replace Configuration.** Click on the "Put..." button and choose the file you just modified. A message will appear in a moment explaining that it has been put on the server.

7. **Test configuration:** If you are editing run-data and it seems that your configuration has not changed, your configuration probably had a bug in it and the server reloaded the last-known-good configuration (see Section 3.4.2). If you try to close the configuration window and it complains that the server is not responding, then the server could not load a last-known-good configuration. In either case, check the server log[4] to figure out why the server has failed. If you get really stuck, click on the "Reset" button in the configuration manager window, and it will reload the first configuration that was installed.[5]

[2]Note that on some platforms, you might need to click "Cancel" to the "No floppy disk in drive.." message; this appears to be a problem with Java's File Chooser.

[3]Note: You might need to reformat the XML in your editor; it doesn't always appear correctly when downloaded from the server. This only seems to happen when Postgres is running under Windows.

[4]The server log is the text output of the Server, which gives detailed messages about configuration errors.

[5]This functionality is provided by using `RmTool` to load a predefined container which was saved the first time the Server booted, and then again using `RmTool` to save this container as the new configuration.

Chapter 5

System Evaluation

The following sections evaluate the success of this project against the goals in the first chapter, from both technical and non-technical perspectives, and they consider how the system is an improvement to CAFE.

5.1 Non-technical Evaluation

5.1.1 Project Management Tools

As discussed in the previous chapter, the various methodologies used to manage the project significantly improved the quality of the design and ensured communication between the author and the MTL team. The MTL Operations team agreed that status updates, demonstrations, and weekly meetings were essential to ensure good project design that would meet with the evolving project specification [4, 9, 8]. It became clear from the project managements methodologies that complex software design research is best done collaboratively, or at least with significant input from those with more experience in the project's particular research area.

Dr. Diadiuk was especially pleased by the on-line task list, because it gave her a good perspective on how work was coming along [4], and the other methodologies (access to code, status updates, incremental testing) ensured that the team was always aware of the author's progress. Also, the MTL developers are happy that a large

amount of documentation exists, and this will certainly be useful as the code is integrated into CORAL and maintained over the years [9, 8].

On the other hand, the team did not make use of the Swiki collaborative web environment to a great degree. While Dr. Diadiuk did examine the progress reports and on-line task list, no design discussions occurred on the Swiki. Rather, design changes were always suggested in-person at the weekly meetings, perhaps because it was a convenient time when everyone was together. This issue only emphasizes the importance of weekly meetings, but it does suggest that for projects with only one developer, read-only access to on-line information is sufficient; comments on the Swiki were an unnecessary complexity.

On the whole, however, the project management methodologies were essential in producing a quality product, showing that new, Extreme Programming-like methodologies [2], such as evolving specifications (rather than detailed specifications at the beginning of a project) and constant feedback, are good directions for the software development process.

5.1.2 User Satisfaction (MTL and SNF)

The two teams are quite satisfied, in large part due to the good communication engendered by the software design methodologies.

Because the MTL team was given ample opportunities to give feedback and ask for design modifications, the results met their expectations. The system has the functionality necessary to integrate with CORAL at the MTL site. It is difficult to measure system performance and stability until it is integrated in the larger CORAL context side, but, thanks to both weekly testing and integration testing, no crashes or performance glitches have been found during MTL's exploration of RM.[1] Because of the MTL team's satisfaction, RM will be integrated into CORAL, meeting the primary goal of this thesis.

[1]The one exception is an obscure error that occurs when initializing the XML SAX (Simple API for XML) parser on some clients with some Java Runtime Environments (JRE), and only when Client logging is turned off. However, this appears to be less an error in RM than in certain version of the Java Runtime Environment (JRE).

The SNF team will not be using the Run Manager per se, but their satisfaction was also a primary concern because they control the CORAL CVS repository; therefore, the SNF team's satisfaction with the project was the determining factor in whether RM would eventually be integrated with mainline CORAL releases. The SNF team was afraid that a student developer, because he lacks real-world experience, would have poor design methodologies, unrealistic expectations, and too much focus on client-side and GUI code, ignoring the critical (and more difficult) back-end server code [15, 11]. The MTL team knew this at the beginning of the project and therefore emphasized the importance of project management goals from the outset. Therefore, the author put these design methodologies in place and started designing the system with the back-end first not only because these are increasingly proven design strategies [2], but also to satisfy the SNF team. The strategy worked; SNF was very pleased with the design process and its results, and they have not only agreed to add RM to their CVS repository, but Bill Murray at SNF will take the lead role in integrating RM into CORAL.

5.1.3 Improvements on previous work

As discussed in Section 2.1, meeting the listed criteria would put RM far beyond its predecessor, CAFE, and indeed it has. Not only does it provide a modern graphical user interface, it also uses human-readable and easily updatable configuration files which obviates the need to put critical information in unchecked comment fields. RM's design is significantly more flexible than the data-collection implementation in CAFE.

Additionally, RM goes far beyond the existing capabilities in CORAL. CORAL currently only tracks tool usage time, whereas RM provides a customized set of fields for tracking many details in any tool's processes.

5.2 Technical Evaluation

The following subsections will touch on important design decisions in each area and discuss their success, keeping in mind the goals in Section 2.2.

5.2.1 Database and Configuration Design

The choice of a flexible XML language for configuration specification seems the perfect choice, because it provides a human-readable, portable, and easily maintainable language for specifying configurations. The language was defined to be as separate from Java and database design as possible, which has advantages and disadvantages. On the one hand, it means a configuration language that is not fixed to any specific implementation and is free to describe configurations without tying itself to the terminology of any specific language. On the other hand, tying the system to a Java GUI would have allowed Java-like parameters in the field types, such as specifying event-capturing events; rather than having a `<tooltip>` tag, the alternate design might have had a `<mouseOverEvent>` tag. Though in some ways the latter design has more power, it would not have allowed the same level of flexibility; for example, the client's *model/view* design would not have been possible if the language had been tied to a GUI environment.

Another important design decision in the XML language design was the separation of process steps into a hierarchy of derivable types, to which tools could "map". This separation saved a lot of configuration redundancy (for example, *etching* only needed to be defined once, despite the fact that nearly ten machines in the lab support *etching*). The derivable-types decision also saved a lot of configuration redundancy; because all machines have a few fields in common (such as "In-use Time"), a root process step for all lab machines can be defined from which all other process-steps derive. This derivable-types decision also prevented redundancy in defining a certain set of process-steps in the photolithography group, because several of these process-steps also shared fields. Although these decisions increased Server complexity (to parse the complex structures), they decreased configuration complexity significantly.

Configuration complexity turned out to be a good area to minimize, because the configuration will be maintained by a technician, not a software developer. Borrowing object-separation and derivation from object-oriented programming ideas created a more powerful and compact configuration language than would have otherwise been possible.

Using Castor XML to build Java data-structures out of the XML configuration was a suggestion from *O'Rielly's Java & XML book* [10], and it worked out well. It eliminates the redundancy of modifying Java code, IDL, and the XML Schema definition when data-structures change. Instead, Castor XML generates Java code from the XML Schema, and the IDL passes text strings between client and server, which Castor automatically converts to and from XML on-the-fly in `RmRemoteInterfaceImpl` and `RmRemoteInterfaceClientWrapper`. If this were not the case, the IDL would need to specify complex data structures to hold pieces of the configuration, which would need to be maintained along with the Schema. This unexpected benefit has a slight run-time performance hit (parsing XML is not extremely quick), but it saves so much trouble in maintainability and development time that it seems well worth the small cost. Castor XML also saved development time by essentially writing the XML parser automatically. The Castor XML-based design already saved significant maintenance time when updating the Schema definitions after the June 5^{th} demonstration; significant design changes were made in hours fewer time thanks to this design decision.

The database design was a bit more problematic. The hope was that Castor JDO, because it is integrated with Castor XML, could significantly smooth the transition from XML-input to SQL-output. To some degree, it fit the bill. It can automatically load complex data from tables in the database and convert them to easily manipulable Java objects, which can be re-saved to the database. Unfortunately, there were so many restrictions on how data could be saved and loaded that JDO often became cumbersome. For example, if an object is loaded and the current database transaction is committed and then modified, that modified object can no longer be saved into the database. Rather, the object must be reloaded in a new transaction, the modifications

must be copied to the newly-loaded object, and then the record must be committed. Castor gives reasonable arguments about how these design decisions ensure database integrity [5], but it would be nice to turn off some of these security functions when the server is designed to be the only application writing to certain tables. Another burdensome restriction involves writing an object to the database with an array of JDO-mappable objects inside. In this case, JDO does not automatically write the array; these must each be written to the database manually. Although neither of these restrictions create unsolvable design issues, they do increase code complexity and thus implicitly reduce maintainability.

