RESULT SPECIFICATION AND MODEL CONNECTION IN THE MÖBIUS MODELING FRAMEWORK

BY

AMY LOU CHRISTENSEN

B.S., Purdue University, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

ABSTRACT

Möbius is an extensible framework for system modeling. Models can be designed using several modeling languages, which are called *formalisms*. Multiple solution methods, including discrete-event simulation and state-space solvers, are available. Some systems are modeled with a combination of several models. There are two methodologies for supporting communication between models: composed models communicate via shared state variables, and connected models communicate by passing solution values, which are called *results*. We will present the Möbius platform for supporting model connection. First, we explain the database used to store results. Next, we propose a software architecture for building a broad class of connection formalisms and solvers. We conclude with a connected model case study using the Möbius modeling tool.

To my family and friends.

ACKNOWLEDGMENTS

I would like to acknowledge all of the individuals who provided technical guidance, advice, finances, and encouragement during this endeavor. First, I would like to thank my adviser, Professor William H. Sanders, for inviting me to join the Möbius research group. Thank you for the patience, technical guidance, and mentoring that you provided throughout the project. Thanks to my teammates Dan Deavours, David Daly, Jay Doyle, Patrick Webster, and Graham Clark for their help and advice. Thanks also to Jenny Applequist for editing this document and helping me through the thesis approval process. I would also like to thank the Motorola Validation Center and the National Science Foundation (contract number EIA 99-75019) for their financial support of the Möbius research project.

I would not have survived the trials of a thesis project without the support of my family and friends. First, I would like to thank my parents, Bob and Anna Ruth, for their encouragement. I extend many thanks to my two best friends, Tate and Samson, for lending a sympathetic ear when needed. Thanks also to Lee for her many graduate school stories and insights. I would also like to thank my friends in Colorado for providing a persistent, motivational force to guide me back to Illinois to finish my master's degree. I extend my thanks to everyone who helped me to accomplish this goal.

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1. System Modeling Overview

Demand for computer systems and communication networks is increasing. Considerable cost is associated with the creation of such systems. Developers are concerned about design cycle time, performance, and reliability. Techniques that allow a developer to estimate performance and reliability measures based on a model (description) of the system can shorten the design cycle by identifying problems before building the system. Modeling can also decrease system component costs by allowing designers to choose equipment that will meet rather than exceed the performance and reliability specification. A designer chooses a set of measures to encapsulate the performance and reliability of the system, decides how to represent (abstract) the system as a model, and chooses the techniques that will be used to solve that model for the needed measures.

Measures of performance or reliability are determined by the purpose of the system. For example, a performance measure for a redundant array of inexpensive disks (RAID) system may be the time needed to retrieve a block of data. The mean, variance, and distribution of this measure in steady state or at various points in time could be useful in choosing the type of disks to use and the algorithms and hardware used to connect them. The probability that the system has enough disks in operation that the system can continue to function is a potential measure of reliability.

System abstraction is heavily dependent upon the measures under study and the intuition of the designer. It is critical to identify the parts of the system that are relevant to the necessary measures and to represent accurately the behavior of these components, which may be physical or algorithmic, in one or more modeling languages. Some popular modeling languages are queueing networks, stochastic activity networks (SANs), Petri nets, and fault trees. A physical abstraction could be a data bus, memory chip, or operating system model, while operations such as routing data packets in a network could be specified at an algorithmic level.

Solving the system model is the final step in the modeling process. Solver selection is dictated by the modeling language chosen, the specified measures, and the accuracy required. Analytic (mathematical) techniques are very fast and accurate but can only be used in a few modeling languages. Simulation-based techniques can solve a broader class of models, but are typically slower and less able to capture the effects of rare events in the system, resulting in greater error. Building and instrumenting a prototype of the system is the most accurate but expensive option.

1.2. Möbius Overview

Möbius is an extensible software tool for modeling stochastic, discrete-event systems [1], [2]. Möbius utilizes software design techniques that promote easy integration of new modeling languages, solution techniques and measures. A primary goal is to provide a rich set of generic resources and well-defined programmer interfaces to those resources to create a complete development environment for new modeling ideas. The abilities to define and vary parameters, print, or save solutions into a database are generic contributions that empower researchers by decreasing the time needed to transform a modeling idea into a usable format. Another goal is to provide a framework that can be used to specify and solve a complex system in several parts, each of which may employ a different modeling language or solution technique. Next, we will define the key elements of the Möbius framework.

Several components are needed to create a completely specified Möbius model, referred to as a *solvable model*. The components are illustrated in Figure 1. First, models are created using one of the available modeling languages, called *formalisms*. *Atomic model formalisms* are at the lowest level, involving only one model type. Models are described using a graphical or textual formalism editor that provides primitives that are meaningful in that formalism. Primitives that require parameter values may be set to constant values or defined with global variables (identified by symbolic names) that will be bound to a value at a later time. All atomic formalism primitives are automatically translated into a generic language of state variables and activities that describe how the state changes. In addition to atomic model formalisms, there are two other model families that provide facilities for coupling models. *Composed model formalisms* introduce the ability to join models through the

sharing of state variables. *Connection model formalisms* use solutions calculated from one solvable model as inputs into other solvable models. After a set of (atomic, composed, connected) models has been defined, measures are defined using one of the existing *reward models*. Next, a *study* is created that will bind all global variables to a constant value or a range of values. The study will instruct Möbius to solve the models once for every permutation of the global variables. Finally, a solution technique (*solver*) is chosen. A solvable model can be executed, generating solutions for every permutation of global variable values specified by the study. Information provided in a solution is controlled by the settings of the reward variables and the capabilities of the solver. A user may elect to turn off some of the reward variable calculations, such as the distribution of a reward variable, to speed up the solution process. Solvers generally produce some generic information about the models as well as solutions to each of the reward variables defined.

Several solvers may be able to solve a given formalism, and a given solver may be able to solve models written in several formalisms. Such reusability is a key feature of the tool and is accomplished via the definition of a language by which solvers and models communicate, referred to as the *abstract functional interface* [3].

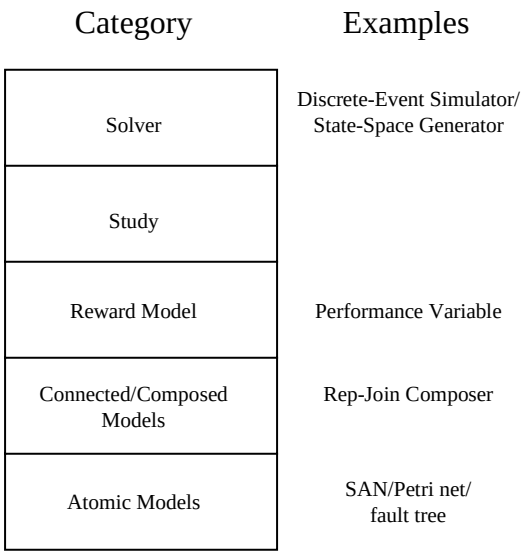


Figure 1. Components of a Möbius Solvable Model

All components of a Möbius solvable model are specified using editors in a Java-based GUI program. To increase speed, components are translated into C or C++ and compiled to the native language of the target machine before being executed. Möbius global

variables become C/C++ global variables that are initialized according to the specified study immediately before the solver executes.

1.3. Overview and Definition of Model Results

The ultimate goal of any modeling tool is to provide the user with data that gives insight into his or her model. A solvable model is executed on one or several computers, resulting in answers to the reward variables defined as well as other pertinent information about the model for every combination of global variables. This thesis project focused on creating a framework for collecting all of this information into a database and making the information available to connection model formalisms and solvers. The database, referred to as the *results database*, is explained below.

An execution of a solvable model is defined as a *run* (note that the solver may be invoked multiple times as directed by the study). A *result* is a piece of information produced by a solver or that identifies the run. Generally, we will use the term *result* in the former sense; however, the latter is important because it documents the conditions under which the solver was executed, such as the precise time, model versions, global variable values, and settings used. System modeling is an iterative process. A model is built and solved, creating output that causes the modeler to make changes and re-solve until all of his or her goals have been met. Users will likely change details of their models over time, while keeping answers from previous runs to document the design process. This creates the possibility for confusion about which model version or settings were used to produce a given answer. Keeping a detailed record of settings is also useful in connection formalisms, giving more insight into how a given number was produced.

There are three ways in which a system modeling tool can benefit from a results database. First, it serves as an archive, a permanent record of work done by the tool. Second, it makes results available in a way that promotes enhanced analysis. One example is the design of experiments editor that is currently being developed for Möbius. It looks at how reward variable solutions are affected by global variable values. This gives the user insight into model sensitivity and determines optimal values for each variable. The third and most important purpose of the results database is to serve as the foundation for model connection.

Results are stored into the database in a format that is easily retrievable for input into other models.

A connection formalism is a divide-and-conquer approach to system modeling. A single system is abstracted as several smaller models that are connected by passing results. A result calculated from one model may be used by other models. The submodels are solved in an order that allows all of the input results for a model to be valid before it is solved. There may be circular dependencies between models. This requires some form of initial guess for the values of some results, followed by an iteration process that arrives at a final answer.

There are two primary motivations for using a connected model. First, the system may require the use of several different formalisms or solvers; hence, the entire model cannot be solved as a single unit. Second, it may be more efficient to solve a single model in several pieces. State-based models are likely to become very large as a model grows, causing the memory required and solution time to increase exponentially. Solving a model in pieces eliminates this problem and allows us to solve the model faster and handle larger model sizes. We will use Möbius to solve a queuing network with these properties in a later chapter.

1.4. Existing Results Frameworks

We know of three system modeling tools that support model connection. SHARPE is a multiformalism tool that allows connection between models at parameter boundaries [4]. Solutions are retrieved using several generic built-in functions such as cdf (cumulative distribution function) that give a single number or an exponential polynomial result and its mean and variance. The meaning of this result is formalism-specific. There are also sets of functions that are specific to each formalism. An example is the “qlength” function, which returns the average queue length of a station in a product-form queueing network model. All single number results can be used in mathematical expressions. These expressions can be used in other model declarations wherever single number parameters exist. Distributions are represented as exponential polynomial functions. Some result functions return data in this format, and may be used in other model parameters that accept distributions. A loop com-

mand is provided that can cause a set of models to be solved a fixed number of times, allowing iteration for cyclic connections. It is also possible to specify an initial value for a result, used during the first iteration.

SMART is also a multiformalism modeling tool [5]. It shares our goal of creating a framework to allow easy integration of new solutions. Model solutions can be used in real, integer, or Boolean expressions that can be used to set parameters in other models. SMART developers also plan to pass distribution information as an exponential polynomial. Fixed-point iteration is supported.

POEMS is a tool for modeling complex parallel and distributed systems [6]. It is restricted to formalisms for describing application software, operating systems, and hardware architectures. POEMS combines existing tools to create a single multiformalism platform. There are restrictions on combinations of different model types. A system is specified by a multidomain dependence graph. Models, which must be of the supported model types, are nodes in the graph, and dependencies are specified by unidirectional edges. Each model type is characterized by a set of protocols that it can utilize to accept or request information. An edge is legal if the request interface of the model at the tail and the accept interface of the model at the head share a common protocol. In many cases, an extra step is required to massage the data output from one tool into a format acceptable for the next tool.

1.5. Möbius Results Framework

The remainder of this thesis will explain the Möbius platform for managing results and building connected models. We explain our database and query language design and propose connected model extensions to the abstract functional interface. These extensions are the foundation for building connected model formalisms and solvers. We also illustrate the new framework by solving a connected model using Möbius.

Each Möbius connected model comprises a set of solvable models (called *submodels*) with unidirectional arcs (*conduits*) that are used to direct results from a submodel into high-level language code blocks called *connection functions*. Inside the connection functions results may be processed, using the full features of the language, to produce numbers used to drive other submodels. These numbers are passed through conduits into the succes-

sor models by overriding global variable values defined in the model. Connected model formalisms use primitives to instantiate submodels, conduits, and connection functions. Solvers use some algorithm, such as fixed-point iteration on one or more machines, to invoke solution of the submodels and execute the connection functions to update the global variables of models that follow. Global variables can be used to set any parameterizable primitive in Möbius. This is the same strategy chosen by the SHARPE tool.

Information can be retrieved through three popular programming language interfaces to allow users and Möbius developers to write complex programs to study results. A specialized query language is also available.

2. RESULTS DATABASE

2.1. Introduction

The results database is at the core of the connection framework. We identified several key goals for this part of the project. First, the database must be accessible from anywhere inside or outside the Möbius tool. Several internal components must be able to access the database. All solvers must be able to write results into the database, and data analysis tools such as our Design of Experiments editor [7], must be able to retrieve results. Connection solvers must also be able to read results inside connection functions to compute global variable override values used in the connection process. Additionally, tools such as a database browser for viewing results or an export utility for porting data to other tools may be built as stand-alone applications.

The second requirement is flexibility. Our solution must be capable of supporting new solvers and reward models as they are added to the tool. This requires that the solution accommodate new settings and solutions data and provide a query language capable of performing searches required by future analysis tools.

The third requirement is an intuitive interface. It is unlikely that the modelers who use the tool or contribute to the development effort will have backgrounds in databases. We would therefore like the solution to minimize database jargon, so that those individuals can master, with minimal effort, the skills necessary to write, read, and query the database.

The fourth requirement is the minimization of licensing issues. Möbius is used in research as well as industry settings, so it is important that the cost for database licenses be minimal. Conversely, we would also like to take advantage of faster, more expensive databases, which our industry partners may prefer to use.

The last requirement is support for multiple platforms and machines. Although Möbius is currently supported only on the Solaris operating system, we hope to support HPUX and Linux in the near future. Our database solution must run on all platforms as well. Also, some Möbius solvers distribute a solvable model across several machines. The results database must be accessible from any machine in the network.

Our design goals presented several challenges. Möbius analysis tools and segments of the code that produce information used to identify runs are written using Java. Möbius solvers are written in C and C++. Our database must be reachable from all of these languages to meet our accessibility requirements. Next, the amount and format of data saved varies among solvable models. Each solver has its own options and solution values. Even the storage requirements for a given solvable model can change, because global variables and reward variables can be added or removed. Conventional databases like SQL were designed to save data with a rigid, unchanging format making them less amenable to our needs. The final major challenge is that of avoiding dependence upon a particular database vendor. New products will become available, and licensing agreements and fee structures can change over time. We would like to be able to switch database engines with minimal overhead. This is in line with our goal of allowing Möbius to interface to any database solution that our industry supporters would like to use. While it would be possible for us to build a database from scratch, it would not be as fast or efficient as a commercial database, and would require massive programmer resources for development and support. Our solution meets all of the specified goals.

2.2. Software Architecture

2.2.1. System diagram

We chose to build the database in two layers. The Möbius application accesses the database using the highest level (see Figure 2) The lower layer, the virtual database, provides a programmer interface that abstracts a database as a service with the ability to define C-style data structures called *templates*. Templates can be instantiated and subsequently read, written, or searched using a generic query language. The virtual database abstraction is implemented using a Java interface to the chosen database software. Our initial implementation is built on Postgre SQL [8].

The upper layer of the database is the Möbius access (MAC) layer. It uses the virtual database to build the data structures used to store Möbius data. Global variables, solver settings, and solutions such as mean and variance are defined at this level. It is important to note that this is the only entry point to the database. *All* Möbius components and stand-alone

tools use this interface. We could have met our vendor independence goals using only a single layer. We chose to implement the second layer because there is a natural breaking point between a generic database and an application that uses it for a specific purpose. The two-layer approach is a superior software design, because it makes the software easier to understand. The virtual database could also be used as a foundation for building databases for other purposes.

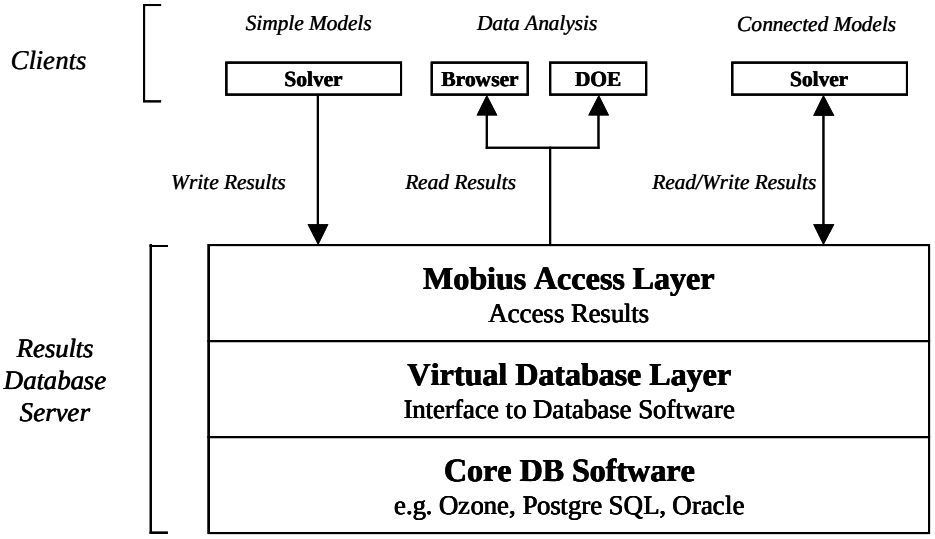


Figure 2. Result Database Architecture

2.2.2. Implementation

The results database was written in Java and C. We chose to implement both database layers using Java. A C language interface is implemented using a socket protocol between a MAC object, instantiated in the Möbius (Java) application, and the C solvers. Both C and C++ solvers use the C language MAC interface. Java code is a good choice for the database layers for several reasons. First, it is available for all of the operating system platforms that we will support. Second, it is a high-level language that is easy to use and has extensive built-in support for managing strings. Most database programming interfaces are string-based; Postgre SQL is completely string-based. Good string classes allow us to interface to the database with more concise code, improving readability and reducing bugs. We have found that most popular databases already provide a Java interface. Our Postgre SQL implementation uses the JDBC interface [9]. It is a standard database interface for SQL-

style databases. The building of command strings and queries to send to a database is not sufficiently CPU-intensive to warrant the extra effort needed to write the application in a lower-level language such as C or C++.

Our C language interface uses an ASCII-based protocol implemented on TCP/IP sockets. A MAC server thread running inside the Möbius application accepts database requests from solvable models that are executing somewhere on the network (see Figure 3). In Figure 4, the protocol is illustrated using a sequence diagram. First, a solvable model requests that a database operation be performed. A message giving the name of the needed method and values for all of the parameters is sent to the MAC server thread listening at a predefined port number. The server will execute the method and send data returned from the method back to the calling application. A byte protocol was chosen for ease of use. Socket communication will occur between different programming languages on machines that may have opposite endianness, and the nature of database queries makes the format of the data returned very irregular. Both of these factors make binary encoding more difficult. While a binary code may produce messages that are a bit smaller, socket transfers tend to be rather fast. Data buffering also makes transfer speed less dependent on size. Sockets allow distributed solver processes to send results to the MAC object in the Möbius application.

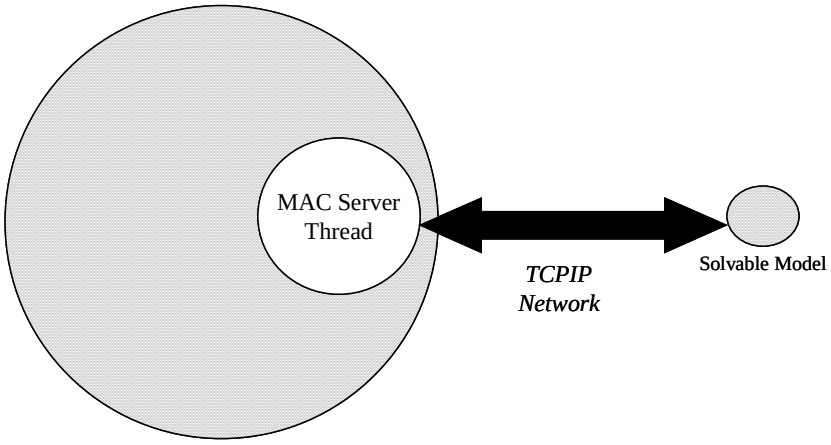


Figure 3. Möbius Process Space

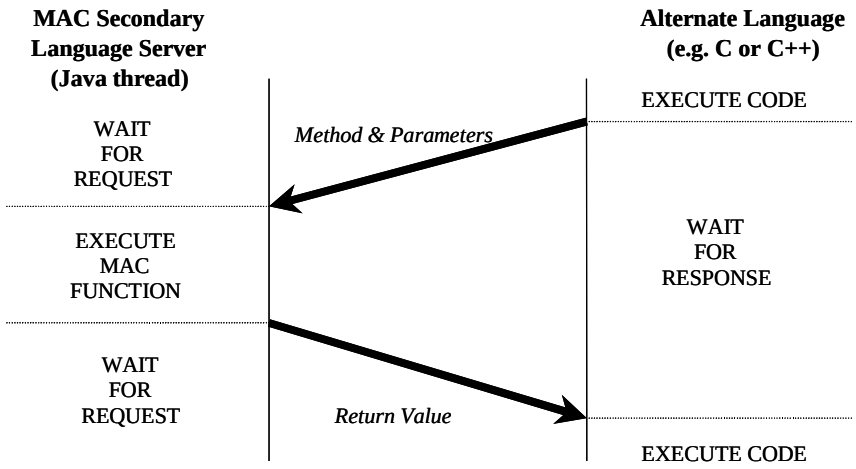


Figure 4. C Language Socket Protocol

2.3. Virtual Database Layer

The virtual database represents the essential capabilities of a database tool. While the interface and storage techniques in individual software packages vary, each one provides the user with mechanisms to create and maintain data structures and perform queries (searches) on their content. We designed the naming conventions and query language to work well with both SQL and object-oriented databases [10], [11].

A database is a persistent repository for data. It can be thought of as a set of data structure instances (*records*) that may contain different types of information. The declaration of a record is a *template*. Each template is composed of a set of name/type pairs called *fields*. A field may be of a primitive data type, such as a character, integer, or floating point number. See Table 1 for a list of primitive types. A field can also be a record, referred to as a *subrecord*. A field can be declared to be a single value or an array. A database is defined as a collection of records. A record is a collection of fields. This relationship is illustrated in Figure 5. Our abstraction is analogous to the struct construct of the C language. A struct (template) is a collection of variables that can be of a language primitive type or another struct (subrecord). A field can be addressed in several ways.

Subrecord fields can be accessed using C-style dot notation (`myrecord.mysubrecord.myfield`). Arrays are indexed using square brackets (`myrecord.myarray[5]`). Subrecord arrays can also be addressed using an associative index. An associative index allows a user to choose an array element based on the value of

some field rather than position. A simple expression using relational operators (=, <, >) contained in curly brackets compares a field to some value. An example is "`myrecord.mysubrecord{pvname='speedup'}.mean`". This refers to the mean field of the subrecord array *mysubrecord* whose field pvname is *speedup*. Note that an associative expression must be true for exactly one element of the array. An associative index can also be dereferenced with angled brackets. Dereferencing causes the element number computed for the associative index to be used in another array reference. This feature is used to conserve space for an array of variables that are defined once but have many different values. Dereferencing is used to address global variables and performance variables in the MAC layer.

Next, we will present the interface to the virtual database. Methods are categorized by function as administration, template, record, or query.

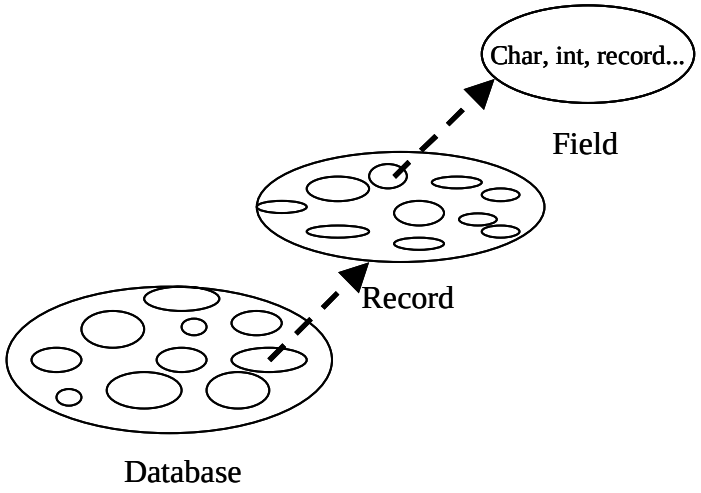


Figure 5. Generic Database Abstraction

Table 1. Primitive Data Types

Name	Description
varchar	variable length character string
int4	Four-byte whole number
int8	Eight-byte whole number
float8	Eight-byte floating point number (double)
boolean	Binary value (t or f)

2.3.1. Administration methods

Administration methods handle database operations that are not directly related to the data. High-level operations, such as creating a new database or specifying information about the location of the core database server, are defined in this group.

`Boolean create(dbname)`

Description: creates a new database with the specified name

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean delete(dbname)`

Description: permanently removes the database with the specified name

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean open(dbname)`

Description: opens a database; only one database can be open in a given virtual database object

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean close()`

Description: closes the database that is currently open

Parameters: none

Returns: false if errors are encountered else true

`Boolean copy(newdbname)`

Description: copies the contents of the currently open database into a new database with the specified name

Parameters: `string` **`newdbname`** the new database reference name

Returns: false if errors are encountered else true

`Boolean setUserLogin(username, password)`

Description: sets user login information for the core database software

Parameters: `string` **`username`** username given by the database administrator

`string` **`password`** password for account

Returns: false if errors are encountered else true

`Boolean setServerInfo(hostname, port)`

Description: sets the location of the core database server

Parameters: `string` **`hostname`** the name of the machine running the server

`int` **`port`** the port number for the server

Returns: false if errors are encountered else true

2.3.2. Template methods

Template methods control data structure definitions for the database. Templates declare the skeleton (format) of the data. A template specifies the type, name, and size of each field. A description can also be provided for use in a browser or report. To assist the user, the description can be displayed along with the data.

`Boolean createTemplate(name, description, fields)`

Description: creates a new template with the specified name

Parameters: `string` **`name`** the reference name

`string` **`description`** a short descriptive phrase used by the browser

`string` **`fields [][4]`** a two-dimensional description of the fields.

One row describes each field.

Column 0: type of field

Column 1: name of field

Column 2: size of field (zero for a single value)

Column 3: description of field

Returns: false if errors are encountered, else true

`Boolean deleteTemplate(name)`

Description: permanently removes a template and all data of that type

Parameters: `string` **`name`** the reference name

Returns: false if errors are encountered else true

`Boolean modifyTemplate(name, modifications)`

Description: modifies the structure of the template by adding, removing, or changing the size of an array

Parameters: `string` **`name`** the reference name

`string` **`modifications[][]`** a set of modifications with one row per modification. Each row contains the following:

Column 0: action (ADD, REMOVE, RESIZE)

Column 1: field name

Column 2: size (for RESIZE only)

Column 3: type (for ADD only)

Column 4: description (for ADD only)

Returns: false if errors are encountered else true

`string[][] readTemplate(name)`

Description: return a description of all of the fields

Parameters: `string` **`name`** the reference name

Returns: a list of fields in the same format used in createTemplate

2.3.3. Record methods

Record methods operate on instances of templates. Fields can be read or written after a record has been created (instantiated). An arbitrary list of fields can be accessed in a

single method call. Values are passed using a flexible container called a *DataSet*. The DataSet container is based on the ResultSet construct of the JDBC interface.

A DataSet is a collection of elements specified on a per instance basis. A name and (primitive) data type are specified for each element in the DataSet (see Figure 6). Individual elements are accessed using `readX` and `writeX` methods, where *X* is a valid primitive type (i.e., int4). A DataSet can contain multiple sets of values. Multiple sets are iterated using the `next` and previous `methods`. This capability is used by the query engine. Read and write operations use one set.

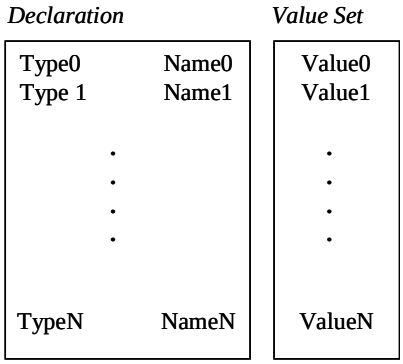


Figure 6. DataSet Container

```
foo.reset()

while( foo.next() ){
  print foo.readInt4("myint4")
  print foo.readVarchar("myvc")
}
```

Figure 7. Iterating A DataSet

Figure 7 illustrates pseudocode used to print all of the values in an example DataSet called *foo*. Foo contains two elements, an int4 named *myint* and a varchar called *myvc*. The `reset` method sets the current set to –1. The `next` method increments the set pointer and returns true until the last set has been reached. If foo contained no sets, the while loop

would not be executed, because the `next` method would return false the first time it is evaluated.

`Boolean newRecord(name, template, description)`

Description: instantiates a new record of the specified template.

Parameters: `string` **`name`** the reference name

`string` **`template`** a template name

`string` **`description`** a short descriptive phrase used by the browser

Returns: false if errors are encountered else true

`Boolean freeRecord(name)`

Description: permanently removes a record from the database

Parameters: `string` **`name`** the reference name

Returns: false if errors are encountered else true

`Boolean copyRecord(s_name, d_name)`

Description: copies the contents of a record into another (existing) record
Both records must be of the same template type.

Parameters: `string` **`s_name`** the source record

`string` **`d_name`** the destination record

Returns: false if errors are encountered else true

`Boolean lockRecord(name, islocked)`

Description: sets the read/write permission of a record

Parameters: `string` **`name`** a record name

`Boolean` **`islocked`** true for read-only, false for read/write
Read-only access prevents a record from being modified or deleted.

Returns: false if errors are encountered else true

`Boolean getLockBit(name)`

Description: returns the read/write permission of a record

Parameters: `string` **`name`** a record name

Returns: the state of `islocked` defined in `lockRecord`

`Boolean writeRecord(name, data)`

Description: write values into fields

Parameters: `string` **`name`** a record name

`DataSet` **`data`** the field name/value pairs

Returns: false if errors are encountered else true

`DataSet readRecord(name, fieldnames)`

Description: read field values

Parameters: `string` **`name`** the reference name

`string[]` **`fieldnames`** a list of field names

Several shortcuts are available.

Omitting the index of an array causes all of the array values to be returned. A subrecord name causes all of the fields to be returned

Returns: a dataset containing the values of each of the fields

Null is returned if an error occurred.

`Boolean modifySubrecordArraySize(name, newsize)`

Description: changes the size of a subrecord array field. Array sizes for primitive fields are fixed. A subrecord must be declared as an array (e.g., size > 0) to be resized.

Parameters: `string` **`name`** the reference name

`int` **`newsize`** a new array size

Returns: false if errors are encountered else true

`int readFieldSize(fieldname)`

Description: returns the size of a field in a record

Parameters: `string` **`field`** the *full* name of the field (includes record name)

Returns: the size of the field (zero for non-array), -1 on error

2.3.4. Query

A query searches a collection of records (referred to as the *targets*) using a Boolean expression. The Boolean expression is applied to each record. A *match* occurs if the expression evaluates to true, causing a set of fields to be returned from that record. We define a single flexible method for performing queries.

`DataSet query(return_fields, target_list, expression)`

Description: return a set of fields from every record in the target list that matches the given expression

Parameters: `string[]` **`return_fields`** the *relative* name of each field to be returned, starting after the name of the record being queried

`string[]` **`target_list`** a list of records or template types to search. if a template name is specified, all records of that type will be queried

`string` **`expression`** a Boolean expression written in the virtual database query language

Returns: one set of return values for each match. An empty dataset (zero sets) is returned if no matches are found.

Null is returned if an error occurred.

A query expression is composed of field names, operators, and special characters. Field names may be expressed in dot-style, associative index, or dereferenced associative index notation (see Appendix A for query examples). Operators include the standard rela-

tional and logical operators. Special characters are used to control how the expression is applied to each record in the target list. Special characters are highest in precedence, followed by relational and then logical operators. Parentheses are used for readability or to override the default precedence order.

There are three special characters. The dollar sign ($) represents the name of the record that is currently being queried. It is the most common way to create a full reference from a relative field name. A double dollar sign ($$) indicates that the reference is an absolute path name. The field name following a double dollar sign is a complete (absolute) reference including a valid record name. This is useful for comparing several records in a target list (specified using $) to a single value stored in some other record (specified using $$). Every field name in a query expression must be prefaced by $ or $$. The third special character is the at sign (@) which represents array explosion. Array explosion is used to instruct the query engine to apply the query to each element of an array. An array is referenced using the at sign (`@myarray`) instead of an associative or positional operator that would single out a particular element. Array explosion notation is used in the return field list as well as in the expression. A special field-naming convention is used for the return fields containing an array explosion. The return value is named *val_field_X* where *X* is a reference number starting at zero that indicates the return field (i.e., the first return field is number zero). This is required because the name of each match will be different due to the exploded array indices. The convention allows the user to retrieve the data using a generic name. Additional information is included in the DataSet returned from each query using array explosion. A string variable containing the name of the field that produced the match (i.e., the full name using square brackets to indicate the element) is returned under the name *name_field_X*. An integer value containing the index (element number) of the exploded array is also provided as *index_Y* where *Y* indicates the array explosion (i.e., the first @ is zero). The special string field *vdb_record_name* is included in every set. It contains the name of the record that produced the match.

The relational operators are equals (=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), and tilde (~). The first five operators are used to compare a numeric or Boolean field value to another field value or a constant value. The

tilde (~) operator is used for string operations. Strings (varchar primitive type) can be compared to absolute values or partial values. Wild cards are indicated by a percent sign (%). The expression `"$name ~ 'queue_a%'"` would be true for all records in the target list whose name fields begin with `'queue_a'`. Note that string values are always enclosed in single quotes. The logical operators are and (&), or (|), and not (!).

Our query language meets the demands of Möbius data analysis applications while maintaining an abstraction generic enough to be implemented using a wide variety of database packages. See Appendix A for a tutorial containing detailed example queries.

2.4. Möbius Access (MAC) Layer

The MAC layer builds a database for storing modeling results. It uses the generic database presented by the virtual database layer. Solvable models produce results. This presents a challenge because the solvers and reward models all have different user-controllable options. A solver will support some set of the reward models implemented in Möbius. Additionally, different solvers may generate different solution values for the same reward model. Recall our goal of making the results database intuitive. We would like to organize the data to preserve similarities among solvable models while making allowances for their differences. This is accomplished using the inheritance model made popular by object-oriented programming languages. Results from any type of solvable model are saved into a data format called a *run*. Data in a run is divided into several categories. The structure of some portions will be the same regardless of the solver and reward models used. Other parts are solver/reward-specific. These parts contain data in a format defined by their type. Every run inherits a general structure, with reward/solver-specific formats where needed.