Castor JDO raises more significant red flags in the areas of redundancy and portability. In Castor XML, data-structures are defined only once in the Schema, and Java objects are built from that definition. Castor JDO, on the other hand, requires an additional set of definitions, describing how Java classes are mapped into the database. Castor JDO does provide an undocumented mapping-file generator, which can be used to help the developer get started, but it does not produce a complete mapping file, and thus the generator cannot be integrated into the automatic build process. Also, Castor JDO supports translation to and from both Oracle and Postgres (which are both required for this project to meet the goals described in Section 2.2), but not entirely automatically. Field types vary from database to database, and Castor JDO supports many of the variations, but the mapping file must be modified to reflect the changes. A better design might have been to use generic names for field types in the mapping file, which Castor JDO could convert on-the-fly to match the appropriate database-specific type. (See Appendix B for more information on database type differences in Oracle and Postgres.) Despite these deficiencies, Castor JDO did produce more maintainable code more easily than using straight JDBC (Java Database Connectivity). Most importantly, all of its code is transaction-safe, and changing the database structure only requires changes to XML files; no Java code need be touched.

A final design decision related to configuration and database design is the choice to store the XML configuration as a CLOB on the server. This has advantages and disadvantages. On the one hand, the configuration is less easily accessible (because it is

hidden in the database). On the other hand, requiring an administration tool to read and write the configuration has the advantage that the tool can also notify the server of a configuration update at the same time. Were this not the case, an administrator might update the configuration and fail to notify the server that it should reset itself, meaning that the configuration changes would not be loaded by the server. Another advantage is that the configuration need not be on the same computer as the server, so access to it is not a requirement in administering the configuration; however, other CORAL servers store their configuration on the server machine, so CORAL sites already must be designed such that this concern is not a problem. Overall, storing the configuration in the database is more advantageous than disadvantageous, even though it did increase code complexity (writing `RmGlobalContainerHandler)`. Overall, the availability of new technologies such as XML, XML Schema, freely available SQL databases, and data-binding toolkits like Castor allow the creation of simpler, more robust, more maintainable, and more flexible software designs than were possible a few years ago. Even though the full path from XML to Java to SQL is still somewhat rocky, even with the best freely-available tools, the author feels that the tools discovered for this project allowed it to boast a design far superior (and more easily-implementable and enhancable) than that of its predecessors a few years ago.

5.2.2 Server Architecture

The Server architecture is relatively straightforward compared to the complex design decisions made in developing the configuration language and database architectures.

One important design decision was the use of an `RmHandler`-based remote-object architecture which relies on Java Reflection technology (see Appendix A), because it provides automatic marshalling/unmarsahlling of Castor-XML objects on the client and server. This decision significantly simplified code maintainence. By using Reflection and `RmHandler` objects, providing a new remote capability is nearly trivial. It involves just adding a method to the `RmHandler` class where it best fits and then adding simple one- or two-line stubs to the IDL, `RmRemoteInterfaceImpl`, and `RmRemoteInterfaceClientWrapper`. Before this Reflection-based structure was in

place, complex code to convert `null` values, exceptions, and Castor Java objects to their CORBA equivalents needed to be written every time a new remote method was added. By using Reflection, only simple stubs need be written and all the complex code can be placed in an appropriate `RmHandler`. This Reflection-based automation could save nearly an hour of development time every time a remote method is added to the system.

The rest of the Server simply implements, with proper error handling and validation, the design decisions made in the previous section.

5.2.3 Client and Admin Tool architecture

The RM client's architecture attempts to simplify and modularize what is actually a complex assortment of user operations on data retrieved from the server. This is partly accomplished by the *model/view* design, which separates the user interface from the actual logic which manipulates data retrieved from the server (and pending save to the server). This is a design pattern utilized by Swing, Java's GUI architecture, and overall it created a modularized design that sped development time, aided maintainability, and helped ensure stability (by being able to test the model separate from the view). However, due to time constraints, the model was not totally separated from the view. Specifically, because `RmRunDataCollectorValue` objects are lightweight wrappers around database `Element` objects (single fields inside an `RmRunData` object), the view must contain logic on how to store its view into an `Element`. This essentially involves deciding whether to store its data as an integer or string or both. This logic is really part of the *model* and ought to be fully separated from the *view*. Nonetheless, despite the limitations due to time constraints, this design decision created a more maintainable and stable system.

Another important design decision in the Client was the designed severability of `RmGui` from `RmDemo`, the latter of which will be replaced by the existing CORAL client. This partially meets the easy-integration requirement (see Section 2.2.3), in that the programming interface (API) of `RmGui` is simple enough that it is easy to plug a new GUI "container" around it. This design decision slowed development time,

in that care was taken to write a simple interface in `RmGui`. However, the decision will significantly speed up integration, which will be done by a developer other than the author (who is therefore not as familiar with the RM codebase). Therefore this design decision, whatever the cost, was necessary to meet the goals of RM.

A final important Client design decision was the implementation of viewlocks. As discussed in Sections 2.2.3 and 3.5, viewlocks meet the purpose of preventing a single user from editing data for the same tool on two different terminals at once (because one terminal will inevitably become inconsistent). While viewlocks are a necessary feature, they were implemented on `RmRunData` records to simplify design. That is, an `RmRunData` record can be locked by a specific user; if it is locked, it cannot be edited on another terminal. This has the odd side-effect that a single user could be viewing `RmGui` in *process selection* mode on two terminals at once, because no run-data record yet exists for the tool until a process has been selected. This side-effect cannot be countered, because run-data records are tied to a process-step name, so before a process is selected, no run-data record can exist, and therefore no viewlock can be set. This is a recoverable issue, though, as it does not cause any damage; when a process is selected on both terminals, the one selected second will close, because it notices an `RmRunData` record was already created for its process-step/user-id combination. It might have been a cleaner design decision to implement viewlocks separately from `RmRunData` records, but because a record does exist for the majority of the life-cycle of a run-data form, the side-effects of this minor design flaw are almost unnoticeable.

Overall, the Client has a modular and clean design with an easily-integrable API, despite a minor design flaw in the viewlocks. Supporting compatibility with CORAL slowed development and made the task of writing the Client more difficult, but the Client was able to take advantage of the newest technologies (including use of Castor XML) without being ultimately hindered by backwards compatibility. In the author's opinion, this indicates a good design that mixes compatibility well with innovation.

Chapter 6

Suggestions for further work

CORAL is a next-generation lab manager, and the Run Manager module met all of the design requirements in compatibility and features to provide a next-generation data-collection system which will integrate seamlessly with CORAL. However, technology changes quickly, and new technologies are both available and on the horizon which could allow even better designs in the future, sacrificing only a modicum of compatibility.

SNF has recently considered replacing CORBA with the Simple Object Access Protocol (SOAP) in future CORAL modules. SOAP is slower than CORBA in regard to data transmission, but it supports the `http` protocol and is therefore accessible behind firewalls.[1] This could make CORAL usable to a greater number of remote users, who currently must use CORAL either at the lab or on an Internet connection without a firewall.

Additionally, both SNF and MTL are starting to explore writing a new client (or perhaps just some new client modules) using a language that will run within a web browser. This will eliminate the large initial download of the Java Runtime Environment (JRE) and the Client software, which currently takes almost an hour on a modem connection. There are several server-side languages, such as Java Server

[1] Firewalls are common on many commercial networks to prevent unwanted Internet traffic, but they also often block desired Internet traffic such as CORBA.

Pages (JSP) and PHP: Hypertext Preprocessor (PHP)[2], which generate HTML on-the-fly and give the application the appearance of a web page. JSP is based on (and heavily compatible with) Java, and would therefore not entail a huge learning curve for developers. Additionally, client-side languages exist which can interact with the web-browser on the client side and provide increased interactivity. These include JavaScript and Microsoft's VBScript, but neither language is both standardized and cross-platform capable. SNF and MTL have some questions on how the existing Java client will interact with the new web client during the transition, but for the moment they intend to write some stand-alone reporting modules using these new technologies.

Also on the horizon are new technologies for connecting Java with databases and XML. Castor is a step in simplifying those interconnections, but new packages are appearing which may offer even more advanced capabilities. Sun is now offering a Java XML Binding (JAXB) toolkit, which promises to do the same things Castor XML does, only with the additional benefits of corporate support from Sun, the designers of Java. However, JAXB is still a new product (and hence largely untested by the public), and it is not compatible with XML Schema at the moment. Technologies improving on Castor JDO's data-binding functionality are also becoming available. Sun is developing their own JDO engine, and several commercial JDO engines are coming into availability, each with their own capabilities and pitfalls. Unfortunately, none of the other JDO engines are currently mature and/or free, and therefore these technologies are not yet in common use. However, replacing RM's current technology with the capabilities of new data-binding engines could be an important step forward in building an even more modular, quick, and maintainable data-collection system than RM is currently.

Finally, further out on the horizon, the fabrication facility at University of California Berkeley (UCB) has evolutionary ideas on building a third-generation lab manager to replace CORAL, which they expect to have in use by 2005. Their design will in-

[2]Yes, this is correct. PHP is a recursive acronym. According to their website: *"PHP stands for PHP: Hypertext Preprocessor... This type of acronym is called a recursive acronym [1]."*

volve a flexible XML language to specify bindings directly between Java GUI elements and database tables, using stored procedures in the database to fill in much of the server logic. Significant work will be required to build custom Java GUI components and to write complex stored procedures, but eventually the currently complex array of CORAL modules could be replaced by a single, general module to which XML code and custom GUI components are added. Unfortunately, such a design does tie the XML language almost inevitably to Java (which RM avoids by defining a less general language designed for a specific purpose), and it ties the whole system to a particular relational database design. These technologies have not yet been field-tested in many real-world applications, and many details have not been thought through. Nonetheless, UCB's designs do point to a future where writing (relatively) low-level code in a language like Java will be a thing of the past; many enterprise applications may one day simply be interpreters for a higher-level language specifically designed to define client and server logic. The Run Manager explored the external-language idea, and UCB may be taking it to the next logical step.

Appendix A

Definitions

CORBA: Common Object Request Broker Architecture. CORBA is a "distributed-object architecture." It allows objects written in various language to interoperate transparently across networks [13].

CVS: Concurrent Versions System. A collaborative tool that enables controlled sharing of source code across a group of users. It also provides the ability to store, retrieve, and share development versions of software [19].

field: A single element of run-data for a given process step. Sometimes called a tuple.

IDL: CORBA's Interface Definition Language, used to define interfaces for CORBA objects [13].

Java Reflection: Reflection is a standard Java API (Application Programming Interface) which defines capabilities to perform "introspection" on objects. "Introspection" means gaining details on the underlying structure of an object. For example, one could use Reflection to list an object's public methods and their parameters.

process flow: The collection of steps that make up the research/fabrication on a lot.

process step: A single step in the fabrication process. Usually corresponds to use of a single tool to perform a task; i.e. etching or deposition. Note that in some of the code in the appendices, a process step is called an "augmented process." This refers to the inheritance structure of the XML configuration; an "augmented process" is a process-step containing all its parents' information from the configuration file. In general, the two terms are equivalent.

run-data: The data collected at a particular process step by the Run Manager.

SQL: Simple Query Language. A language used to store and retrieve data in relational databases such as Postgres and Oracle.

Swiki: An open-source collaborative project-management website [3].

tool: A lab tool is any machine or non-machine entity in the lab that is used for research and tracked for accounting or maintenance. Non-machine entities might include renting a uniform, buying wafers, or receiving training.

traveller: A lot history report; essentially a lab notebook describing what a researcher did with the wafers in this lot. This describes the run-data of the research process after-the-fact. A traveller can be used if the user wants to repeat a past process (e.g., to make more identical devices).

UML: Unified Modeling Language. This is a "general-purpose notational language for specifying and visualizing complex software" [6], which defines standards for describing object-oriented programs at a class-hierarchy level. It is used for several of the figures in this text. Note that in this text lightning bolts are used in UML figures to indicate abstracted detail.

wafer: "A thin slice of semiconducting material, such as a silicon crystal, upon which microcircuits are constructed by diffusion and deposition of various materials [12]." This is the main object in the fabrication process.

Appendix B

Detailed Database Layout

Figure B-2 shows the UML structure for `RmRunData` and `RmGlobalContainer,` which Castor JDO translates field-for-field to SQL when loading and storing in the database. Because the use of these structures are described in Section 3.3.2, and because the fields match closely with the SQL field names, they are not described in detail here.

Figure B-1 shows the database layout. `RmRunData` objects are translated into `rundata` entries; `Element` object are translated to `rundata_tuples` entries; and `RmGlobalContainer` objects are translated into `rm_globals` entries. When there are two datatypes given for a single field, the first is the Postgres type and the second is the Oracle type. The meaning of each field is described in detail below.

Finally, Table B.1 shows two sample run-data entries as they are stored in the database. Cross-reference this Table with the field descriptions below to better understand the structure of stored data.

B.1 `rundata`

`tid`: Transaction Id. Each record is assigned a Transaction Id (tid), which remains constant even after updates; this tid connects `rundata_tuples` entries to a rundata entry.

`process, coraltoolid`: Process id and CORAL tool id from the configuration, for data reconstruction by accounting scripts.

`userid`: The user's id that generated this entry. Only one uncommitted entry can exist for a single user id using a single CORAL tool.

`viewlock`: As discussed in the Goals section and the Client section (see Sections 2.2.3 and 3.5), a user may only edit run-data on one terminal at a time, to avoid inconsistency problems. Thus viewlock contains the user's id of the current viewer of this record, or `null/‘‘<none>’’` if it is not locked.

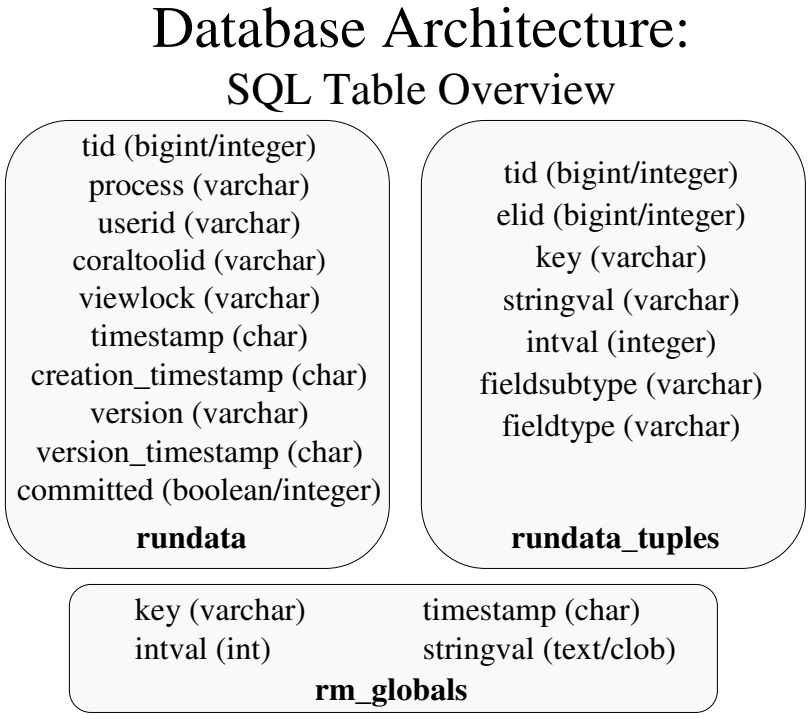

Figure B-1: Database Architecture: SQL tables.

`timestamp/creation_timestamp`: The last modified time and the creation time, stored as a string outputted from Java SQL Timestamp format.[1]

`version/version_timestamp`: The version string describing this configuration (from the XML), and the timestamp of when that configuration was last modified. These are needed so the Run Manager can determine whether the configuration was modified since the data was last edited.[2]

`committed`: True when the record is committed. After it is committed, it can no longer be edited and should consequently be counted in accounting records.

B.2 `rundata_tuples`

`tid`: A transaction id in the rundata table to which this tuple corresponds.

`elid`: An element id (elid), which has no meaning whatsoever and changes every time

[1]Storing as a SQL date instead of a string is not possible due to the limitations in Castor XML/Castor JDO. Castor XML cannot generate Java classes with types to which Castor JDO can translate to a SQL date.

[2]Actually, only the timestamp is strictly necessary, but a version string seems like a potentially useful addition.

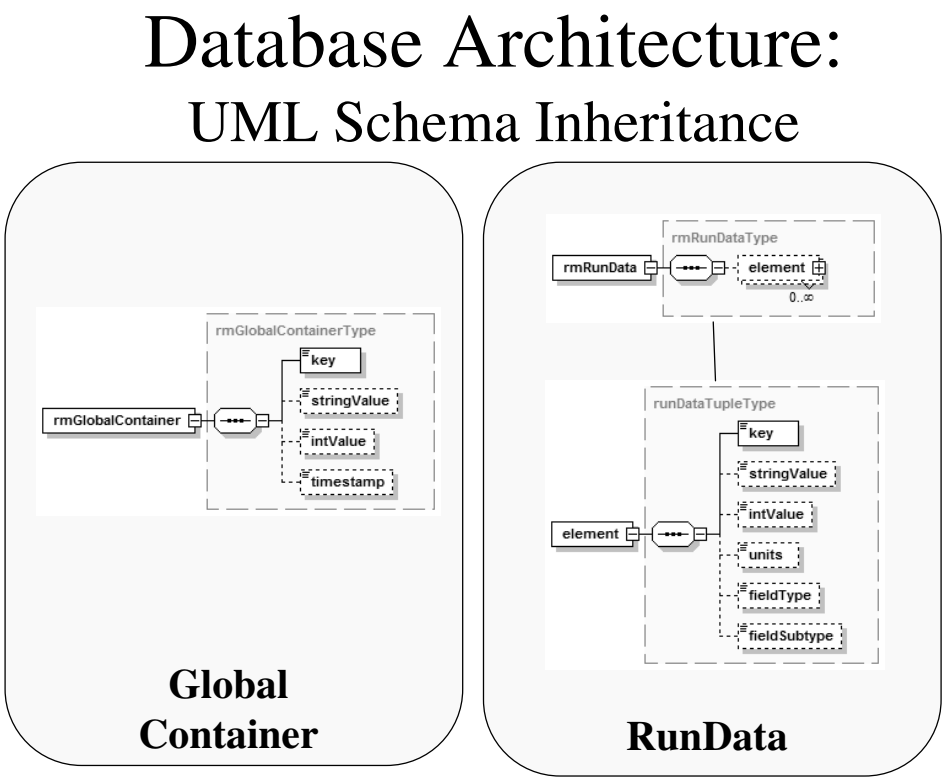

Figure B-2: Database Architecture: UML Java data-structures which map to the database tables (Castor-converted from the XML Schema).

the run-data is updated, but is required by Castor JDO, because JDO requires a unique id field in every table.