A run is divided into four sections. The *header* contains general information to identify the run, and is the same for all models. It contains information such as the versions of all of the models, the project name, and global variable declarations (name and type). The *settings* section contains the state of all of the solvers' parameters. Next is *the reward variable declaration* section. The name and settings of each reward variable are defined here. The fourth section contains data for each experiment, where an *experiment* is defined as one solver execution. There will be one experiment for each permutation of global variables in the study.

We will now define several terms that apply to this layer. Note that they are similar to those used in the virtual database. We introduce new terminology to emphasize constructs used at the Möbius (system modeling) level. A *structure* is defined as a format for storing data. Structures are implemented using templates in the virtual database. An *entity* is an instance of a structure, stored as a record. Finally, a unit of data, a field, is referred to as a *variable*.

A *results database* is a collection of run entities. A run structure exists for each solver, inheriting the generic four-part format previously defined (see Figure 8). A run contains a header subentity named *hdr*, which contains the same information for all run entities. The variables contained in a header are described in Table 2.

Table 2. Components of a Header Structure

Variable Name	**Description**
run_name	A unique identifier for this run entity
proj_name	The Möbius project that produced the results
solver_code	A code to identify which solver was used
solver	The solver instance (created under the solver tab) used
comment	A place to include notes about the run
started	Day/time that the solvable model started
finished	Day/time that the solvable model completed
duration	Total time for the run
models	Version number for each component instance in the project
gd	Global variable declarations

The second section stores all of the solver's parameter values. These parameters are set using a GUI interface under the solver tab. An example parameter is a random number seed used by a simulator. The settings structure might also contain solver information that is reported only once per run.

The third section contains an array of reward variable declarations for each reward variable type supported by the solver. The structure contains the name and settings. A user enters this information under the reward tab in Möbius. A setting might be a confidence interval. Note that a particular solver may not use some settings' values. For example, ana-

lytic solvers disregard the confidence interval setting, because they produce exact answers. The name of the array is specified using the MAC API described later.

The final subrecord of the run structure is the experiment array. The size of the array is determined by the number of experiments. The experiment structure also contains both generic and solver-specific parts. There are two variables common to all solvers. First, there is a variable that stores the name of the experiment. Second, there is a subrecord that stores the values of each global variable defined in the header. The values are stored in arrays according to the variable type. A global variable's value is stored at the same index as its declaration. A dereferenced associative index can be used to retrieve a global variable value based on its name. There are also two solver-specific sections. The first is an information substructue that stores general results supplied by the solver after each experiment (for example the number of states produced by the state space generator). The second is an array of solver solutions for each reward variable type. There will be one array for each reward variable type supported and one element for each reward variable defined. The solution structure might contain a mean, variance, or distribution.

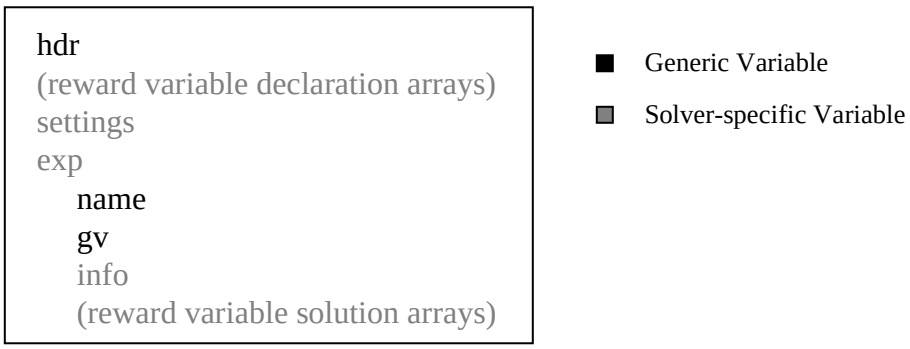


Figure 8. Run Structure Inheritance

We use several internal structures to manage the results database. RunDir is an array of entities containing the names and solver types of all of the runs in the database. StructList is an array of name and type information for all of the solvers. There are five structure types in the results database. Structures can describe *reward variable settings* (RV), *solver settings* (s_settings), *solver experiment information* (s_exper_info), or a *solver solution* to a particular reward variable type (s_solution). Any other structure (i.e., a subentity used in the previously defined structure types) is called a *generic structure*.

The final internal entity tracks the version history of the database. New solvers and reward models will be added to Möbius. The settings and solutions of existing solvers may change. It is important to provide a way to upgrade a results database in response to these events without losing data. All of the structures used in the database are defined when it is created. At creation time, we assign a version number to identify the state of the structures. The current version number is incremented as new solvers become available or changes are made to existing ones. The version number is examined each time the database is opened. If the version number does not match the current version number, an upgrade method can be performed to add or adjust structures.

In the next section, we will present the API supported by the MAC layer. The MAC is the *only* interface between the results database and the outside world. It is closely tied to the constructs in the virtual database. There are methods for database administration, structure management, entity management, and queries. Parameters are often the same as defined in the virtual database.

2.4.1. Database administration

Database administration methods take care of high-level database operations. The methods are the same as the virtual database administration methods.

`Boolean dbaCreate(dbname)`

Description: creates a new database with the specified name

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean dbaDelete(dbname)`

Description: permanently removes the database with the specified name

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean dbaOpen(dbname)`

Description: opens a database; only one database can be open in a given

MAC object

Parameters: `string` **`dbname`** the reference name

Returns: false if errors are encountered else true

`Boolean dbaClose()`

Description: closes the database that is currently open

Parameters: none

Returns: false if errors are encountered else true

`Boolean dbaCopy(newdbname)`

Description: copies the contents of the currently open database into a new database with the specified name

Parameters: `string` **`newdbname`** the new database reference name

Returns: false if errors are encountered else true

`Boolean dbaSetUserLogin(username, password)`

Description: sets user login information for the core database software

Parameters: `string` **`username`** username given by the database administrator

`string` **`password`** password for account

Returns: false if errors are encountered else true

`Boolean dbaSetServerInfo(hostname, port)`

Description: sets the location of the core database server

Parameters: `string` **`hostname`** the name of the machine running the server

`int` **`port`** the port number for the server

Returns: false if errors are encountered else true

2.4.2. Structure administration

Structure administration methods control the data structure definitions. Recall that each solver must specify several data structures. The settings structure saves user options and solver information that are reported only one time. The info structure contains general information that is reported once per experiment. A solution structure must be specified for each reward variable type that a solver supports.

`Boolean dbsAddGenericStructure(name, description, vars)`

Description: adds a generic structure to the database

Parameters: `string` **`name`** a reference name used in subentity declarations

`string` **`description`** an explanation used in a browser

`string` **`vars [][4]`** a two-dimensional description of the variables.
One row describes each variable.
Column 0: type of variable
Column 1: name of variable
Column 2: size of variable (zero for a single value)
Column 3: description of variable

Returns: false if errors are encountered else true

`Boolean dbsAddRewardVariableType(name, settings_vars)`

Description: adds a new reward model to the database

Parameters: `string` **`name`** a reference name. It is the name used for the reward variable settings and solution subentities of run structures

`string[][4]` **`settings_vars`** user-configurable options specified in the reward tab.

Returns: false if errors are encountered else true

Boolean dbsAddSolver(name, settings_vars, info_vars)

Description: adds a new solver to the database

Parameters: `string` **`name`** a reference name

`string[][4]` **`settings_vars`** user-configurable options

specified in the solver tab

`string[][4]` **`info_vars`** generic results reported once per experiment

Returns: false if errors are encountered else true

Boolean dbsAddSolverSolution(solver, reward_type, vars)

Description: adds support for the specified reward type to the specified solver

Parameters: `string` **`solver`** the solver name

`string[][4]` **`reward_type`** a reward model

`string[][4]` **`vars [][4]`** solutions computed by the solver for each reward variable

Returns: false if errors are encountered else true

Boolean dbsRemoveStructure(name, category)

Description: permanently deletes a structure and all instances of it

Parameters: `string` **`name`** the reference name

`string` **`category`** the structure type (generic, rv, s_settings, s_exper_info, s_solution)

Returns: false if errors are encountered else true

Boolean dbsModifyStructure(name, category, modifications)

Description: adds, removes or resizes variables

Parameters: `string` **`name`** the reference name

`string` **`category`** the structure type (generic, rv, s_settings, s_exper_info, s_solution)

`string` **`modifications[][]`** a set of modifications with one row per modification. Each row contains the following:

Column 0: action (ADD, REMOVE, RESIZE)

Column 1: field name

Column 2: size (for RESIZE only)

Column 3: type (for ADD only)

Column 4: description (for ADD only)

Returns: false if errors are encountered else true

`string[] dbsGetRewardVariables( )`

Description: returns a list of supported reward models

Parameters: none

Returns: the name of each reward variable type defined

`string[] dbsGetSolvers( )`

Description: return a list of solvers

Parameters: none

Returns: the name of each solver supported

`string[] dbsGetSolutionTypes(name)`

Description: returns a list of reward models supported by a solver

Parameters: `string` **`name`** the name of a solver

Returns: the name of each reward variable type supported

2.4.3. Entity administration

Entity administration methods manage structure instances. A wide variety of read methods are provided.

`Boolean dbeNewRun(name, solver, description)`

Description: creates a new run entity

Parameters: `string` **`name`** a reference name

`string` **`solver`** the solver name used

`string` **`description`** a description provided to a browser

Returns: false if errors are encountered else true

`Boolean dbeFreeRun(name)`

Description: permanently removes a run entity

Parameters: `string` **`name`** a reference name

Returns: false if errors are encountered else true

`Boolean dbeCopyRun(s_name, d_name)`

Description: copies the contents of a run into another run

Parameters: `string` **`s_name`** the source record

`string` **`d_name`** the destination record

Returns: false if errors are encountered else true

`Boolean dbeSetRunLock(name, islocked)`

Description: sets the read/write permission of a run

Parameters: `string` **`name`** the reference name

`Boolean` **`islocked`** true for read-only, false for read/write

Read-only access prevents an entity from being modified or deleted.

Returns: false if errors are encountered else true

`Boolean dbeGetRunLock(name)`

Description: returns the read/write permission of a run

Parameters: `string` **`name`** the reference name

Returns: the state of `islocked` defined in `dbeSetRunLock`

`DataSet dbeReadVariables(name, variablenames)`

Description: reads an arbitrary set of values

Parameters: `string` **`name`** a run name

`string[]` **`variablenames`** a list of variable names

Several shortcuts are available.
Omitting the index of an array causes all of the array values to be returned. A subentity name causes all of the variables to be returned

Returns: a dataset containing the values of each of the variables
Null is returned if an error occurred.

`DataSet dbeReadRun(name)`

Description: reads the entire run entity

Parameters: `string` **`name`** a run name

Returns: a dataset containing the values of each of the variables
Null is returned if an error occurred.

`DataSet dbeReadHeader(name)`

Description: reads the header subentity

Parameters: `string` **`name`** a run name

Returns: a dataset containing the header variables
Null is returned if an error occurred.

`DataSet dbeReadModelInfo(run, model)`

Description: read the version information for a specific model

Parameters: `string` **`run`** a run name
`string` **`model`** a model name

Returns: a dataset containing the version of the model
Null is returned if an error occurred.

`DataSet dbeReadGlobalVariable(run, exp, gvtype, gvname)`

Description: reads the value of a global variable for a given experiment

Parameters: `string` **`run`** a run name
`string` **`exp`** an experiment name
`string` **`gvtype`** the data type of the global variable

`string` **`gvname`** the name of the global variable

Returns: a dataset containing the global variable value
Null is returned if an error occurred.

`DataSet dbeReadRewardVariable(run, exp, rvtype, rvname)`

Description: reads reward variable data for a given experiment
The settings and solutions subentities will be returned.

Parameters: `string` **`run`** a run name
`string` **`exp`** an experiment name
`string` **`rvtype`** the reward model
`string` **`rvname`** the name of the reward variable

Returns: a dataset containing the reward variable value
Null is returned if an error occurred.

`DataSet dbeReadExperiment(run, exp)`

Description: reads all the data for a given experiment

Parameters: `string` **`run`** a run name
`string` **`exp`** an experiment name

Returns: a dataset containing the experiment data
Null is returned if an error occurred.

`Boolean dbeWriteVariables(name, data)`

Description: writes values into variables

Parameters: `string` **`name`** the reference name of an entity or subentity
`DataSet` **`data`** the variable name/value pairs

Returns: false if errors are encountered else true

2.4.4. Query

The query function is mapped directly onto the query method supplied by the virtual database. It may be useful to know the structure-naming conventions used in the database. The name of a solver settings structure is *Xsettings* where *X* is the name of the solver. The

experiment info subentity is called *XExperInfo*. The experiment structure is *ExperX*. The solver *X* solution to reward model *Y* is named *XY*. A generic structure adopts the name specified in the declaration method.

`DataSet dbqQuery(return_fields, target_list, expression)`

Description: return a set of variables from every entity in the target list that matches the given expression

Parameters: `string[]` **`return_fields`** the *relative* name of each variable to be returned, starting after the name of the entity being queried.

`string[]` **`target_list`** a list of variables or structure types to search. If a structure name is specified, all entities of that type will be queried. Subentities are also valid targets.

`string` **`expression`** a Boolean expression written in the virtual database query language

Returns: one set of return values for each match. An empty dataset (zero sets) is returned if no matches are found.

Null is returned if an error occurred.

3. MODEL CONNECTION FRAMEWORK

3.1. Introduction

An important research use of the results database is the support of model connection. Recall that a connected (hierarchical) modeling methodology uses results calculated from one model as inputs to another model. This technique is useful because it is a flexible way to allow communication between heterogeneous models (expressed in different modeling formalisms). Connection may also be used to solve a large homogeneous model more efficiently. We can view a connected model as a directed graph in which nodes represent individual models (called submodels) and arcs represent results flowing between models (see Figure 9). This idea is also used in the POEMS project [5]. Note that the ideas presented in this Chapter have not yet been implemented in Möbius.

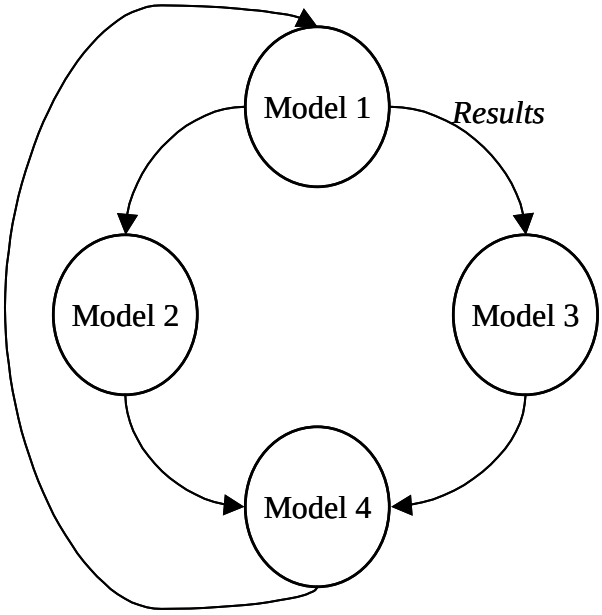


Figure 9. Abstraction of a Connected Model

Connected models have been the topic of several research publications (e.g., [12], [13]). Model connection research maps onto the Möbius framework very well. Generic result transformation techniques can be realized through development of a new connection formalism. It is possible to carry out experiments for a particular system by building a

model of the system that uses the atomic/composed/connected formalisms available in Möbius. It is possible to explore solution techniques by building a new connected model solver.

We know that connection models span a huge class of problems, including connection of homogeneous and heterogeneous models in various topologies that may be cyclic or acyclic. Because of this wide diversity of models, it is not possible to find a single "representative" connected system upon which to model our solution. However, the solution process can be generalized at a higher level. We illustrate this idea with the flow chart in Figure 10. Several steps are involved in solving a connected model; exploration, solution, and completion. We use this insight to derive the generic functionality that our connection platform will require.

To solve a connected model, we begin with an exploration phase. In this phase, generic information about the submodels and connections that have been instantiated is examined. The number of submodels and the solution techniques they employ may be of interest to the solver. Similarly, it is important to understand the topology of the connections.

During the solution phase the solver will move about the connected model, solving submodels until some stopping criteria have been reached. First, in the navigation phase we identify a submodel to solve. Next, we initialize the submodel by passing in the current value of each result used in that model. Then we solve the submodel. Finally, we move to the interpret phase, in which we look at the submodel's results and determine whether we have finished solving the connected model.

The completion phase allows activities to finish analysis of the entire system. This section might calculate measures for the entire model based on the results of the submodels.

The Möbius connection platform defines the roles of connection formalisms and solvers and defines an interface between them. We will also define the software architecture used by connection solvers. The formalism/solver interface will become part of the Möbius abstract functional interface [3] (see Figure 11).