`key:` The field id from the configuration (e.x. "process-time").

`stringval, intval` : A string and an integer storage field, which each data type uses differently to store the entered data. Other storage fields could be added to handle other data types.

`fieldsubtype:` In the example configuration in Appendix D, the contents of type="[...]" in `InputChoice` fields is copied here. This will be used for data reconstruction by accounting scripts, which need to know what type of data a multiple-choice field contained.

`fieldtype:` The field type (i.e. `InputChoice, InputInt,` etc).

B.3 rm_globals

`key:` Corresponds to a "filename" for this data.

`timestamp:` A last-modified timestamp, stored in the same format as rundata timestamps, described above (in Section B.1).

`stringval/intval:` A string and an integer storage field, which can be used in any arbitrary way to store data attached to this key.

Additional note: The Run Manager is careful not to reuse tids, but holes of unused tids can result from this[3], and elids are incremented every time a tuple is updated. Therefore, even though bigints are used for ids, the database should be "vacuumed" every few weeks by an administrative script to compress the id space. The author doesn't feel that this is a design flaw, only a minor inconvenience.

[3]Holes in tid numbering will occur whenever a record is deleted, because the Run Manager will refuse to reuse that tid number in order to ensure uniqueness.

Table B.1: Two sample run-data entries stored by the Run Manager.

tid	process	version	userid	coraltoolid	view-lock	timestamp	creation-timestamp	version-timestamp	committed
14	dep	0.134	jklann	icl:endura	jklann	2002-07-17 16:34:29.273		2002-07-16 01:55:00.512	f
13	etch	0.134	jklann	icl:pecvd-rie		2002-07-18 00:04:23.626		2002-07-16 01:55:00.512	f

tid	elid	key	stringval	intval	fieldsubtype	fieldtype
14	212	inuseTime		10		InputInt
14	213	processTime		10		InputInt
14	214	numWafers		10		InputInt
14	215	sizeWafers	6	2	int	InputChoice
14	216	recipe				InputString
14	217	material	Dep material #1	1	string	InputChoice
14	218	thickness		47		InputInt
13	228	inuseTime		10		InputInt
13	229	processTime		10		InputInt
13	230	numWafers		10		InputInt
13	231	sizeWafers	6	2	int	InputChoice
13	232	recipe				InputString
13	233	depth		10		InputInt
13	234	gas	Gas #1	1	string	InputChoice
13	235	mask			string	InputChoice
13	236	rf	2.0			InputFloat

Appendix C

Task List

The task list has been copied from the Swiki (see Section 3.7) used for this thesis in its state after the project completed. This task list was continually updated week-to-week as goals and tasks evolved and as tasks were completed. Not included here is the "laundry list", which is mentioned below as a 55-hour item. However, the task list is only included as an example of a project management tool; the "laundry list" was structured similarly and so is not included.

Note that the task list is not a complete assessment of time spent on this project, as it does not include time spent on: writing this thesis document and the associated proposal, meetings, and administrative and setup tasks. Moreover, all times are "on-task-time," as opposed to "actual time," so coffee breaks, meetings, and other general interruptions are not included.

Compare estimated to actual time and notice that the author learned how much longer tasks take than expected time, in many cases.

Task	**Estimated Time**	**Completion**	**Actual Time**
Setup:			
Set up laptop	10hrs	**DONE**	20hrs
Learn ant (needed to set up a development environment)		**DONE**	5hrs
Set up initial dev environment (with Castor support)		**DONE**	5hrs
Modify dev environment to support CORBA/RMI	5hrs	**DONE**	7hrs
Modify dev environment to support SQL, set up SQL	8hrs	**DONE**	7hrs
Set up local build of CORAL (with Ike's help)	10-15hrs	90%	16hrs
XML:			
Learn XML Schema (notes: XML Schemas)	5hrs	**DONE**	10hrs
Design and implement XML Schema	5hrs	**DONE**	12hrs
Create sample instance document		**DONE**	2hrs

Task	Est. Time	Completion	Actual Time
XML/ Java:			
Read about XML parsing and choose a Java parsing tool (i.e. Castor)	5hrs	**DONE**	5hrs
Set up Castor and modify the XML code to mesh with it	5hrs	**DONE**	5hrs
Write initial & implement server-side interface to get process information	3hrs	**DONE**	6hrs
Write extra validation for XML (makes both code-writing and xml-writing easier)	5hrs	90%	5hrs
Java/ CORBA:			
Read about remote interface techniques (variations on CORBA that will keep code maintainable) verify their reasonableness, and choose one	10hrs	**DONE**	14hrs
Learn IDL & CORBA/RMI stuff	10hrs	**DONE**	8hrs
Implement final remote interface using learned techniques	8hrs	**DONE**	7hrs
rundata:			
Write code to interpret augmented process, get inputs, validate them against process definition see (sample run 5-28)	10hrs	**DONE**	10.5hrs
Write remote interface to store run-data	5hrs	**DONE**	5.5hrs
Learn some PostgresSQL/JDO stuff	10hrs	**DONE**	7.5hrs
Set up run-data storer to actually store run-data	5hrs	**DONE**	8.5hrs
GUI:			
Make the 'dialog box' using existing interfaces	15-20hrs	**DONE**	23hrs
Wrap dialog box in fake 'login' and 'activation' windows to test it		**DONE**	7hrs
Misc:			
Add: load xml from database (& command-line utility)	2hrs	**DONE**	7hrs
Build demo for June 5th (make a gui a webapp, add admin tools)	8hrs	**DONE**	12.5hrs
Work on the laundry list of small changes Prototype Todo List, prepare for SNF demo	?hrs	(continuous)	55hrs
Keep this swiki updated, write emails, make UML, JavaDoc, other docs	30hrs	(continuous)	50hrs
Continually write test code as development continues	30hrs	(continuous)	20hrs