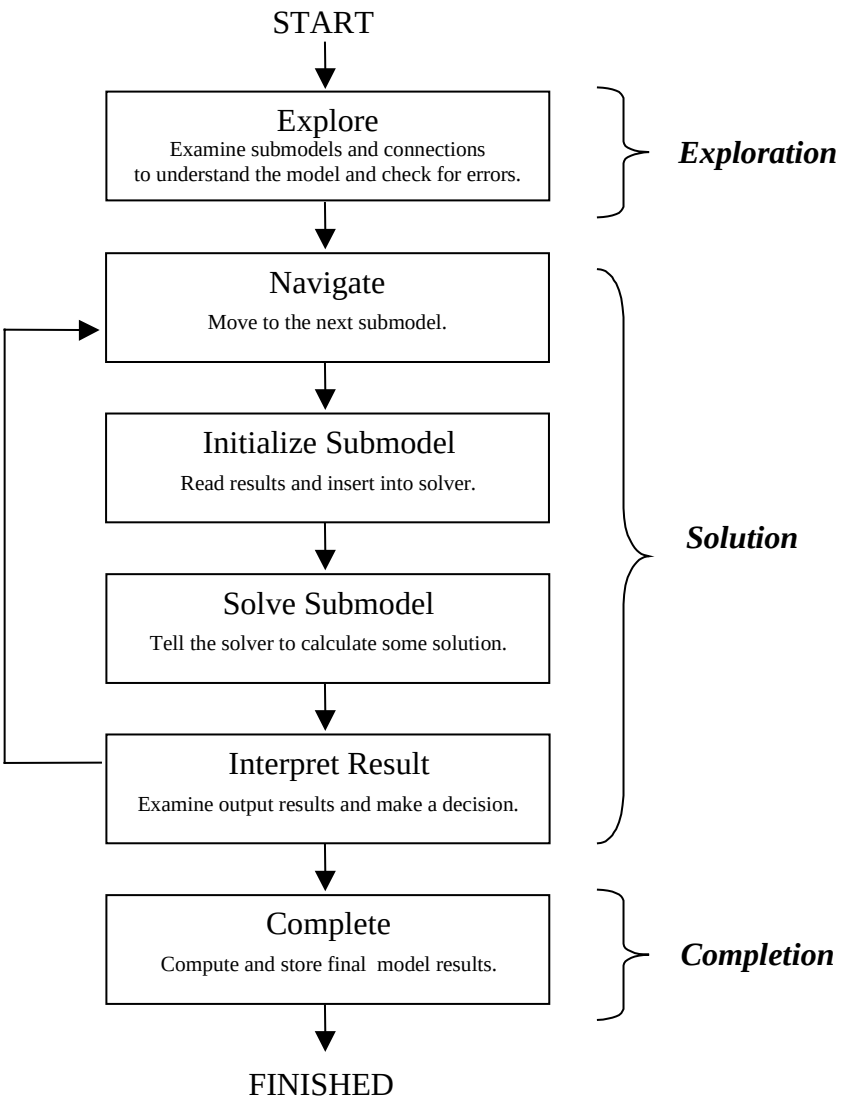


Figure 10. Connected Model Solution Process

3.2. Connected Model Interface

This section explains the abstract functional interface extensions for connected models. First, we explain the basic elements defined at this level. Next, we present the proposed extension methods.

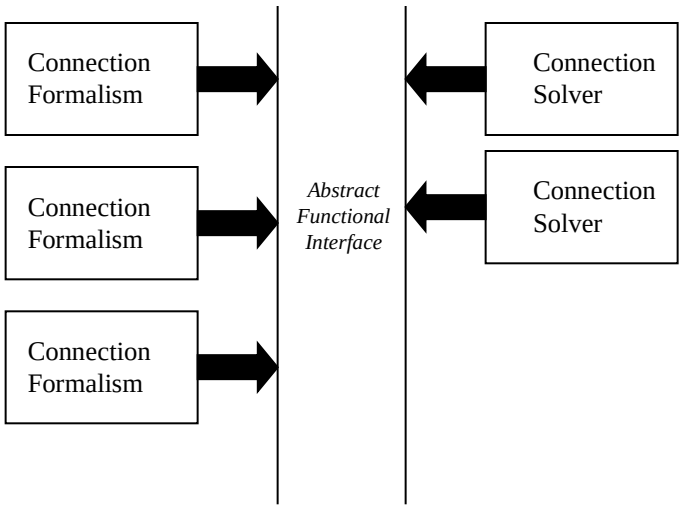


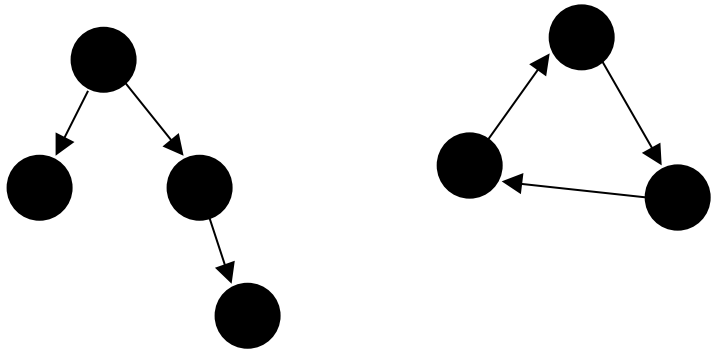
Figure 11. Connection Research Platform

A connected model can be represented as a set of solvable models and a diagram for passing results among them. This relationship can be represented as a directed graph, a set of nodes and arcs, and a traversal order. Nodes represent completely specified models and arcs represent the flow of results from one model into another. Traversal order can be implied in acyclic graphs (see Figure 12). A set of producer-only models can be identified and solved first, making results available to drive at least one consumer model; that will in turn provide results needed to solve other consumer models. This process continues until all models have been solved.

Cyclic connections (illustrated in Figure 12) introduce two additional requirements. First, if a set of submodels is connected in a cycle, it follows that no submodel can be driven until it has already been solved, thereby producing the input results needed to drive it. This circular dependency is impossible to satisfy in one pass. We must provide a way to drive the first model in the cycle with an estimate of what its input results will be. This guess is used to allow an initial solution of the submodel, which will be used to drive the other models in the cycle. At that point, all of the needed results will be available. It will then be possible to solve the cycle of submodels a fixed number of times or until the results passed between the submodels meet some convergence criteria. Second, a convention must be defined for specifying the order in which cyclic submodels are solved. Recall that acyclic connections have an implied order. This is not true in the cyclic case, because there are no producer-only

submodels. Every submodel consumes results. The interface must provide a way to specify, using estimation, which submodel will be solved first.

Our overall goal is to maximize the modeler's ability to combine results. Mathematical techniques for transforming results into values that are meaningful in another model are a common subject of connected model research. It is natural to view the exchange of results as a one-to-one mapping of results generated in one model and used in another. However, it is usually necessary to perform some computation on the result before passing it along. It is conceivable that a modeler would want to use results generated in several models to compute a value used in another model. These situations motivate the inclusion of an intermediate processing point between the models that produce the results and the models that consume them.



Acyclic Models ***Cyclic Models***

Figure 12. Connection Topologies

We will now discuss the building blocks of our connection interface: the submodel, the connection function, the conduit, and the database. We will define a graphical notation for representing each one. This chapter introduces the proposed infrastructure for supporting model connection. No connection formalisms and solvers have been implemented in Möbius at this time. Chapter 4 uses a prototype of the proposed software architecture to implement a connected model in the Möbius application.

3.3. Möbius Connection Constructs

3.3.1. Submodel

A *submodel*, which is graphically represented as a hollow circle, is a completely specified model. The models, measures, and solution technique have been defined. The submodel is an active primitive that will produce results and may consume them as well. Each submodel instance has a name and identification number.

3.3.2. Connection function

A *connection function*, which is graphically represented as a hollow square, is a processing station for results. It is possible to direct results from a submodel into a connection function. At that point a method operates on the results to produce values that will be used to drive other submodels. The connection function exists for two reasons. First, it provides a way to perform the mathematical transformations that are often necessary. Second, it provides a facility to compute an initial value for the first submodel solved in a cycle. Each connection function has a name, function body, identification number, and set of conduits associated with it. Data enters and leaves the connection function via conduits. All submodel and connection function identification numbers are unique.

3.3.3. Conduit

Conduits, which are graphically represented as unidirectional arrows, are used to direct results to and from a connection function. We distinguish the to/from cases because the underlying software implementation mechanisms are quite different. Each conduit is defined by a type (database or driver), a source, a sink, and a list of results that are passed through it.

Database conduit

Database conduits, which are represented as dashed arrows, imply reading data from the results database. The source of the conduit may be a submodel, implying results generated from the last solution of that model. Alternatively, the source may be a database primi-

tive, as described in the next section. The sink of the database conduit is a connection function.

Driver conduit

Driver conduits, which are represented as solid arrows, pass input data to other submodels. The source is the connection function, in which the values have been computed. The sink is a submodel. Driving values are computed and passed into a submodel before it is solved.

3.3.4. Database

The *database*, which is represented graphically as a cylinder, implies reading an arbitrary result from a results database. This feature is useful for computing initial values for the first submodel in a cycle. It could also be used to bring old solution values into a connection function to help determine stopping criteria. Each database has a name that indicates the results database to read from. Databases also have unique identification numbers.

3.4. Extensions to the Abstract Functional Interface

The abstract functional interface is the interaction point for formalisms and solvers. It is important to understand the roles that we assign to each side of the junction. A connection formalism is responsible for presenting a connected model as a set of submodels, connection functions, conduits, and databases. A submodel may have predecessors and successors. A submodel encountered along any backward conduit path starting at the driver conduits of a submodel is defined as a predecessor of that submodel. Similarly, a submodel encountered along any forward conduit path beginning at the database conduits of a submodel is a successor of that submodel. The predecessors of a submodel may influence the values of its input results. The successors of a submodel may be influenced by its results. A cycle is characterized by the appearance of a submodel in another submodel's successor and predecessor lists.

We illustrate the predecessor/successor relationship with an example of a connected model that is described using the Möbius constructs introduced in Section 3.3 (see Figure 13). Our connected model is composed of five submodels. Submodel D may be influenced

by its predecessors, submodels A and B. It may influence its successor, submodel E. Submodel D has no relation to submodel C. It is impossible to make an absolute determination of the relationship between two models without a detailed analysis of the code inside each connection function. For example, CF 2 might contain code that will only use results from submodel A to calculate values for submodel D. Therefore, submodel B will not influence submodel D, although it appears in B's predecessor list. Our predecessor and successor definitions will identify all true dependencies in the model, but may also identify false relationships due to the connection function phenomenon we have just described. These false dependencies may enforce a more rigid solution order than necessary, but will not prohibit a solution.

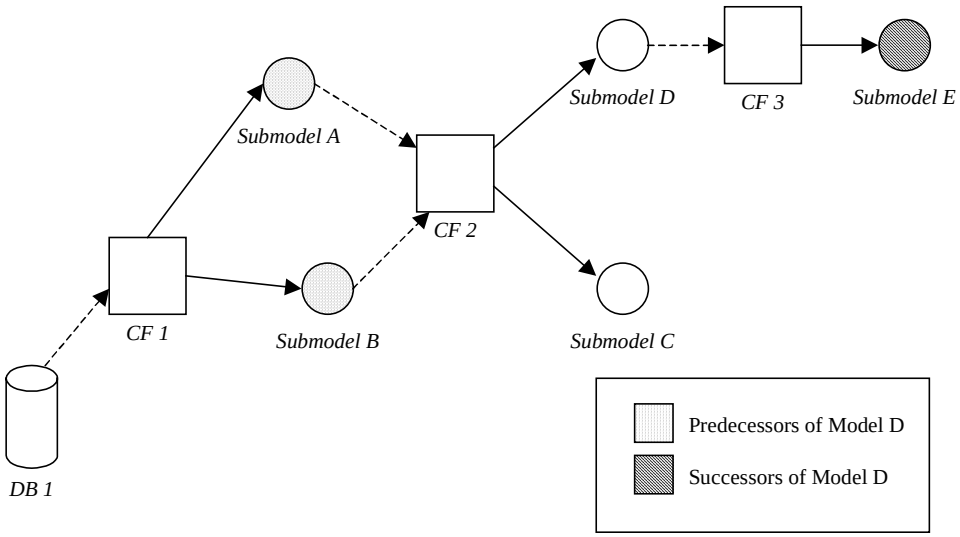


Figure 13. Example Connected Model Represented with AFI Building Blocks

A connection solver executes the submodels in accordance with the following rules.

1. For acyclic topologies, a submodel will not be solved until all of its predecessors have been solved.
2. A cyclic solution begins with the solution of the submodel with the lowest identification number.
3. Every connection function attached to a submodel via a driver conduit will be evaluated before the submodel is solved.

4. The solver will finish immediately if an EXIT instruction is encountered in a connection function. The EXIT instruction is a special method supported by all connection solvers. It allows a model to override the stopping algorithm implemented by the solver.

Our division of responsibility between formalisms and solvers keeps most of the control at the formalism level. This is a good choice, because we want to give formalism developers the ability to build powerful, flexible primitives. Users interface with Möbius at the formalism level. We want to provide a rich interface for abstracting a system as a connected model. We anticipate that most connection solver research will focus on developing convergence techniques for cyclic models. For other research, a simplified connection solver capable of solving submodels in accordance with the general rules is sufficient. Cycles could be handled in a trivial way such as by iterating a fixed number of times.

3.4.1. Abstract functional interface methods

The building blocks described in the previous section are passed between connected formalisms and solvers using the abstract functional interface. A connection formalism must present the following methods to a connection solver. These methods retrieve the instances of each building block type.

`GetSubmodels()`

Description: gets a list of the submodels in the connected model

Parameters: none

Returns: the name and identification number of each submodel instance

`GetConnectionFunctions()`

Description: gets a list of connection functions

Parameters: none

Returns: the name, identification number, and function body of each CF

`GetConduits()`

Description: gets a list of conduits

Parameters: none

Returns: the type, source, sink, and result list for each conduit
The result list contains the name and data type of each result flowing through the conduit.
Database conduit results are run entity variables.
Driver conduit results are global variables.

`GetDatabases()`

Description: gets a list of database blocks

Parameters: none

Returns: the name of each results database

3.5. Connection Software Architecture

We will now examine the software architecture of a connected solver. We will explain how each building block is realized in the Möbius framework and will also discuss the process for linking connected models and solvers.

3.5.1. Solver architecture

Recall that submodels must be completely specified models (models, measures, and solution techniques have been defined). Möbius solvable models meet this requirement. A solvable model has two parts. First, a C++ or C program is compiled into a native binary executable program. This program will solve the model for a single experiment. The second part of the solvable model is a Java thread that invokes the program. It relays user parameters stored in the solver GUI to the executable using command line options. It invokes the program once for each experiment in the study. Our prototype solution invokes a solvable model using the Java thread. Our final solution may invoke the solver directly. Solvable models have the advantage of being pre-instrumented to write to the results database. A connected model produces a collection of run entities, including one for each submodel and, optionally, one to store results computed for the entire connected model. Cyclic connections cause a new run entity to be created each time a given submodel is solved. This provides a complete account of the convergence process.

The connection function is a set of high-level language instructions used to transform results from a submodel into values that can be used by other submodels. We implement this function as a method call in the language of the solver. Our prototype uses Java to perform connection functions.

Database conduits pass results into a connection function. We implement this function by reading the specified variables from an existing results database. If the source of a database conduit is a submodel, the results should come from the last execution of that submodel. They reside in the database currently open in the Möbius application. If the source of the conduit is a database block, the results will be read from the given database name. The value of each result is read and made available in a variable accessible from the connection function.

Driver conduits pass values computed inside a connection function into another submodel. Each value is removed from the connection function after being placed in a predetermined variable accessible by the solver. The value is then passed into the solvable model via a global variable. Recall that a global variable can be used to specify the value of any formalism primitive parameter. Its value is determined by the study during a normal solvable model execution. This procedure is altered when a solvable model is used as a submodel inside a connected model. A driver conduit will force the program to disregard the value set by the study and will insert the value computed in the connection function. This process is called *overriding a global variable*. This procedure enables a model to receive results at any parameter boundary, and is therefore a very flexible solution.

The final feature is the database block. Its purpose is to make an arbitrary results database accessible to connection functions. We implement the database by closing the current database and opening the requested one before reading results specified on any database conduit connected to the database block.

3.5.2. Model/solver binding process

A solvable model is a completely specified Möbius model. In this section, we use the term *solvable model* to refer to the executable program created by Möbius. We will explain the process for creating a solvable connected model. It is similar to the process for

building a solvable atomic model. The abstract functional interface defines the methods and data structures used to present a model to a solver. A model is built using the primitives available in a particular formalism. An intermediate (formalism) library is used to transform primitives into the language of the abstract functional interface. These components, all of which are written in the same programming language, are compiled together to build an executable program.

A connected model is described using a connection formalism. The connection formalism library maps its primitives onto the submodel, connection function, conduit, and database constructs supported by the abstract functional interface. All are compiled into an executable, as shown in Figure 14. Note that submodels, which are implemented as solvable models, are *not* included in the binary. They are invoked by the connection solver in their own process spaces. This is the same as the procedure used to solve a model from the Möbius GUI application.

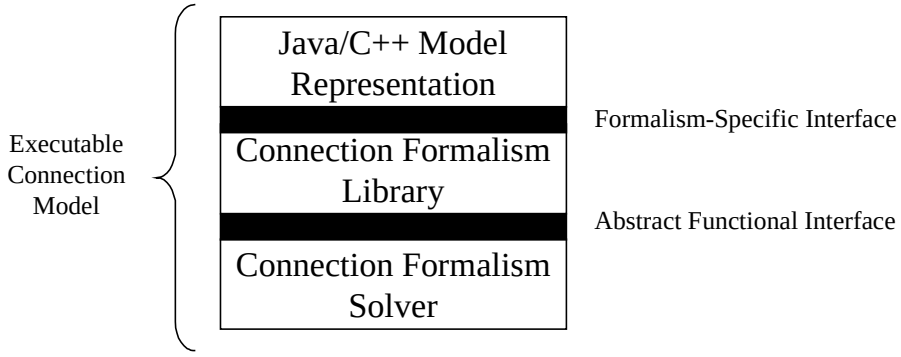


Figure 14. Solvable Connected Model

3.6. Connected Solver Toolkit

The Möbius connection framework defines a set of useful constructs and rules for evaluating them. We define a library, called the Solver Toolkit, that connected model solvers can use to manage this process. Connection functions are used to process results before they are passed into another model. This is an important feature, but is probably not of interest at the solver level. Solvers observe the final value of results that are passed into a submodel. This is the point at which convergence can be measured. The Solver Toolkit presents the model using our simple directed graph abstraction. Each node represents a

submodel and and each arc represents values flowing from one model into another. Figure 15 illustrates the solver-level graph for the model described in Figure 13. Note that arcs correspond to the predecessor/successor relationship defined earlier. The graph is represented by a two-dimensional connection matrix. It is an $N \times N$ matrix of Boolean values, where N is the number of submodels. A "1" in row I column J indicates an arc from submodel I to submodel J.

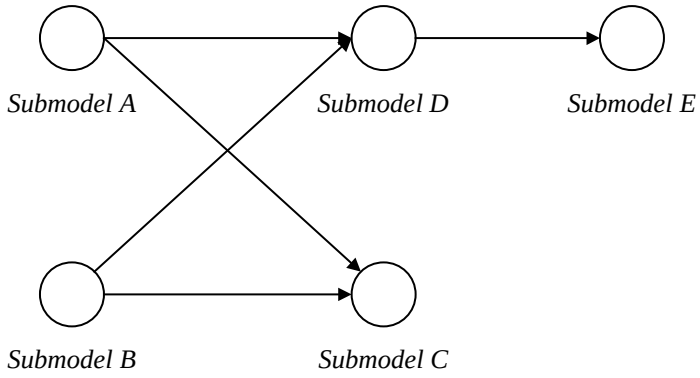


Figure 15. Solver Toolkit View of Example Model

The Toolkit will manage the low-level process of feeding results from a database into a connection function, evaluating that function, and passing the resulting values into the required submodels using the global variable override technique. It is appropriate to package this functionality as a library, because it is common to all connection solvers. The Toolkit is a starting point for designing a connection solver. The low-level details are taken care of, leaving the developer free to focus on his or her particular solution idea. The Solver Toolkit uses the abstract functional interface to get the details of the connected model. While the abstract functional interface is always accessible, we expect that connection solvers will find the toolkit interface sufficient for their needs. The toolkit interface is defined below.

`Boolean[][]GetConnectionMatrix()`

Description: returns the connection matrix for the model

Parameters: none

Returns: a 2-D array of bits representing submodel connections

`enum GetTopology()`

Description: characterizes the topology of the connection graph

Parameters: none

Returns: linear, tree, cyclic, irregular

`int GetNextAcyclicSubmodel()`

Description: returns a valid acyclic model to solve

Parameters: none

Returns: the identifier of a submodel whose predecessors have been solved

`int[][] GetCycles()`

Description: returns all the cycles in the graph

Parameters: none

Returns: a 2-D array of submodel identifiers where each row contains a cycle

`Boolean SolveSubmodel(id)`

Description: solves a submodel and execute the required connection functions

Parameters: `int` **`id`** submodel identifier

Returns: false if errors were encountered, else true

`DataSet ReadInputs(id)`

Description: reads the current values of the results being passed into a specified model

Parameters: `int` **`id`** submodel identifier

Returns: each global variable override value

4. CONNECTED MODEL CASE STUDY

4.1. Introduction

A connection formalism can be used to reduce computer resource rquirements and solution time for models solved using state-based techniques. We will demonstrate the use of connection within the Mobius modeling tool by solving such a system twice: abstracted first as a single model and then as a connected model. We will examine the solution time, state-space size, and accuracy. We will also demonstrate the impact of result estimation on a cyclic connected model.

We conducted the experiments using a prototype of the connection platform defined in Chapter 3. We designed and implemented a Java class to invoke and drive the Möbius application in an automated fashion. The program solves each submodel by opening its project and solver instance in the Java GUI and creating an automated "push" of the start button. Results are saved into the database and subsequently retrieved for the connection function computation. The results are then passed to successor submodels using the global variable override technique presented in Section 3.5.1.

4.2. System

Queueing networks have been a subject of connected model solution research [12], [13]. We will analyze an open network of fixed-length queues (see Figure 16).

There are three queues in the network. Each has a single processing station with service time characterized by an Erlang-2 distribution. Queue A services customers at a faster rate than the other queues. Jobs enter the network through Queue A, with an exponentially distributed time between job arrivals. Jobs are discarded if the queue is full. After receiving service at Station A, jobs proceed to the remaining queues (B and C) or exit the system based on a fixed probability. Queue B and Queue C receive 25% and 50% of the jobs, respectively. The remaining 25% of jobs exit the system. Jobs routed to Queue B or Queue C are discarded if their respective queues are full. All jobs leaving stations B and C are routed back to Queue A. We will study the utilization (fraction of time that the server is busy), average queue length, and probability that each queue is full in steady-state.

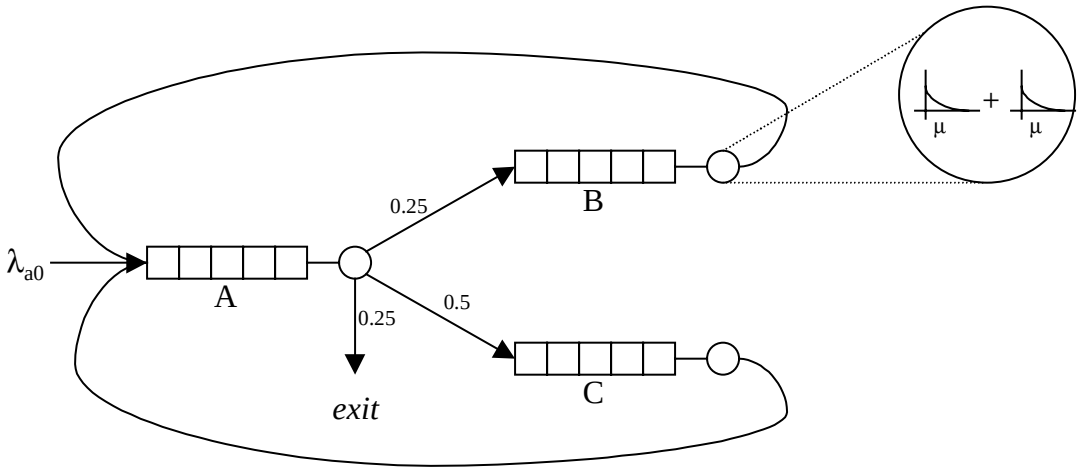


Figure 16. Erlang-2 Queue Network

Mathematical techniques are available for solving some classes of queueing networks, called *product-form networks*. Although fast and accurate, these solutions make certain assumptions about the system. For example, they assume that there is an infinite queue capacity; it is therefore impossible for us to use a product-form solver for our model. Queuing networks can also be solved with state-based techniques. These solvers represent a model as a Markov process. A state represents the status of the system at a given point in time. Any possible configuration of the system will have a state associated with it. In our system, the state can be characterized as the number of jobs in each queue and service station. We will use a state-based solution technique on our system. However, if applied directly, state-based techniques will result in unmanageable state-space sizes. With a connection model it is possible to break the system into smaller pieces that can be solved separately.

4.3. Model

We modeled the system using the SAN formalism and the connection framework defined in Chapter 3. First, we created a model for a single Erlang-2 queue to serve as a submodel in our connected model (see Appendix C for the complete model description). An Erlang-2 server has two stages, each of which has an exponential service time. A job cannot enter the server until the previous job has received service at *both* stages. Thus, the overall

service rate is the sum of the two exponential service rates; this is an Erlang-2 distribution. Recall that Queue A services customers at twice the rate of Queues B and C. Therefore, we parameterized the service rate using a global variable. The new customer arrival rate was also declared a global variable.

Our connected model has three submodels (see Figure 17). We use the notation *X.y* to indicate the reward variable *y* of submodel *X*. Similarly, *X.Y* indicates the global variable *Y* of the submodel *X*. Each is an instance of the single Erlang-2 queue described in the previous paragraph. We circulate jobs between queues in an abstract way based on the following observation. A job that leaves a given queue can be described as a job that arrives at a successive queue. For example, a job that exits Queue C will arrive at Queue A. Therefore, we can connect the queues by adjusting the arrival rate of a queue based on the rate at which customers exit the queues that feed it. The exit rate becomes a result that is passed between queues in the connected model. However, note that the exit rate is not a reward variable defined in our single Erlang-2 queue model. We compute the exit rate from the utilization variable (called *busy*). Equation (1) shows how we calculated the job exit rate for a single queue based on the utilization and mean service time of each stage of the service station:

$$\begin{aligned} \text{EXIT\_RATE} &= \text{Utilization} * \text{ServiceRate} \\ &= \text{Utilization} * 1/\mu \\ &= \text{Utilization} * 1 / ((1/\lambda) + (1/\lambda)) \end{aligned} \tag{1}$$

This transformation demonstrates the use of the connection function construct to perform computation on results before they are used in another model. The exit rate represents all jobs leaving the queue. We need to determine the rate at which they arrive at successive queues. Note that all jobs leaving Queues B or C will be routed to Queue A. Therefore, the exit rate of each queue becomes an arrival rate at Queue A. Jobs leaving Queue A are routed to three places. A job may be passed to Queue B or Queue C, or it may exit the system. Our model divides the total exit rate of Queue A to reflect the rate at which customers will arrive at B or C. The arrival rate of jobs from Queue A to Queue B is defined as the total exit rate of Queue A multiplied by the probability of being routed to Queue B. The same computation is performed for Queue C.

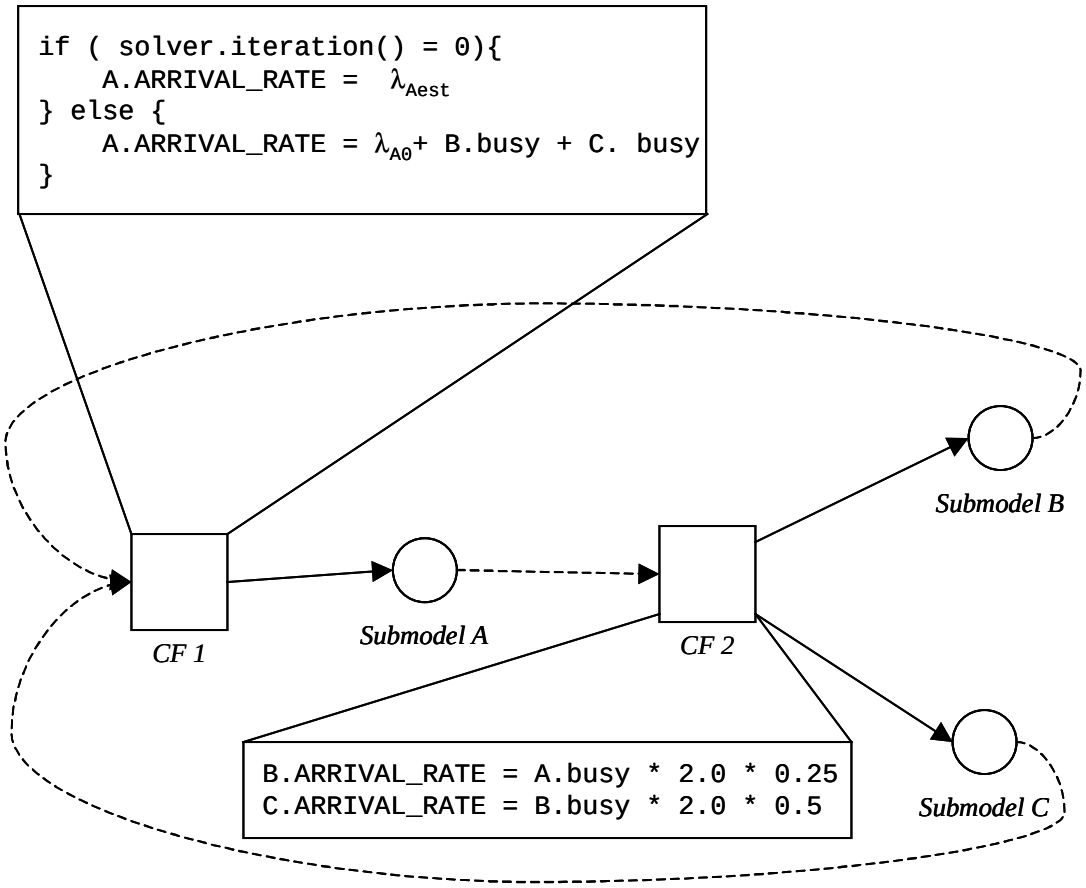


Figure 17. Connected Model of Erlang-2 Queue

4.4. Experiments

We conducted three experiments on the connected model. In doing so, we compare the connected model results to those of a single model of the entire network expressed using the SAN formalism. The single model is solved analytically using the Möbius state-space generator [14] and iterative steady-state solver when possible. State-based techniques become prohibitively slow and resource-consuming for large queue sizes; therefore, we use the Möbius Simulator [15] for large capacities. The single-queue submodels are solved exclusively with the state-space generator and iterative steady-state solver. Iteration in the connected model continues until all input results are exactly equal according to the precision of the machine on which we execute our solution or until the iteration count reaches a maximum value (8 for the first two experiments and 10 for the last one). We chose a high-precision convergence criterion because we wanted to observe as much convergence behav-

ior as possible given our time and resource constraints. The iterative steady-state solver uses Gauss-Seidel and an infinity difference norm of 10^{-9} as the stopping criterion. Simulation values were calculated to a 95% confidence level using a 1% relative confidence interval.

The chosen reward variables represent measures that may react differently to our connected model. We will relate each measure to the distribution of the queue length. The busy measure is either one or zero at any instant of time, and in steady-state is the probability that the server is busy. If the queue length is at least one, we know that the station is busy. The busy measure is thus the sum of the probabilities of all nonzero queue lengths. At a given point in time, even if the queue length is off by several jobs, the *busy* measure may still be correct. For this reason, *busy* has a rather high tolerance for error.

The queue length measure can tolerate some amount of error because it is an average. The distribution of the variable may have error in our connected model. The averaging operation is capable of filtering out minor differences between our model and the true model.

The full measure is also a binary value at a given point in time. It represents the probability that there is a maximum number of jobs in the queue. This variable is thus is a measure of a single point on the queue length distribution. This requires that our queue length distribution be accurate at a fine level. This measure is the least error-tolerant.

Each experiment varies one parameter. The default values for the other system variables are shown in Table 3.

Table 3. Default Experiment Values

Variable	Description	Default Value
μ_A	mean service rate for each stage of A	4.0
μ_B ,μ_C	mean service rate for each stage of B, C	2.0
λ_{A0}	mean arrival rate of new customers to A	1.0
λ_{Aest}	initial estimate of λ_A	1.0
N	maximum capacity of A, B, C	10

4.4.1. Varying maximum capacity

This experiment varies the maximum capacity of the queues (*N*). *N* does not include the job in service. The primary goal of this experiment is to explore the difference in state-space growth between the single and connected models.

Recall that the state space represents every possible combination of the state variables used to describe the status of the system at any given point in time. There are three variables for our single Erlang-2 queue. In particular, the state of a single Erlang-2 can be represented as the number of jobs in the queue and the number of jobs in each stage of the server. The number of jobs in the queue varies between zero and the maximum capacity, and the number of jobs in each stage of the server can be zero or one. Therefore, the state-space size for a maximum capacity *N* will have a state-space growth of *O(N)*. The single model representation of our queuing network requires three sets of the state variables needed for a single queue. This leads to a much higher $O(N^3)$ growth. We solved the single model using the analytic solver for capacities up to 40. The single system was solved with simulation for the 100, 500, and 1000 cases. Figure 18 illustrates the rapid state-space growth of the single model. The solution time of the iterative steady-state solver is directly related to the size of the state space. We see this relationship reflected in the exponential growth in execution time, shown in Figure 19.