Appendix D

Run Manager Example Configuration

```
<?xml version="1.0"?>
<rm:rmConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:rm="http://www.jeffklann.com/rmconfig1"
      version="0.134"
      versionSchema="0.6"
      xsi:schemaLocation="http://www.jeffklann.com/rmconfig1
        /Users/jklann/dev/rm/labnet/recipe/xml/rmschema.xsd">

    <!--  Sample rmconfig Instance v.0134 - Jeff Klann

        This is a sample instance document for an 'rmconfig'. It defines a lab environment
        with only 'dep' and 'etch' processes, based on my thesis proposal. Despite the simple
        lab environment, it should show off the power and compactness of the schema, and be
        a good testbed to discover where features are lacking/overblown.

        Note that the schema currently requires qualified names in item tags. This clarifies
        the XML a bit, but it adds a lot of bulk (i.e. all the 'rm:'s). Since it's unlikely
        mixed namespaces will be used in a lab configuration, it might be reasonable to turn this
        off.

        Some documentation is below, but see the Schema for more detailed notes.

        Note that the xsi:schemaLocation tag ought to point to the actual location of the
        Schema file. This isn't necessary, but if it does, then editors like jEdit will
        automatically validate your xml against it.

        changelog:
          1.3->1.3.1: fixed a duplicate machine reference, got rid of (now unnecessary)
            instance-time typing.
          1.3.1->1.3.2: several changes to keep up with schema, including adding tool names
          1.3.2->1.3.3: just updated reference to schema version and filename and some comments
          1.3.3->1.3.4: added wafer size
    -->

    <!-- 1. Process Library -->

    <!-- Note the use of all three input types and multiple repitions of some inputs (like
        gas in etch). Notice also the process hierarchy (and that everything derives
        from a root object), and the use of optional components like tips, units, categories,
        and min/max/default input values, where appropriate. -->

    <!-- Also note that in some places default values are used, in some places fields are
        marked as optional, and in other places the run-data window will by default require
        a value to be entered before saving. -->

    <rm:process id="root">
```

```
    <rm:description>This is the root process desription.
             Everything should be derived from it, but it
             should not be used directly.
    </rm:description>
    <rm:inputInt id="inuseTime" default="10" min="0" max="unbounded">
        <rm:description>Total in-use time</rm:description>
        <rm:tip>This is the total time this process took you to complete.</rm:tip>
        <rm:units>minutes</rm:units>
    </rm:inputInt>
</rm:process>

<rm:process id="rootMachine">
    <rm:parentId>root</rm:parentId>
    <rm:description>This is the root machine process description.
            All machine processes should be derived from it,
            but it should not be used directly.</rm:description>
    <rm:inputInt id="processTime" default="10" min="0" max="unbounded">
        <rm:description>Process time</rm:description>
        <rm:tip>This is the total time equipment was used to complete this process.</rm:tip>
        <rm:units>minutes</rm:units>
    </rm:inputInt>
    <rm:inputInt id="numWafers" default="10" min="0" max="100">
        <rm:description># of wafers</rm:description>
    </rm:inputInt>
    <rm:inputChoice id="sizeWafers" type="int" default="2">
        <rm:description>Wafer size</rm:description>
        <rm:units>inches</rm:units>
        <rm:choice id="1">4</rm:choice>
        <rm:choice id="2">6</rm:choice>
    </rm:inputChoice>
    <rm:inputString id="recipe" cat="optional">
        <rm:description>Recipe</rm:description>
        <rm:tip>Optionally enter a string describing the machine's recipe
                    used here, for your own record.</rm:tip>
    </rm:inputString>
</rm:process>

<rm:process id="etch">
    <rm:parentId>rootMachine</rm:parentId>
    <rm:description>etching</rm:description>
    <rm:inputInt id="depth" default="10" min="0" max="100">
        <rm:description>Depth</rm:description>
        <rm:units>inches</rm:units>
    </rm:inputInt>
    <rm:inputChoice id="gas" type="string" default="1" maxOccurs="unbounded">
        <rm:description>Gas type</rm:description>
        <rm:choice id="1">Gas #1</rm:choice>
        <rm:choice id="2">Gas #2</rm:choice>
        <rm:choice id="3">Gas #3</rm:choice>
    </rm:inputChoice>
    <rm:inputChoice id="mask" type="string">
        <rm:description>Mask material</rm:description>
        <rm:choice id="1">Mask #1</rm:choice>
        <rm:choice id="2">Mask #2</rm:choice>
    </rm:inputChoice>
    <rm:inputFloat id="rf" cat="optional" min="0" max="unbounded" default="2">
        <rm:description>RF Power</rm:description>
        <rm:units>mW</rm:units>
    </rm:inputFloat>
</rm:process>

<rm:process id="dep">
    <rm:comment>You can put a parsed comment in your process description if you like.</rm:comment>
    <rm:parentId>rootMachine</rm:parentId>
    <rm:description>deposition</rm:description>
    <rm:inputChoice id="material" type="string" maxOccurs="5">
        <rm:description>Deposition material</rm:description>
        <rm:choice id="1">Dep material #1</rm:choice>
```

```
            <rm:choice id="2">Dep material #2</rm:choice>
            <rm:choice id="3">Cheese</rm:choice>
        </rm:inputChoice>
        <rm:inputInt id="thickness" min="1" max="50">
            <rm:description>Deposition thickness</rm:description>
            <rm:units>microns</rm:units>
        </rm:inputInt>
    </rm:process>

    <!-- 2. CORAL Tool references (i.e. machine names) -->

    <!-- These are almost trivial. -->
    <rm:machine id="icl:ebeam">
        <rm:name>ICL E-beam evaporator</rm:name>
        </rm:machine>

    <rm:machine id="icl:endura">
        <rm:comment>This illustrates the ability to put parsed comments
            in a machine description.</rm:comment>
        <rm:name>ICL Endura</rm:name>
    </rm:machine>

    <rm:machine id="icl:centura">
        <rm:name>ICL Centura</rm:name>
    </rm:machine>

    <rm:machine id="icl:ame5000">
        <rm:name>ICL AME5000</rm:name>
    </rm:machine>

    <rm:machine id="icl:pecvd-rie">
        <rm:name>ICL PECVD-RIE</rm:name>
    </rm:machine>

    <rm:machine id="icl:plasmaquest">
        <rm:name>ICL Plasmaquest</rm:name>
    </rm:machine>

    <!-- 3. Map of CORAL Object references to a set of processes. -->

    <!-- Notice the compactness with which a map can be specified. There is only
         one map here because only 'machineType' was used when defining CORAL
         Object references above. -->
    <rm:map object="rm:machineType">
        <rm:comment>This maps machines to processes.</rm:comment>
        <rm:mapping objectId="icl:ebeam">
            <rm:processId>dep</rm:processId>
        </rm:mapping>
        <rm:mapping objectId="icl:endura">
            <rm:processId>dep</rm:processId>
        </rm:mapping>
        <rm:mapping objectId="icl:centura">
            <rm:processId>etch</rm:processId>
        </rm:mapping>
        <rm:mapping objectId="icl:ame5000">
            <rm:processId>etch</rm:processId>
        </rm:mapping>
        <rm:mapping objectId="icl:pecvd-rie">
            <rm:processId>etch</rm:processId>
            <rm:processId>dep</rm:processId>
        </rm:mapping>
        <rm:mapping objectId="icl:plasmaquest">
            <rm:processId>etch</rm:processId>
            <rm:processId>dep</rm:processId>
        </rm:mapping>
    </rm:map>
</rm:rmConfig>
```

Appendix E

Run Manager Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.jeffklann.com/rmconfig1"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns="http://www.jeffklann.com/rmconfig1"
                elementFormDefault="qualified"
                attributeFormDefault="unqualified"
                version="0.6">

<xsd:annotation>
        <xsd:documentation>
   Recipe Manager Schema v0.6 by Jeff Klann

         This schema describes a document type for defining run-data to be collected in the
        Common Object Representation for Advanced Laboratories (CORAL). This is done by defining
        an XML lab configuration and placing inside processes, CORAL tool references, and mappings
        between them. The processes are a hierarchy of objects that define the run-data to be
        collected, and the CORAL tool references and mappings tie those processes to machines
        in the database.

        The general form of a configuration is as follows:

        <rmconfig>
                <process id="[id]">*
                        <description>[description]</description>
                        <tip>[description]</tip>?
                        <some_input_type id="[id]"
                           other_options_depending_on_input_type="[various]">*
                                <description>[description]</description>
                                <tip>[description]</tip>?
                                <units>[description]</units>?
                        </some_input_type>
                </process>

                <machine id="string">*
                        <name>[name]</name>
                        <description>[description]</description>
                </machine>

                [other CORAL tool types here]

                <map object="[derivitive_of_coral_object_type]">*
                        <mapping objectId="[id]">*
                                <processId>[id]</processId>*
                        </mapping>
                </map>

        </rmconfig>

        This, of course, leaves out much detail. There are several kinds of inputs that can
```

```
        appear in a process description, and several types of CORAL tools that can be
        referenced (and the set is extensible, of course!). Comments are allowed, and some
        optional components have been left off the above sketch. But it should give you an
        idea of the Schema's design. Read the details of the Schema for more information;
        documentation is embdedded. Also, a sample instance document is included.

        The schema is flexible enough to allow an extensible set of CORAL tools; a
        multiple-inheritance hierarchy of processes with repeatable inputs of strings,
        numbers, and multiple-choices; and an arbitrary number of mappings between them.
        XPaths are used throughout to do whatever sanity checking Schema 1.0 allows.
        Further, the 'venetian-blind' design was used for this schema, showing off the
        power of using type libraries to build a schema, rather than less-powerful model
        groups or the Russian Doll design.

        One other important note: many types have 'ids', 'descriptions' and 'tips'. Ids are
        only used internally for cross-referencing in the XML, and hence should be short -
        they're the programmer's identifier. Descriptions are what are intended to be shown
        to the user when they want to know what this object is, and therefore should be short
        but not unintelligeble to the average human user. Finally, a tip is extra information
        a confused user might want about how to use the type - along the lines of a tooltip.
        </xsd:documentation>
</xsd:annotation>

        <!-- ..................................................... -->
        <!-- ................. simple type elements................. -->
        <!-- ..................................................... -->

        <!-- Extensions of strings: 'description' and 'units' collapse whitespace (note
                that 'units' is defined separately from a 'description' type so it's
                easy to restrict it to only certain types of units in the future); 'id'
                only allows basic alphanumeric characters and no spaces (so it can be
                used in xsd:list); unboundableNonZeroIntType is a string which represents
                a nonzero integer but could also contain the string 'unbounded',
                representing an infinite number (this is a Schema convention). -->
        <xsd:simpleType name="descriptionType" id="descriptionType">
                <xsd:restriction base="xsd:string">
                        <xsd:whiteSpace value="collapse"/>
                </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType name="unitsType" id="unitsType">
                <xsd:restriction base="xsd:string">
                        <xsd:whiteSpace value="collapse"/>
                </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType name="idType">
          <xsd:restriction base="xsd:string">
                 <xsd:maxLength value="32"/>
                 <xsd:whiteSpace value="replace"/>
                 <xsd:pattern value="[0-9a-zA-Z_!-.:]*"/>
          </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType name="unboundableNonZeroIntType">
          <xsd:restriction base="xsd:string">
                 <xsd:maxLength value="32"/>
                 <xsd:whiteSpace value="replace"/>
                 <xsd:pattern value="([1-9][0-9]*)|unbounded"/>
          </xsd:restriction>
        </xsd:simpleType>