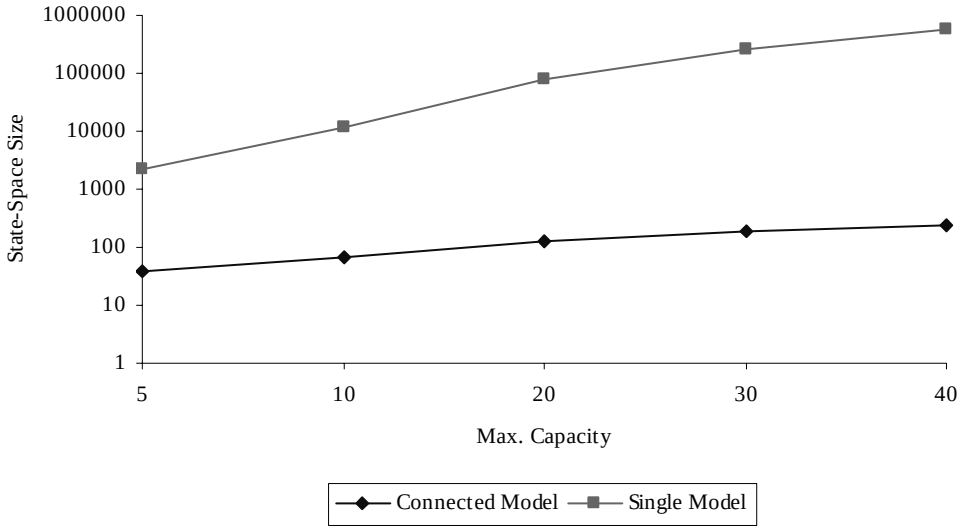


Figure 18. State-Space Size with Varying Queue Capacity

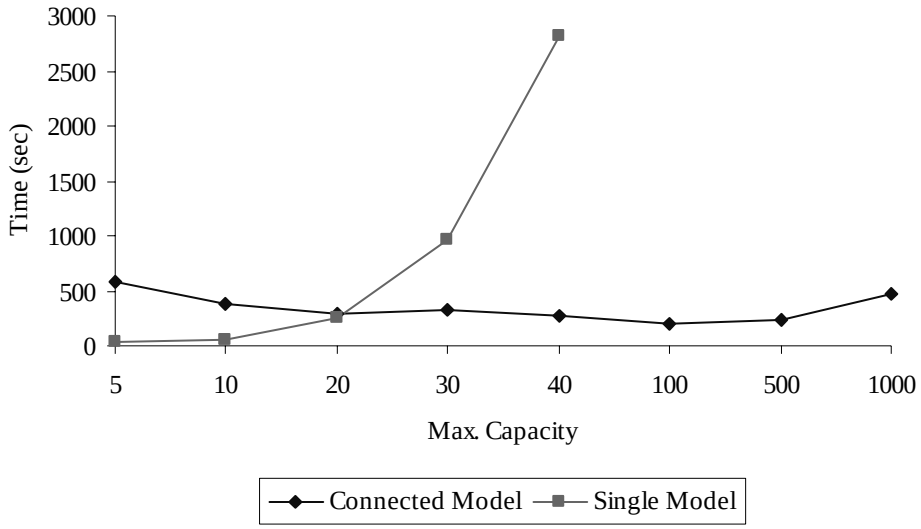


Figure 19. Total Solution Time with Varying Queue Capacity

We observed an interesting relationship between queue capacity and the rate of convergence of the connected model. This relationship is shown in Figure 20. In particular, the connected model converges faster as the queue capacity is increased. Note that 8 was the maximum number of iterations, and the connected model did not converge for capacities 5 and 10. We believe that this effect can be explained as follows. Recall that convergence is determined by the change in the input results to each model between iterations, and that the results passed in our connected model are based on the utilization (busy) measure. Also note that as queue capacity is increased, fewer incoming jobs are rejected. Therefore, as queue size is increased, more jobs enter the queue, and the server becomes busier for a given arrival rate. Increased capacity thus decreases the sensitivity of the *busy* (utilization) variable to changes in arrival rate. The arrival rate is the avenue by which a queue can affect another queue (recall that jobs are passed to the next queue through an increase in the arrival rate of that queue). The utilization is not sensitive to the added arrivals. Therefore, it changes very little during the next iteration, causing minimal changes in the following queues. This results in quicker convergence for the connected model.

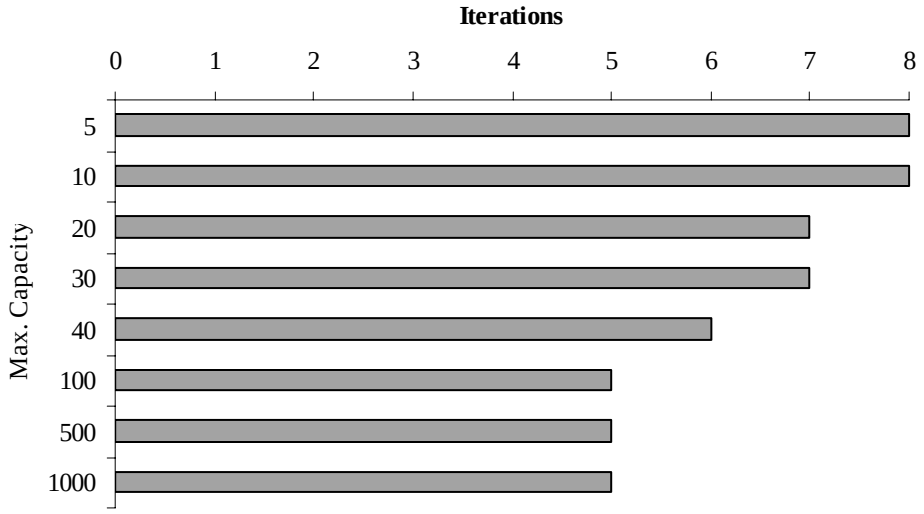


Figure 20. Connected Model Convergence with Varying Queue Capacity

We now examine the error introduced by decomposing the single model into three connected pieces. Recall that in the connected model, the arrival rate of Queue A is computed as the sum of the new job arrival rate and the exit rates of Queues B and C. Our Erlang-2 queue model assumes that the distribution of the arrival rate to each queue is exponential. This approximation introduces a source of error into our connected model. The error will be reflected in the reward variable solutions. We measured this effect using the percent error of the connected model solution as compared to the single model solution. The connected model results were compared to an analytic solution of the single model for capacities less than or equal to 40. The connected model was compared to simulation values of the single model for larger capacities. These percent errors are illustrated in Figure 21 and Figure 22 for the *busy*, *queue length*, and *full* measures. The values obtained from the connected model are generally very good. Most of the answers are within 5% of their true values.

As expected, the probability that the queue was full incurred greater error than the other measures. The Queue B percentage error is extremely high because the values are very small. The full probability is high enough to be measured in the first two experiments. The error for these experiments is approximately 100%. The extremely large error for the middle experiments is an example of error introduced due to insufficient accuracy of the solver.

Note that both the single and connected models are affected by this problem. The final experiments reported no data because the single model reported a zero value, making a percent error calculation impossible. We believe that the error measured in the Queue C jobs and full measures are related to statistical error present in the simulation model of the single system. Therefore, it is possible that larger capacities do not reduce the accuracy of the connected model.

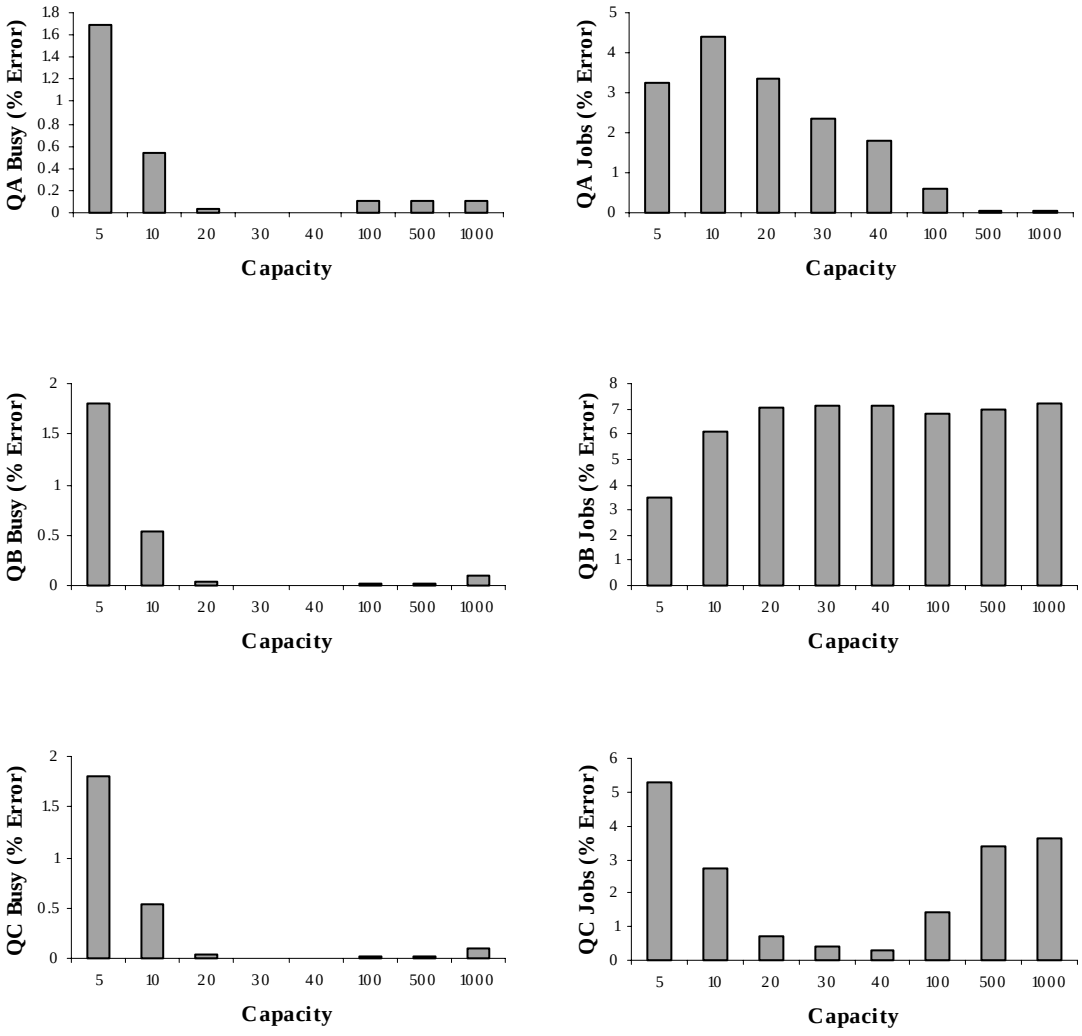


Figure 21. Percent Error of Connected Model with Varying Queue Capacity

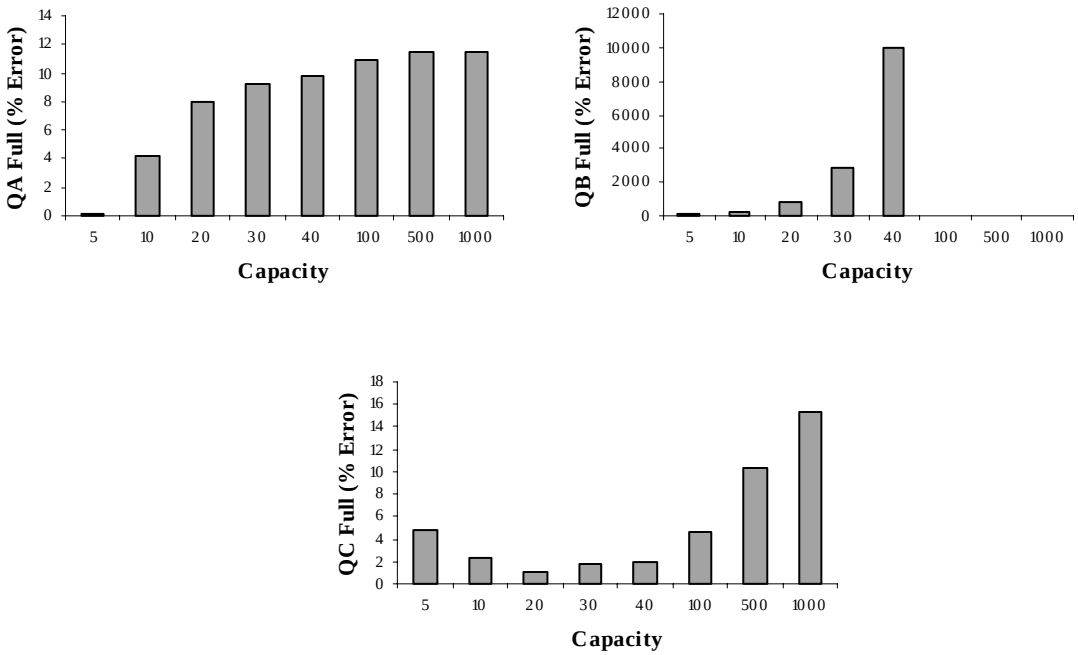


Figure 22. Percent Error of Connected Model with Varying Queue Capacity (cont.)

4.4.2. Varying new arrival rate

Our second experiment examined the effect of changing the rate at which new jobs enter the network.

Figure 23 shows that the convergence of the connected model is affected by the external job arrival rate. Our experiments with new arrival rates of 0.5 and 1 did not converge after the maximum number of iterations (8). As can be seen in the figure, the system converges more quickly for higher arrival rates. Recall that our iterative solution process must guess the arrival rate for Queue A for the first solution. Each successive solution is computed as the sum of the new arrival rate and the rate of jobs leaving the previous queues. We are solving for a steady-state solution. The model solution results indicate that Queue A will become a bottleneck in steady state for large new arrival rates. The average number of jobs in the queue is greater than 10 for new arrival rates of 2.5 or higher. The new job arrival rate affects the number of iterations required to make Queue A highly utilized. The arrival rate to Queue A is the sum of the new arrival rate and the arrival rate produced by the feedback process. A large new arrival rate increases the system convergence rate because it saturates

Queue A without waiting for the feedback process to increase the arrival rate slowly to its steady-state value.

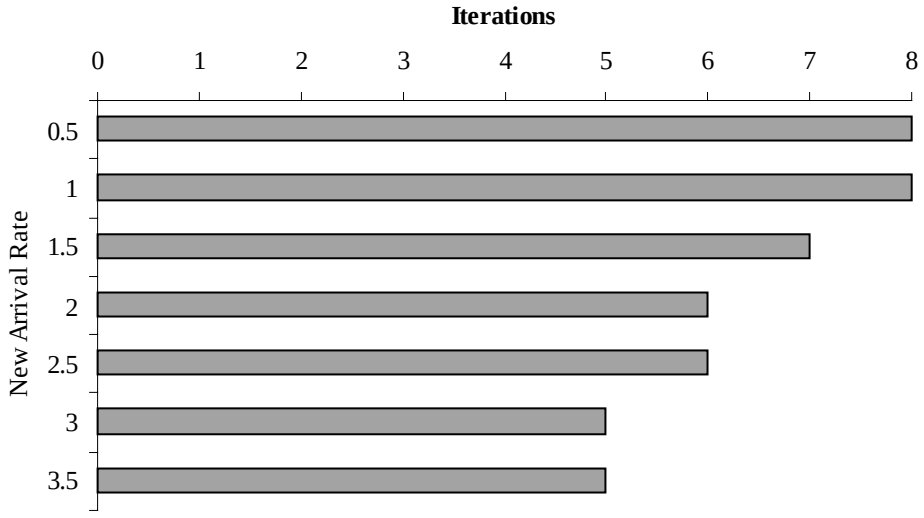


Figure 23. Connected Model Convergence with Varying New Arrival Rate

The connected model performs well in this experiment, as shown in Figure 24 and Figure 25. The values of the three reward variables we studied generally lie within 5% of the single model's values.

4.4.3. Varying Queue A arrival rate initial guess

Our third experiment examines the effects of varying the initial guess for the arrival rate to Queue A. Queue A is common to both solution cycles in our model. Therefore, we chose to solve it first. The results from this solution are used to solve Queues B and C. The results from these two solutions are used to solve Queue A again. This process continues until the answers become equal within the precision of the machine. We will look at the effects of the initial guess on the number of iterations to convergence. We will also look at the convergence of each measure we computed as a function of the number of iterations of the connected model.

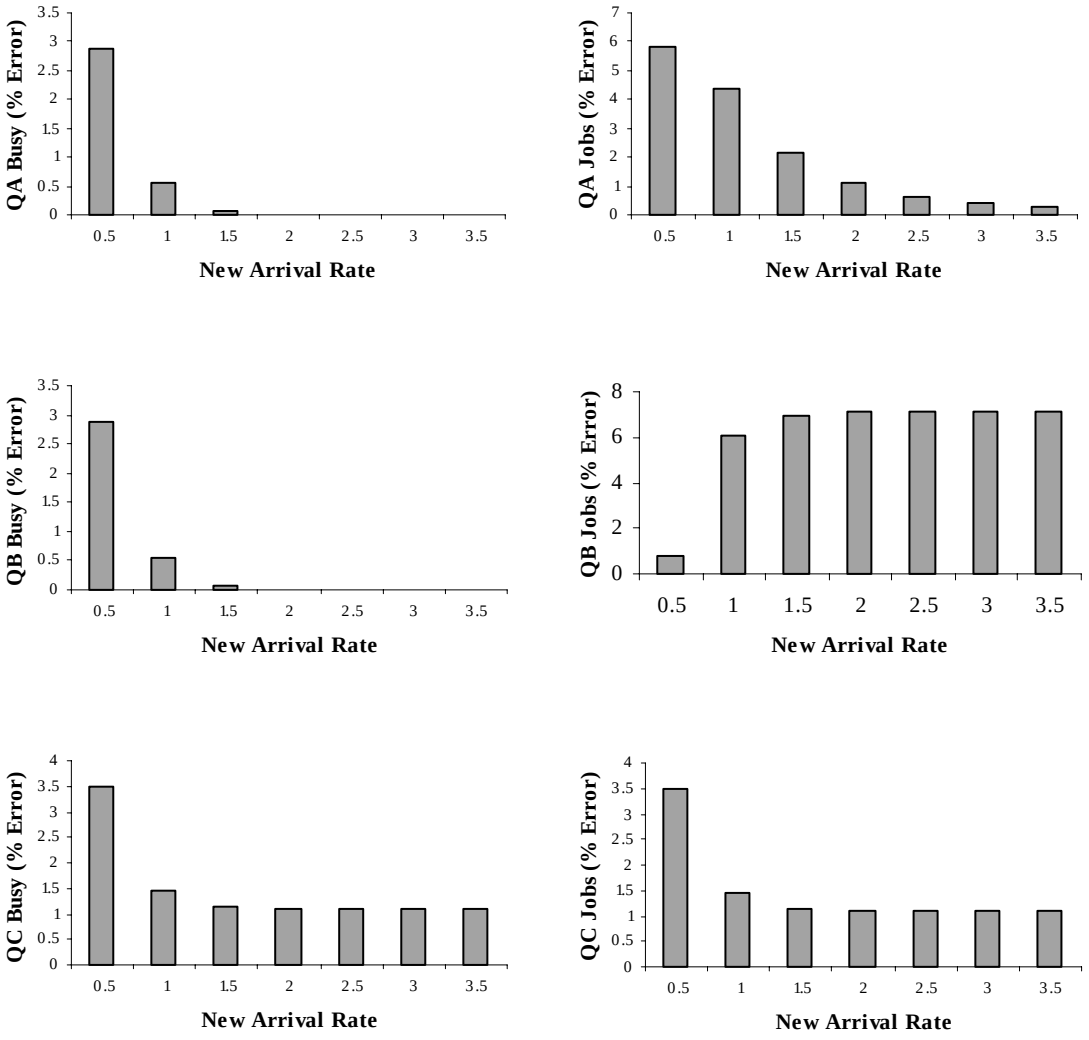


Figure 24. Connected Model Accuracy with Varying New Arrival Rate

Figure 26 shows that the initial guess does indeed have an effect on the number of iterations required to solve the connected model. We know that the arrival rate of Queue A is the sum of the new arrival rate and the *busy* variables of Queues B and C. Convergence occurs when the passed results become constant. Choosing an initial guess for the arrival rate to Queue A based on the sum of the new arrival rate and the modeler's best guess for the steady-state values for Queues B and C should start the system in a state that is very close to convergence. Therefore, the number of iterations will be fewer if this value can be guessed

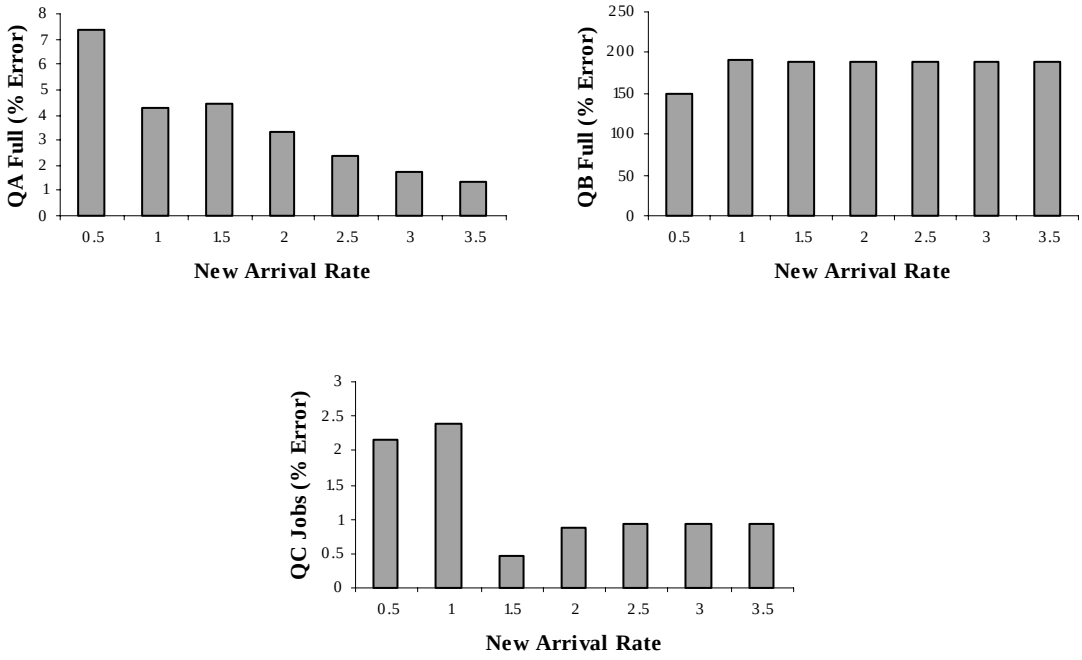


Figure 25. Connected Model Accuracy with Varying New Arrival Rate (cont.)

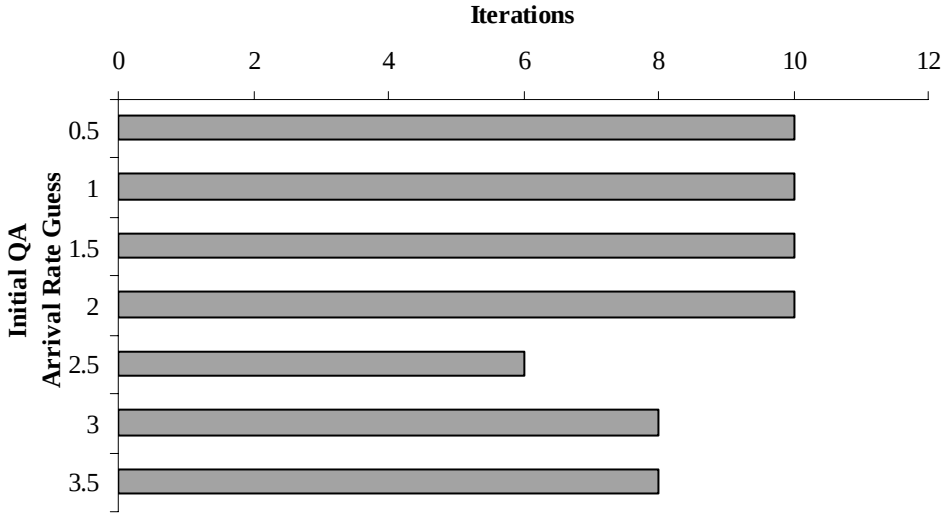


Figure 26. Connected Model Convergence with Varying Queue A Arrival Rate Estimate

more accurately. We observe this phenomenon in our model. The steady-state values of the *busy* variables in Queues B and C are approximately 0.5 and 1.0, respectively. Therefore, the arrival rate of Queue A, which is the sum of the new arrival rate (1.0) and the *busy*

measures for Queues B and C (0.5 and 1.0), stabilizes at around 2.5. We see that an initial guess of 2.5 yields the fastest convergence.

Figure 27 and Figure 28 show the effects of the initial guess on the convergence rate of each reward variable. In steady state there is a balance between the exit and arrival rates of each queue. We know that the arrival rate of Queue A is approximately 2.5 in steady state. An initial estimate greater than 2.5 starts the iterative process by solving Queue A with a larger arrival rate than the system will have in steady state. The exit rate for Queue A will also be higher, causing Queues B and C to be solved with elevated arrival rates. At the start of the second iteration, the estimate for the arrival rate of Queue A is replaced by the appropriate calculation. The effects of the initial guess are slowly removed by the feedback process. The converse is true for a low initial estimate. A low estimate causes the feedback portion of Queue A's arrival rate calculation to start much lower than its steady-state value. This effect will be overcome after several iterations. We see these trends in the convergence behavior of the reward variables. The first four series in the graph initialize the queues with arrival rates that are lower than the steady-state values. The *busy*, *jobs*, and *full* measures are lowered accordingly. The fifth series starts the system very close to steady state; therefore, it converges quickly. The final four series illustrate an estimate that starts the system with arrival rates that are higher than in steady state. We can see that the jobs and full measures reflect this trend. The trend is also present in the *busy* variable. However, it is more difficult to observe on the graph, because the difference between the initial calculation and steady state is less than 10^{-3}.

4.5. Conclusions

Our case study demonstrates that Möbius can be used to solve a cyclic, connected model. Our example demonstrates many of the constructs supported by the proposed connection extensions to the abstract functional interface. The results database collects results for each solution performed in a cyclic model. This capability can be used to observe the values of reward variables used in the connection process, facilitating understanding of the convergence or divergence of the system. This information can provide insight on a particular connected model. It may also be used to build connection solvers capable of intelligently resetting a system that is not converging with the initial guess that was chosen.

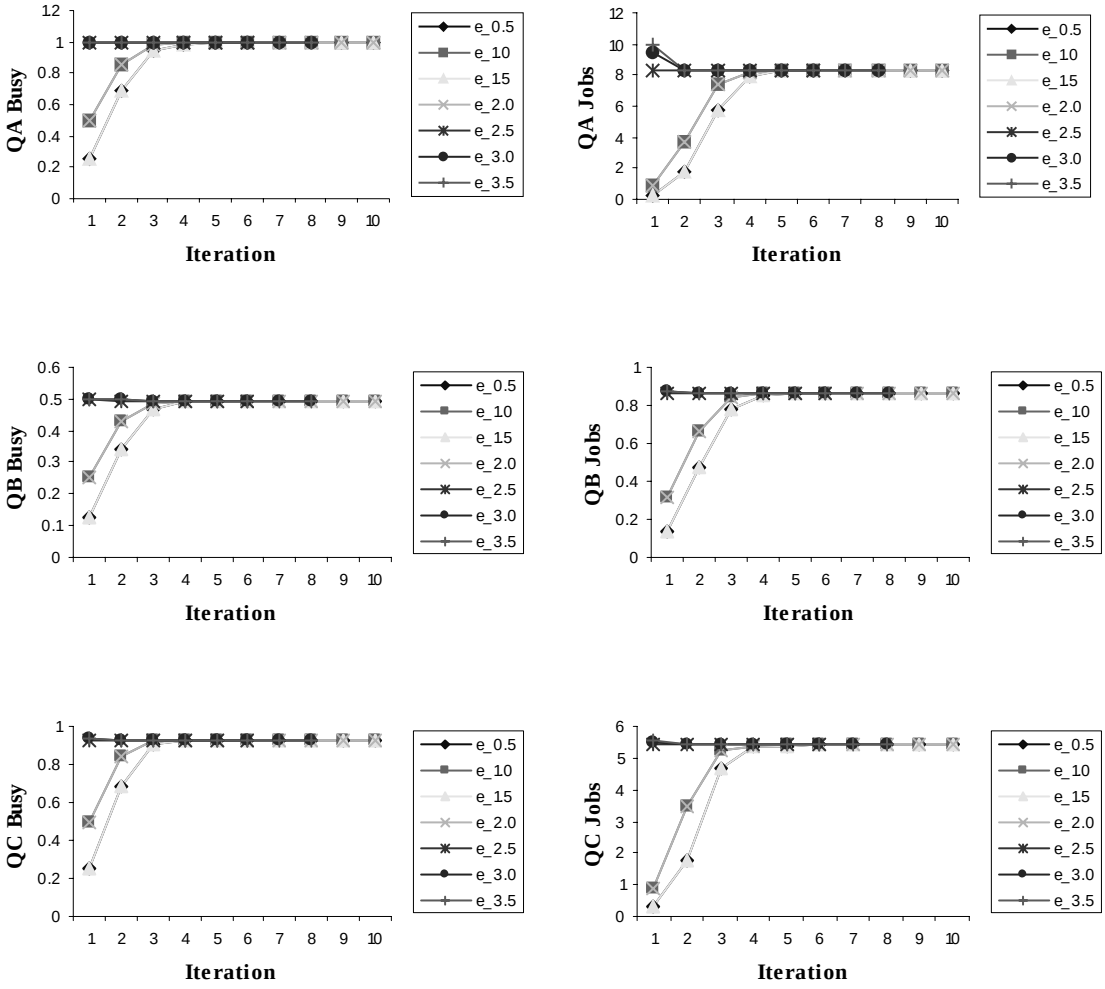


Figure 27. Connected Model Measure Convergence with Varying Queue A Arrival Rate Estimate

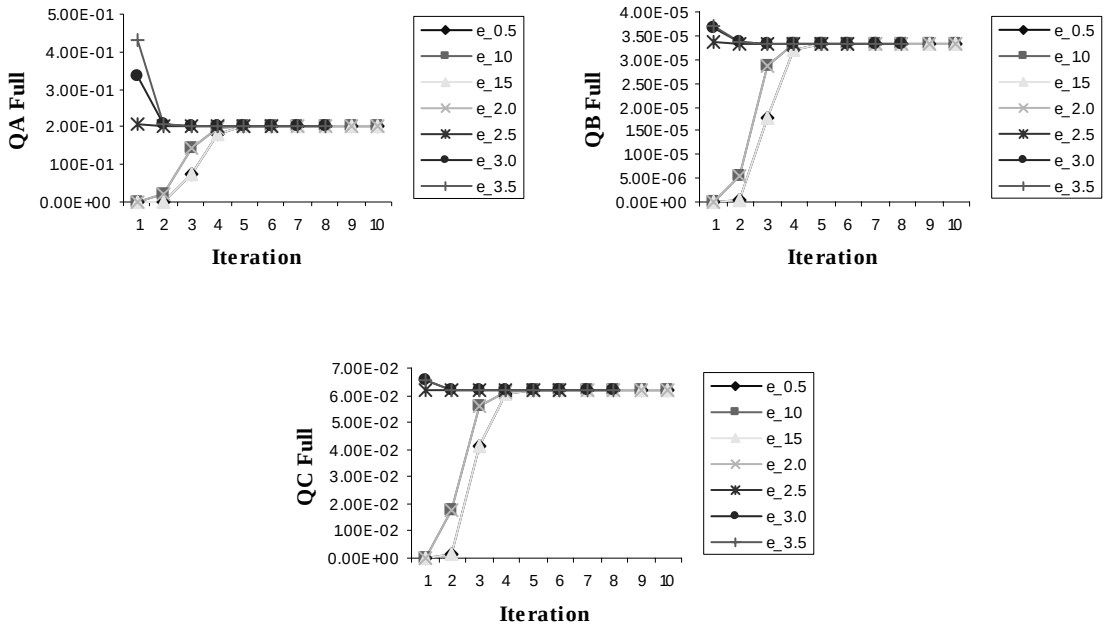


Figure 28. Vary Arrival Rate Estimate Convergence (cont.)

5. CONCLUSION AND FUTURE RESEARCH

We have introduced an infrastructure with Mobius for developing connected models. The results database is the foundation of the infrastructure. It is implemented in two layers. The first layer, the *virtual database*, uses a commercial database to implement a generic database interface. The second layer, the *Möbius access layer*, defines the structures and interface used to store modeling results. Results are stored in a flexible format that will allow each solver to define a structure for storing its settings and solution values. Möbius solvers write all model solutions into the database. A complete record of the project's state, including solver settings and model version numbers, is also recorded. This information is accessible through a Java or C language implementation of the Möbius access layer. Our layered design facilitates integration with multiple commercial databases.

We have also defined a software architecture for building connection formalisms and solvers. We proposed new constructs for building connected models, and extensions to the abstract functional interface to support them. We implemented a prototype connected model using the Möbius tool and performed a case study of a connected model.

Future research projects will use the database to analyze results or build connected models. The design of experiments editor, currently under development, will read results from the database and perform regression analysis to study model sensitivity to global variables and determine values that will optimize a given performance variable. A comprehensive GUI-based browser for viewing and exporting results will also be created. Connection formalisms and solvers will be developed. Also, support for additional database packages may be added through an implementation of the virtual database interface, using the interfaces provided by the database vendor.

APPENDIX A. VIRTUAL DATABASE QUERY LANGUAGE TUTORIAL

This appendix will illustrate the query language by building queries for an example database. The database is a simple model of a school. The Milliwatt Engineering Graduate School has a group of students and several classes. The classes are Introduction to Homework I and II (ENGR401, ENGR402), Advanced Placement Homework (ENGR403), Random Processes (EE500), and Thesis Preparation (EE600). Random Processes requires a prerequisite. A student may elect to take both semesters of Introduction to Homework or one semester of Advanced Placement Homework to fulfill the prerequisite. There are three students in the school.

We use three templates in the database (see Figure 29. Template Definitions). The School template is the root. It contains the name of the school and student and class information. The StudentInfo template contains the name and id number of each student. The ClassInfo template records the name of the class and a grade for each student (stored as a percentage). A percent less than zero indicates that a student has not taken that class. Our database has one School record called *s*. Next, we will show several example queries based on the example school and class records in Tables 4 and 5.

Type	Name	Size	Description
varchar	name	0	school name
StudentInfo	student	3	student information
ClassInfo	class	5	class information

School Template

Type	Name	Size	Description
varchar	name	0	student name
varchar	ssn	0	social security #

StudentInfo Template

Type	Name	Size	Description
varchar	name	0	class name
varchar	id	0	catalog id
float8	student	3	final grades

ClassInfo Template

Figure 29. Template Definitions

Table 4. Primitive Fields of Entity *s*

Field Name	**Value**
name	Milliwatt
student[0].name	Amanda
student[0].ssn	123456789
student[1].name	Andy
student[1].ssn	999446633
student[2].name	Randy
student[2].ssn	875324421

Table 5. Contents of the *class* Subrecord

	class[0]	**class[1]**	**class[2]**	**class[3]**	**class[4]**
name	Introduction to Homework I	Introduction to Homework II	Advanced Placement Homework	Random Processes	Thesis Preparation
id	ENGR401	ENGR402	ENGR403	EE500	EE600
student[0]	88.4	82.1	-1.0	76.2	-1.0
student[1]	-1.0	-1.0	97.3	85.5	98.0
student[2]	68.9	91.2	-1.0	80.1	-1.0

Example 1. Find the name of the course EE500.