        <!-- 'Category' defines categories an input type can belong to. Currently I've hard-coded
                this to the category 'optional', which means the input does not need to be entered.
                However, it is easy to add more categories, or generalize it so the configuration
                designer can input whatever category they like, or even base it on a hierarchy of
                objects in the configuration instance which define categories - which seems
                overkill for our purposes. Note: this is a 'list' - Castor doesn't support real
                lists currently, and I didn't feel like having multiple elements rather than a
                single attribute. -->
        <xsd:simpleType name="inputCategoryListType">
```

```
        <xsd:restriction base="xsd:string">
                <xsd:pattern value="(optional( )*)*"/>
        </xsd:restriction>
</xsd:simpleType>

<!-- ..................................................... -->
<!-- ..............  complex type elements.............. -->
<!-- ..................................................... -->

<!-- 1. Process definitions-->

<!-- A process derives from some set of parent processes and contains a description and
        any number of inputs, which define the run-data to be collected for this process.
        Note that I've hard coded the input-type set here, but this is the cleanest way
        of representing it in an instance (otherwise you'd have to write
        <input xsi:type="inputStringType"> in an instance, rather than just <inputString>).
        If a future designer of this Schema might want to add input types but not have
        access to this section of the Schema (which doesn't make much sense), they
        shouldn't be hard-coded here, despite the added clarity in instance documents. -->
<xsd:complexType name="processType">
        <xsd:sequence>
                <xsd:element name="comment" type="descriptionType"
                        minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="parentId" type="idType"
                        minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="description" type="descriptionType" />
                <xsd:choice minOccurs="0" maxOccurs="unbounded">
                        <!-- Note: I enumerate the input types explicitly below to make
                        instances clearer. It'd be perfectly reasonable to do it in the
                        more general way I did machine. This method does allow me to set
                        up some nice XPath constraints on the choice type, however. -->
                        <xsd:element name="inputInt" type="inputTupleIntType"/>
                        <xsd:element name="inputFloat" type="inputTupleFloatType"/>
                        <xsd:element name="inputString" type="inputTupleStringType"/>
                        <xsd:element name="inputChoice" type="inputTupleChoiceType">
                                <xsd:unique name="choiceIdUnique">
                                        <xsd:selector xpath="choice"/>
                                        <xsd:field xpath="@id"/>
                                </xsd:unique>
                                <xsd:key name="choiceIdKey">
                                        <xsd:selector xpath="choice"/>
                                        <xsd:field xpath="@id"/>
                                </xsd:key>
                        </xsd:element>
                </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="id" type="idType" use="required" />
</xsd:complexType>

 <!-- 1.1. Input elements in a process description -->

 <!-- The following elements define the three input types: string, int, and choice.
        Each allows setting various special attributes, like a default value and
        possibly a min/max value. They all derive from a base input type, which
        require a description and id, and allows you to add a category, tip, units,
        and number of occurrences (a configuration designer might want a user to be
        able to repeat input some number of times, i.e. 'what gas was used in this
        process?' - several gases could have been used). Note that I allow entering
        'units', because the general best-practices for Schema is to force whatever
        might be needed to be entered explicitly; even though it would seem alright
        to just let this be part of the description for this implementation, as one
        never knows what the future holds. -->

<xsd:complexType name="inputTupleBaseType" abstract="true">
        <xsd:sequence>
                <xsd:element name="description" type="descriptionType" />
                <xsd:element name="tip" type="descriptionType" minOccurs="0" />
                <xsd:element name="units" type="unitsType" minOccurs="0" />
```

```
        </xsd:sequence>
        <xsd:attribute name="id" type="idType" use="required" />
        <xsd:attribute name="cat" type="inputCategoryListType" use="optional"/>
        <xsd:attribute name="minOccurs" type="xsd:int" default="1" />
        <xsd:attribute name="maxOccurs" type="unboundableNonZeroIntType" />
</xsd:complexType>

 <xsd:complexType name="inputTupleIntType">
        <xsd:complexContent>
                <xsd:extension base="inputTupleBaseType">
                        <xsd:attribute name="default" type="xsd:int" use="optional" />
                        <xsd:attribute name="min" type="xsd:int" />
                        <xsd:attribute name="max" type="unboundableNonZeroIntType" />
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="inputTupleFloatType">
        <xsd:complexContent>
                <xsd:extension base="inputTupleBaseType">
                        <xsd:attribute name="default" type="xsd:float" use="optional" />
                        <xsd:attribute name="min" type="xsd:int" />
                        <xsd:attribute name="max" type="unboundableNonZeroIntType" />
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="inputTupleStringType">
        <xsd:complexContent>
                <xsd:extension base="inputTupleBaseType">
                        <xsd:attribute name="default" type="xsd:string" use="optional" />
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="inputTupleChoiceType">
        <xsd:complexContent>
                <xsd:extension base="inputTupleBaseType">
                        <xsd:sequence>
                                <xsd:element name="choice" type="choiceTupleType"
                                        minOccurs="0" maxOccurs="unbounded" />
                        </xsd:sequence>
                        <xsd:attribute name="default" type="xsd:string" use="optional" />
                        <xsd:attribute name="type" type="xsd:string" use="required" />
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<!-- 1.2. Multiple choice element type. -->
<xsd:complexType name="choiceTupleType">
        <xsd:simpleContent>
                <xsd:extension base="descriptionType">
                        <xsd:attribute name="id" type="xsd:integer" use="required" />
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>

<!-- 2. CORAL tool references (for mapping between CORAL tools and processes) -->

<!-- The following types allow you to create references to CORAL Tools. These
        references then allow the configuration designer to map processes to
        these references. Currently the only supported type of object reference
        is 'machine'. Note that I've gone with the more general approach here (as
        opposed to input types): you have to explicitly use xsi:type to specify a
        CORAL tool type in the instance. This was done because: a) it doesn't make
        the instance much uglier, and b) it allows a clean separation of CORAL
        tool-reference types from the rest of the Schema, should the Schema split
        into multiple files down the road. -->
```

```
<!-- Note: this is the only occurence of a model group in the Schema. The reason
        I used it here is this: I want to separate CORAL Tool references from the
        rest of the Schema, because Stanford seemed interested in keeping this
        portion as a separate object library from the main configuration. However,
        Castor doesn't currently supported instance typing (e.g. xsi:type="machineType"
        attribute in CORAL Tools), so I use a model group to specify the valid CORALTools
        here, separately maintainable from the instantiation of that group further down. -->
<xsd:group name="CORALToolGroup">
        <xsd:choice>
                <xsd:element name="machine" type="machineType">
                        <xsd:key name="machineIdKey">
                                <xsd:selector xpath="."/>
                                <xsd:field xpath="@id"/>
                          </xsd:key>
                </xsd:element>
                <xsd:element name="instruction" type="instructionType">
                        <xsd:key name="instructionIdKey">
                                <xsd:selector xpath="."/>
                                <xsd:field xpath="@id"/>
                          </xsd:key>
                </xsd:element>
        </xsd:choice>
</xsd:group>

<xsd:complexType name="CORALToolType" abstract="true">
        <xsd:sequence>
                <xsd:element name="comment" type="descriptionType"
                        minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="name" type="descriptionType"
                        minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="id" type="idType" use="required" />
</xsd:complexType>

<xsd:complexType name="machineType">
        <xsd:complexContent>
                <xsd:extension base="CORALToolType">
                        <xsd:sequence>
                                <xsd:element name="description"
                                        type="descriptionType" minOccurs="0" />
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<!-- NOTE: this is only here as an example to show the power of derivation! -->
<xsd:complexType name="instructionType">
        <xsd:complexContent>
                <xsd:extension base="CORALToolType">
                        <xsd:sequence>
                                <xsd:element name="classmateCount" type="xsd:int" />
                                <xsd:element name="courseName" type="xsd:string" />
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<!-- 3. Mapping between CORAL tools and process descriptions -->

<!-- The 'map' is a very condensed piece of XML that maps CORAL tool references to
     processes. 'map's are just pairs of <mapping> which have an id for a CORAL tool
     and a list of ids corresponding to the processes they perform. Properly, each CORAL
     object should only appear once. The map is tied to a particular CORAL tool type
     (via the 'object' attribute), so multiple maps should be used for multiple CORAL
     object types. -->

<xsd:complexType name="mapTupleType">
```

```
        <xsd:sequence>
                <xsd:element name="processId" type="idType" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="objectId" type="idType" use="required" />
</xsd:complexType>

<xsd:complexType name="mapType">
        <xsd:sequence>
                <xsd:element name="comment" type="descriptionType"
                        minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="mapping" type="mapTupleType"
                        minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="object" type="xsd:QName" use="required" />
</xsd:complexType>

<!-- 4. RM config -->

<!-- The root element, which contains any number of processes,
        CORAL tool references, and maps, in that order. I chose
        this ordering because it ought to be easier to parse a map
        after the other pieces have been parsed, if SAX is used. -->
<xsd:complexType name="rmConfigType">
  <xsd:sequence>
        <xsd:element name="process" type="processType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:group ref="CORALToolGroup" minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="map" type="mapType" minOccurs="0" maxOccurs="unbounded">
                <xsd:unique name="objectIdUnique">
                        <xsd:selector xpath="mapping"/>
                        <xsd:field xpath="@objectId"/>
                </xsd:unique>
        </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="version" type="xsd:string" use="required" />
  <xsd:attribute name="versionSchema" type="xsd:string" use="required" />
        <!-- Version reflects an internal version number for the Schema, whereas
                versionSchema reflects the minimum compatible schema version the
                configuration is compatible with. -->
</xsd:complexType>

<!-- 5. Run data config -->

<!-- Here we define xml structures for passing around information on run-data.
        It'd be perfectly fine to define this in Java and a Castor XML mapping
        file, since run data never goes in the instance anyway, but this is a
        beautiful way of keeping all type information inside the Schema. -->
<!-- Implementation note: I wanted to supply default values for required
        parameters, which forces me to declare them as 'optional' in the
        Schema (which makes sense, I suppose). So parameters with default
        values should be considered mandatory, except in the case of the
        id fields which are filled in by the data handler in Java. -->
<!-- These fields reflect what is stored in the database, and between
        examining some sample saved data and reading the JavaDoc for the
        code which translates between the configuration and the rundata,
        most of these fields are straightforward to understand and are analogs of
        what is documented in rmConfig. Hence they are not documented in detail here. -->
<xsd:complexType name="rmRunDataType">
  <xsd:sequence>
        <xsd:element name="element" type="runDataTupleType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="process" type="xsd:string" default="dummy" />
  <xsd:attribute name="version" type="xsd:string" default="-1" />
        <!-- ^ a version number -->
  <xsd:attribute name="versionTimestamp" type="xsd:string"/>
        <!-- ^ a version timestamp string -->
  <xsd:attribute name="userId" type="xsd:string" default="unknown" />
  <xsd:attribute name="CORALToolId" type="xsd:string" default="unknown" />
```

```
  <xsd:attribute name="viewLock" type="xsd:string" />
        <!-- ^ user currently viewing this data -->
  <xsd:attribute name="creationTimestamp" type="xsd:string"/>
  <xsd:attribute name="timestamp" type="xsd:string" />
        <!-- ^ it's be nice to use xsd:data and xsd:time, but jdo doesn't
                support them, so we have to keep it simple. -->
  <xsd:attribute name="transactionId" type="xsd:long" default="-1"/>
        <!-- ^ unique id, filled in by the server on create. -->
  <xsd:attribute name="committed" type="xsd:boolean" default="false" />
        <!-- ^ a transaction id to be assigned when committed. -->
</xsd:complexType>

<xsd:complexType name="runDataTupleType">
  <xsd:sequence>
    <xsd:element name="key" type="xsd:string" default="dummy"/>
    <xsd:element name="stringValue" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <!-- ^ string storage field for inputted data -->
    <xsd:element name="intValue" type="xsd:int" minOccurs="0" maxOccurs="1"/>
        <!-- ^ int storage field for inputted data -->
    <xsd:element name="units" type="unitsType" minOccurs="0"/>
    <xsd:element name="fieldType" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <!-- ^ i.e. type of input tuple represented here -->
    <xsd:element name="fieldSubtype" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <!-- ^ used by choice tuples where a subtype was def'd -->
    </xsd:sequence>
  <xsd:attribute name="transactionId" type="xsd:long" default="-1"/>
        <!-- ^ the transaction id must be duplicated in the tuples for
                implementation-dependent reasons-->
  <xsd:attribute name="elementId" type="xsd:long" default="-1"/>
        <!-- ^ a unique id for this element, to be filled in on create by the server. -->
</xsd:complexType>

<!-- 6. Another non-user type, defined for internal consumption. Describes a container
        for global objects in the database. -->
<xsd:complexType name="rmGlobalContainerType">
  <xsd:sequence>
        <xsd:element name="key" type="xsd:string" default="dummy"/>
        <xsd:element name="stringValue" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <xsd:element name="intValue" type="xsd:int" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="timestamp" type="xsd:string" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>


<!-- instantiation: the only time we actually create an element in the Schema's
        global section, showing off the power of the venetian blind design. -->
<xsd:element name="rmConfig" type="rmConfigType">
  <xsd:unique name="processIdUnique">
         <xsd:selector xpath="process"/>
         <xsd:field xpath="@id"/>
  </xsd:unique>
  <xsd:key name="processIdKey">
         <xsd:selector xpath="process"/>
         <xsd:field xpath="@id"/>
  </xsd:key>
  <xsd:unique name="mapObjectTypeUnique">
                <xsd:selector xpath="map"/>
                <xsd:field xpath="@object"/>
        </xsd:unique>
</xsd:element>

<!-- we must also instantiate a database types (even though you shouldn't use
        them in an instance) so it can be used when we send data across the wire.
        I don't set up any xpath constraints here, because I don't want to take
        the time to validate when sending across the wire. This type is never written
        by a human. -->
<xsd:element name="rmRunData" type="rmRunDataType"/>
<xsd:element name="rmGlobalContainer" type="rmGlobalContainerType"/>
```

```
</xsd:schema>
```

Appendix F

IDL

```
/**
This module defines the remote interactions of the Run Manager over CORBA.
**/

module labnet {
module idl {

/** An exception to raise if a null value results from a method call.
 * RmRemoteInterfaceImpl always sets the error code to 0.
 */
exception NullReturnException {
        string description;
        short errorCode;
};

/** An exception to raise if any error occurs on the server.
 * These errors are re-thrown RmExceptions from the server.
 * RmRemoteInterfaceImpl sets the error code to match that
 * of the RmException which threw it.
 */
exception ServerErrorException {
        string description;
        short errorCode;
};

/* We need this because there is no unsized array of strings in idl. */
typedef sequence<string> stringVector;

interface RmRemoteInterface {
        /*
         * Note that some of these methods cannot actually throw NullReturnExceptions
         * (as they return primitive types or void), and others will never throw
         * ServerErrorExceptions (as the underlying methods do not throw exceptions)
         * in the Java implementation. However, they are included in the 'raises' clause
         * to simplify the interface spec. Clients should be prepared to catch these
         * exceptions in case a different implementation lies underneath the idl.
         */
```

```
********************* Retrieving recipe information ***********************
**                                                                          40
 * Returns the string name of the loaded container for the configuration,
 * or <none> if no configuration is loaded.
 * Use container names in Util to determine whether the returned string
 * is last-known-good or current.
 * @return String name of the loaded container for the configuration
 * @exception NullReturnException Thrown if the return value is null.
 * @exception ServerErrorException Thrown if an error occurs in processing the
 *       request.
 **
string getConfigurationContainerName()                                    50
        raises (NullReturnException, ServerErrorException);

**
 * Returns the instance version number.
 * @return String containing version number.
 * @exception NullReturnException Thrown if the return value is null.
 * @exception ServerErrorException Thrown if an error occurs in processing the
 *       request.
 **
string getInstanceVersion()                                               60
        raises (NullReturnException, ServerErrorException);

**
 * Returns the instance version timestamp (a java.sql.Timestamp converted to a string).
 * @return String containing version number (converted from a java.sql.Timestamp).
 * @exception NullReturnException Thrown if the return value is null.
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
string getInstanceVersionTimestamp()
        raises (NullReturnException, ServerErrorException);               70

**
 * Returns set of CORAL Tool Reference ids which are referenced in some map.
 * @return Array of strings
 * @exception NullReturnException Thrown if the return value is null (won't actually be
 *       thrown here)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
stringVector getCORALToolIds()
        raises (NullReturnException, ServerErrorException);               80

**
 * Returns set of CORAL Tool Reference names which are referenced in some map.
 * If no name is specified for some tool, the id is returned.
 * @return Array of strings
 * @exception NullReturnException Thrown if the return value is null (won't actually be
 *       thrown here)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
 stringVector getCORALToolNames()                                         90
        raises (NullReturnException, ServerErrorException);
```

```
**
 * Returns the CORAL Tool description with given id (as xml).
 * @param id An id string for a CORAL Tool.
 * @return CORAL Object description (not the object group) as xml.
 * @exception NullReturnException Thrown if the return value is null (i.e. no CORAL
 *       Tool exists with given id)
 * @exception ServerErrorException Thrown if an error occurs in processing the request
 *       (won't actually be thrown here)                                          100
 **
string getCORALToolById(in string id)
        raises (NullReturnException, ServerErrorException);

**
 * Returns the set of process ids a CORAL Tool can perform.
 * @param id An id (not the tool name) for a CORAL Tool
 * @return An array of process ids
 * @exception NullReturnException Thrown if the return value is null (i.e. the given
 *       id cannot perform any processes)                                         110
 * @exception ServerErrorException Thrown if an error occurs in processing
 *       the request (won't actually be thrown here)
 *
 **
stringVector getProcessesByCORALToolId(in string id)
        raises (NullReturnException, ServerErrorException);

**
 * Returns the set of process descriptions a CORAL Object can perform.
 * @param id An id string for a CORAL Object                                      120
 * @return An array of process descriptions.
 * @exception NullReturnException Thrown if the return value is null (i.e. the given
 *       id has no associated process descriptions)
 * @exception ServerErrorException Thrown if an error occurs in processing
 *       the request (won't actually be thrown here)
 *
 **
stringVector getProcessDescriptionsByCORALToolId(in string id)
        raises (NullReturnException, ServerErrorException);
                                                                                  130
**
 * Returns the 'augmented ' process description, given a process id as xml
 * (i.e. a version of the process with all the contents of its parent included)
 * @param Any valid process id string
 * @return An augmented process description in xml.
 * @exception NullReturnException Thrown if the return value is null (i.e. no
 *       process exists with the given id)
 * @exception ServerErrorException Thrown if an error occurs in processing
 *       the request (won 't actually be thrown here)
 **                                                                               140
string getAugmentedProcessById(in string id)
        raises (NullReturnException, ServerErrorException);

*************************** Storing run-data ***************************

** The following methods allow the client to store run-data. The standard
```

```
 * use-case is as follows: run-data is created, at some time later it is
 * updated, and finally it is either committed (so it can no longer be
 * updated) or canceled. In case the client closes his or her application
 * in-between steps 1 and 2, a method is provided to retrieve uncommitted
 * run-data by the user 's login name. A user may have only one piece of
 * uncommitted run-data in existence at a time for a single CORAL Tool.
 **

**
 * Sends a run-data element back to the server for saving, given that
 * RmRunData object as a string. Stores this object as 'uncommitted ', so
 * it can still be modified or canceled. The create fails if this user has
 * any other open uncommitted rundata.
 * @param rundata An xml string representing an RmRunData object
 * @return Assigned tid on success, -1 otherwise
 * @exception NullReturnException Thrown if the return value is null
 *      (won 't ever actually be thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing
 *      the request.
 **
long long createRunData(in string rundata)
        raises (NullReturnException, ServerErrorException);

**
 * Sends a run-data element back to the server for update, given that
 * RmRunData object as a string. Replaces an 'uncommitted ' piece of
 * rundata in the database, and fails if this piece of rundata hasn 't
 * been create()d or is already committed. The rundata passed to this
 * method must be fully filled in with valid transaction id and
 * element ids (i.e. it must have been loaded from the db, not created manually).
 * @param rundata An xml string representing an RmRunData object
 * @exception NullReturnException Thrown if the return value is null (won 't
 *      actually be thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
void updateRunData(in string rundata)
        raises (NullReturnException, ServerErrorException);

**
 * Commits an existing piece of rundata, given its transaction id. After this is done,
 * it can no longer be canceled or updated. It is now an official piece of accounting
 * information. The transaction id can be found in a loaded committed piece of rundata
 * in using getTransactionId().
 * @param tid A transaction number assigned by createRunData to an RmRunData object.
 * @exception NullReturnException Thrown if the return value is null (won 't actually be
 *      thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
void commitRunData(in long long tid)
        raises (NullReturnException, ServerErrorException);

**
 * Sets the viewlock on an existing piece of rundata. Will fail if it is already set.
 * @param tid A transaction number assigned by createRunData to an RmRunData object.
```

```
 * @param userId User name to lock with.
 * @exception NullReturnException Thrown if the return value is null (won 't actually
 *       be thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
void setRunDataViewLock(in long long tid, in string userId)
        raises (NullReturnException, ServerErrorException);

**
 * Releases the viewlock on an existing piece of rundata. Will fail if the supplied
 * userid is not the owner of the viewlock.
 * @param tid A transaction number assigned by createRunData to an RmRunData object.
 * @param userId User name that should currently have the lock.
 * @exception NullReturnException Thrown if the return value is null (won 't actually
 *       be thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
void releaseRunDataViewLock(in long long tid, in string userId)
        raises (NullReturnException, ServerErrorException);

**
 * Cancels an existing piece of rundata given its tid. This removes the rundata from the
 * database. Will fail if this rundata is already committed (no deletion will occur in the
 * failure case).
 * @param tid A transaction number assigned by createRunData to an RmRunData object.
 * @return true on success, false otherwise.
 * @exception NullReturnException Thrown if the return value is null (won 't actually be
 *       thrown by this call)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
void cancelRunData(in long long tid) raises (NullReturnException, ServerErrorException);

**
 * Retrieves the single piece of uncommitted rundata any user is allowed to have for a
 * given Coral tool.
 * @param userid A user 's id for which to retrieve rundata.
 * @return A string representing a rundata object as xml.
 * @exception NullReturnException Thrown if the return value is null (i.e. no uncommitted
 *       rundata exists)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
 **
string getUncommittedRunDataByUserAndToolId(in string userid, in string objectid)
        raises (NullReturnException, ServerErrorException);


*************************** Administrative tools ************************

**
 * Retrieves the contents of an RmGlobalContainer object as xml, given a key name.
 * @param id A key name.
 * @return An RmGlobalContainer object as xml.
 * @exception NullReturnException Thrown if the return value is null (i.e. no global
 *       container with this key name actually exists)
 * @exception ServerErrorException Thrown if an error occurs in processing the request.
```

```
        **
        string getGlobalContainerById(in string id)
                raises (NullReturnException, ServerErrorException);

        **
         * Stores an RmGlobalContainer object unconditionally - the existing object is
         * replaced if an object with this object 's key already exists.
         * @param container An RmGlobalContainer object as an xml string.
         * @exception NullReturnException Thrown if the return value is null (won 't
         *      actually be thrown by this call)
         * @exception ServerErrorException Thrown if an error occurs in processing the request.
         **
        void saveGlobalContainer(in string container)
                raises (NullReturnException, ServerErrorException);

        **
         * Retrieves a run-data object back from the server, given a transaction id.
         * @param tid A transaction id
         * @return a string representing a rundata object as xml, or null
         * @exception NullReturnException Thrown if the return value is null (i.e. no
         *      run-data exists with this transaction id)
         * @exception ServerErrorException Thrown if an error occurs in processing the request.
         **
        string getRunDataByTransactionId(in long long tid)
                raises (NullReturnException, ServerErrorException);

        **
         * Causes the server to reinitialize, thus reloading all of its
         * configuration and making a new connection to the database.
         * @exception NullReturnException Thrown if the return value is null (won 't
         *      actually be thrown by this call)
         * @exception ServerErrorException Thrown if an error occurs in processing the request.
         **
        void cycleServer()
                raises (NullReturnException, ServerErrorException);
};
};
};
```

Bibliography

[1] Apache Software Foundation. *PHP: General Information - Manual*, August 2002. *http://www.php.net/manual/en/faq.general.php*.

[2] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, Boston, Massachusetts, 2001.

[3] Collaborative Software Laboratory. *Swiki Swiki*, August 2002. *http://minnow.cc.gatech.edu/swiki*.

[4] Vicky Diadiuk. Personal interviews, February-August 2002.

[5] Exolab, Inc. *The Castor Project*, June 2002. *http://www.castor.org*.

[6] INT Media Group, Inc. *Webopedia*, August 2002. *http://www.webopedia.coml*.

[7] Microsystems Technology Laboratories. MTL annual report. Technical report, Massachusetts Institute of Technology, June 2001.

[8] Ike Lin. Personal interviews, February-August 2002.

[9] Thomas Lohman. Personal interviews, February-August 2002.

[10] Brett McLaughlin. *Java & XML*. O'Reilly, Cambridge, Massachusetts, 2001.

[11] Bill Murray. Personal interviews, February-August 2002.

[12] National Telecommunications and Information Administration. *Federal Standard 1037C: Glossary of Telecommunication Terms*, August 1996. *http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm*.

[13] Perttu Sliden. *CORBA*, August 2002. *http://www.student.oulu.fi/~psliden/corba.html*.

[14] PROMIS Software. *PRI Automation, Inc.*, February 2002. *http://www.pria.com/products/fms/pr_promis.htm*.

[15] John Shott. Personal interviews, February-August 2002.

[16] John Shott, Bill Murray, and Mike Bell. *CORAL Usage Guide*. Stanford Nanofabrication Facility, 2000. *http://snf.stanford.edu/About/Overview.html*.

[17] Slava Pestov et all. *jEdit - Open Source programmer's text editor*, June 2002. *http://www.jedit.org*.

[18] Stanford Nanofabrication Facility. *About SNF*, February 2002. *http://snf.stanford.edu/About/Overview.html*.

[19] TechTarget Network. *searchVB: The VB Specific Search Engine*, August 2002. *http://www.searchvb.com*.

[20] Chelsea Valentine, Lucinda Dykes, and Ed Tittel. *XML Schemas*. Sybex, San Francisco, California, 2001.