Return Variables: `class{id = 'EE500'}.name`

Target List: `s`

Expression: `true`

*Return DataSet (expressed as **name** value pairs):*

`class{id = 'EE500'}.name`	`'Random Processes'`
`vdb_record_name`	`'s'`

This is a trivial query. We search the record *s* based on the expression *true*, which will indicate a match for every record in the target list. The information that we need can be described using the associative look-up naming convention. This information could also be retrieved using the `readRecord` method with the name *s.class{id = 'EE500'}.name*.

Example 2. Find the name of every engineering (ENGR) course.

Return Variables: `@class.name`

Target List: `School`

Expression: `$@class.name ~ 'ENGR%'`

Return DataSet (expressed as name value pairs):

```
val_field_0          'Introduction to Homework I'
name_field_0         'class[0].name'
index_0              0
vdb_record_name      's'

val_field_0          'Introduction to Homework II'
name_field_0         'class[1].name'
index_0              1
vdb_record_name      's'

val_field_0          'Advanced Placement Homework'
name_field_0         'class[2].name'
index_0              2
vdb_record_name      's'
```

This query uses array explosion to repeat a query for every element of an array. We search all records of template School (note that we only have one) returning the name field of every class if its id begins with the letters *ENGR*. The dollar sign represents the name of the record currently being queried. The @ causes the query to be executed five times on the record *s*. `@class` is replaced by class[0] on the first pass followed by class[1] to class[4].

Example 3. Find the classes that Randy has taken.

Return Variables: `@class.id`

Target List: `School`

Expression: `$@class.student<<$student{name='Randy'}>> >= 0.0`

Return DataSet (expressed as name value pairs):

```
val_field_0          'ENGR401'
name_field_0         'class[0].name'
index_0              0
vdb_record_name      's'

val_field_0          'ENGR402'
name_field_0         'class[1].name'
index_0              1
vdb_record_name      's'

val_field_0          'EE500'
name_field_0         'class[3].name'
index_0              3
vdb_record_name      's'
```

This query demonstrates a dereferenced associative index. The student's name is not included in every class record. We do know that the array index for information about Randy is the same in both arrays. Therefore, we can compute Randy's index using his name ($student{name='Randy'}) and use it to index the grade information (<<$student{name='Randy'}>>). Dereferencing brackets (< >) must be doubled inside query expressionsto avoid a conflict with the greater than and less than operators. The expression looks at Randy's grade for each class and returns true if the value is non-negative, indicating that he has completed the class.

Example 4. Find all students with a failing grade (below 70%).

Return Variables: `@student.name, @class.id, @class.@student`

Target List: `s`

Expression:

```
($@class.@student >= 0.0) & ($@class.@student < 70.0)
```

Return DataSet (expressed as name value pairs):

val_field_0	'Randy'
name_field_0	'student[2].name'
val_field_1	'ENGR401'
name_field_1	'class[0].name'
val_field_2	68.4
name_field_2	'class[0].student[2].name'
index_0	2
index_1	0
vdb_record_name	's'

This query uses two array explosions to search all of the grades in all of the classes. The query expression is evaluated once for every permutation of the student and class array indices. Note the extra information that is provided for each return field and exploded array. This query illustrates the array explosion definition of an array. The expression @students is evaluated as a single virtual array even though it represents two real arrays (one stores student information and the other stores grades). We assume that all arrays of the same name have the same size and are related to each other. Therefore, they are exploded as if they were the same array. It is easy to retrieve the name that corresponds to a particular grade. The assumption supports our space-saving technique of defining a result once and saving its values in arrays of the same name where the position of a particular results declaration and value are kept identical.

Example 5. Find all students who have met the Random Processes prerequisite

Return Variables: `@student.name`

Target List: `s`

Expression:

```
(($class{id='ENGR401'}.@student >= 0.0) &
($class{id='ENGR402'}.@student >= 0.0)) |
($class{id='ENGR403'}.@student >= 0.0)
```

Return DataSet (expressed as name value pairs):

```
val_field_0          'Amanda'
name_field_0         'student[0].name'
index_0              0
vdb_record_name      's'

val_field_0          'Andy'
name_field_0         'student[1].name'
index_0              1
vdb_record_name      's'

val_field_0          'Randy'
name_field_0         'student[2].name'
index_0              2
vdb_record_name      's'
```

This query is evaluated once for each student. It is true if the student has a non-negative grade for both ENGR401 and ENGR402 or for ENGR403. We use an associative index to retrieve the grades from the classes in which we are interested.

APPENDIX B. VIRTUAL DATABASE USING POSTGRE SQL

Postgre SQL [8] is a free SQL database package available for many operating systems including Solaris, HPUX, and Linux. It is distributed under the Berkeley License Agreement. The PSQL implementation of the virtual database uses several tables to track templates and records. A template is composed of fields that can be of primitive data types or other templates. Each record becomes a row in the template table. A special varchar column named *vdb_record_name* keeps track of the name of each record instance. A template is represented as a table with one column for each primitive field. Each subrecord becomes a record in the template table for its type. Square brackets are reserved characters in PSQL. We translate array names into a format composed of the base name, two underscores, and the index number (i.e., myarray__0). All operators in our query language have one-to-one mappings with standard SQL notations. Commercial database interfaces are all unique, since vendors focus on particular markets and develop rich sets of useful extensions. We chose to use standard SQL commands whenever possible to maximize the portability of our implementation.

Several tables are used to keep track of template and record definitions. The template table stores the name and description of every available template and a unique integer, called the *template identifier* (TID), key. The field table records the type and name of each field categorized by the TID. A record table keeps track of the name and description of each record. Subrecords are considered to be records at this level. A column is use to record the size of each subrecord array.

Type	Name	Size
int4	A1	1
varchar	A2	2

Template A

Type	Name	Size
float8	B1	1
A	B2	2

Template B

varchar vdb_record_name	int4 A1	varchar A2__0	varchar A2__1
myrecB.B2__0	-	-	-
myrecB.B2__1	-	-	-

Table vdb_A

varchar vdb_record_name	float8 B1
myrecB	-

Table vdb_B

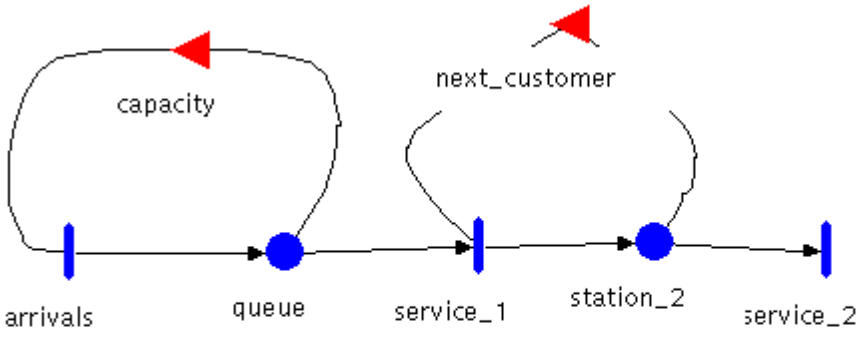
Figure 30. Record Storage in PSQL

APPENDIX C. SAN MODEL OF ERLANG-2 QUEUE NETWORK

This appendix contains a description of the models used in Chapter 4. The documentation was produced by the Möbius documentor.

Single Erlang-2 Queue Model Documentation

San Model: q



Place Attributes:

Name	Initial Marking
station_2	0
queue	0

Timed Activity Attributes: : service_2

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : service_1

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : arrivals

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** ARRIVAL_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Instantaneous Activities Without Cases: :

Activity Name

Input Gate Attributes: next_customer

Field Name	Field Value
Predicate	station_2-Mark() == 0
Function	;

Input Gate Attributes: capacity

Field Name	Field Value
Predicate	queue-Mark() + station_2-Mark() < MAX_CAPACITY + 1
Function	;

Performance Variables on: pv

Performance Variables on : pv

Top Level Model Information Model

Model Name q

Model Type SAN Model

Include File Paths
/home/crhc2/amyc/Möbius_project/erlang_q/Atomic/q/qSAN.h

Compilation Information

Model File Name	pvPVModel	
Node File Name	pvPVNodes	
Archive Name	pvPV.a	
BaseClasses Directory		
/home/crhc2/amyc/work/Möbius/Cpp/Performance_Variables/		

Performance Variable : busy

Affecting Models q

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
if( (q-queue-Mark() > 0) || (q-station_2-Mark() > 0)){
  return 1.0;
}
```

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	10.0
		Batch Size	10.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : jobs

Affecting Models q

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
return (q-queue-Mark() + q-station_2-Mark());
```

Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : full

Affecting Models q

Impulse Functions

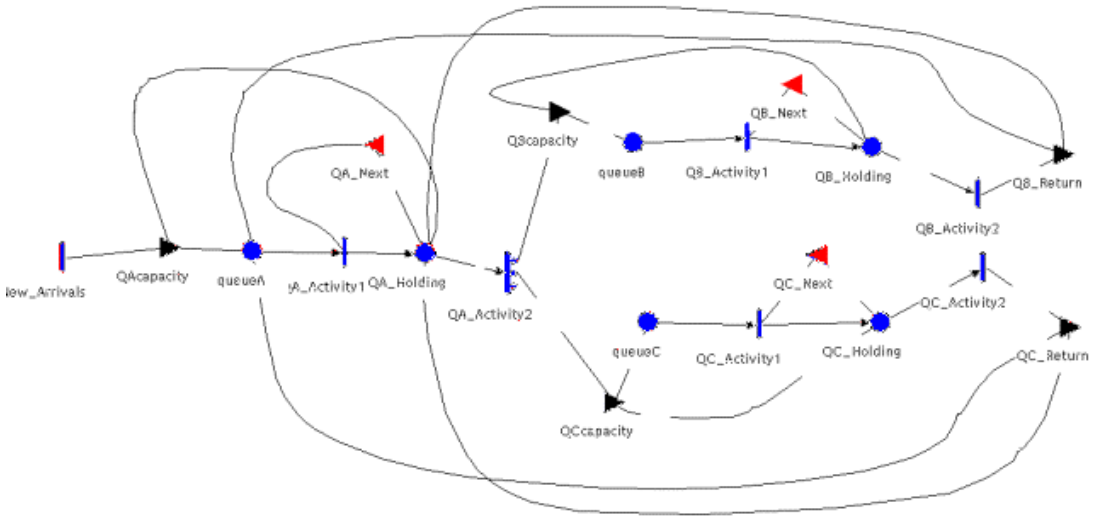
Reward Function *(Reward is over all Available Models)*

```
if( q-station_2-Mark() + q-queue-Mark() == MAX_CAPACITY + 1 ){
```

```
        return 1.0;
    }
```

Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Single Model of Erlang-2 Queue Network



Place Attributes:

Name	Initial Marking
queueC	0
queueB	0
QC_Holding	0
QB_Holding	0
QA_Holding	0
queueA	0

Timed Activity Attributes: : QA_Activity2

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** (STATION_SERVICE_RATE) * 2.0
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	***case 1*** QB_PROB ***case 2*** QC_PROB ***case 3*** EXIT_PROB

Timed Activity Attributes: : QC_Activity2

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : QC_Activity1

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : QB_Activity2

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : QB_Activity1

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** STATION_SERVICE_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : QA_Activity1

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** (STATION_SERVICE_RATE * 2.0)
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity Attributes: : New_Arrivals

Field Name	Field Value
Exponential Distribution Parameters	***Rate*** QA_NEW_ARRIVAL_RATE
Activation Predicate	(none)
Reactivation Predicate	(none)

Instantaneous Activities Without Cases: :

Activity Name

Input Gate Attributes: QC_Next

Field Name	Field Value
Predicate	QC_Holding-Mark() == 0
Function	;

Input Gate Attributes: QB_Next

Field Name	Field Value
Predicate	QB_Holding-Mark() == 0
Function	;

Input Gate Attributes: QA_Next

Field Name	Field Value
Predicate	QA_Holding-Mark() == 0
Function	;

Output Gate Attributes: QCcapacity

Field Name	Field Value

Function	// drop jobs if queue is full
	if((queueC-Mark() + QC_Holding-Mark())< MAX_CAPACITY + 1)
	Mark()++;
	}

Output Gate Attributes: QBcapacity

Field Name	Field Value

Function	// drop jobs if queue is full
	if((queueB-Mark() + QB_Holding-Mark())< MAX_CAPACITY + 1)
	Mark()++;
	}

Output Gate Attributes: QC_Return

Field Name	Field Value

Function	if ((queueA-Mark() + QA_Holding-Mark())< MAX_CAPACITY + 1)
	Mark()++;
	}

Output Gate Attributes: QB_Return

Field Name	Field Value

Function	if ((queueA-Mark() + QA_Holding-Mark())< MAX_CAPACITY + 1)
	Mark()++;
	}

Output Gate Attributes: QAcapacity

Field Name	Field Value

Function	// if there is room in the queue put
	// the job there
	if((queueA-Mark() + QA_Holding-Mark())< MAX_CAPACITY + 1)
	Mark()++;
	}

Performance Variables on: pv_system

Performance Variables on : pv_system

Top Level Model Information

Model Name	q_system
Model Type	SAN Model

Compilation Information

Model File Name	pv_systemPVModel
Node File Name	pv_systemPVNodes
Archive Name	pv_systemPV.a

Performance Variable : QAbusy

Affecting Models q_system

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
// busy if there is a token at either station

if(  (q_system-queueA-Mark() > 0)  || (q_system-QA_Holding-Mark() >  0)){
                return 1.0;
                }
```

Simulator Statistics	Type	Steady State
Options	Estimate Mean	
	Include Lower Bound on Interval Estimate	
	Include Upper Bound on Interval Estimate	
	Estimate out of Range Probabilities	
	Confidence Level is Relative	
Parameters	Initial Transient	10000.0
	Batch Size	1000.0
Confidence	Confidence Level	0.95
	Confidence Interval	0.01

Performance Variable : QBbusy

Affecting Models q_system
Impulse Functions
Reward Function *(Reward is over all Available Models)*

```
// busy if there is a token at either station

if(  (q_system-queueB-Mark()  > 0)  || (q_system-QB_Holding-Mark() > 0)){
  return 1.0;
}
```

Simulator Statistics	Type	Steady State
Options	Estimate Mean	
	Include Lower Bound on Interval Estimate	
	Include Upper Bound on Interval Estimate	
	Estimate out of Range Probabilities	
	Confidence Level is Relative	
Parameters	Initial Transient	10000.0
	Batch Size	1000.0
Confidence	Confidence Level	0.95
	Confidence Interval	0.01

Performance Variable : QCbusy

Affecting Models q_system
Impulse Functions
Reward Function *(Reward is over all Available Models)*

```
// busy if there is a token at either station

if(  (q_system-queueC-Mark() >  0)  || (q_system-QC_Holding-Mark() > 0)){
  return 1.0;
}
```

Simulator Statistics	Type	Steady State
Options	Estimate Mean	
	Include Lower Bound on Interval Estimate	

		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	10000.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : QAjobs

Affecting Models q_system
Impulse Functions
Reward Function *(Reward is over all Available Models)*

```
return(q_system-queueA-Mark() + q_system-QA_Holding-Mark());
```

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	10000.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : QBjobs

Affecting Models q_system
Impulse Functions
Reward Function *(Reward is over all Available Models)*

```
return(q_system-queueB-Mark() + q_system-QB_Holding-Mark());
```

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	10000.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : QCjobs

Affecting Models q_system
Impulse Functions
Reward Function *(Reward is over all Available Models)*

```
return(q_system-queueC-Mark() + q_system-QC_Holding-Mark());
```

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	

	Include Upper Bound on Interval Estimate	
	Estimate out of Range Probabilities	
	Confidence Level is Relative	
Parameters	Initial Transient	10000.0
	Batch Size	1000.0
Confidence	Confidence Level	0.95
	Confidence Interval	0.01

Performance Variable : QAfull

Affecting Models q_system

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
if( (q_system-queueA-Mark() + q_system-QA_Holding-Mark()) == (MAX_CAPACITY + 1) ){
        return 1.0;
        }
```

Simulator Statistics	Type	Steady State
Options	Estimate Mean	
	Include Lower Bound on Interval Estimate	
	Include Upper Bound on Interval Estimate	
	Estimate out of Range Probabilities	
	Confidence Level is Relative	
Parameters	Initial Transient	10000.0
	Batch Size	1000.0
Confidence	Confidence Level	0.95
	Confidence Interval	0.01

Performance Variable : QBfull

Affecting Models q_system

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
if( (q_system-queueB-Mark() + q_system-QB_Holding-Mark() )== (MAX_CAPACITY + 1) ){
  return 1.0;
        }
```

Simulator Statistics	Type	Steady State
Options	Estimate Mean	
	Include Lower Bound on Interval Estimate	
	Include Upper Bound on Interval Estimate	
	Estimate out of Range Probabilities	
	Confidence Level is Relative	
Parameters	Initial Transient	10000.0
	Batch Size	1000.0
Confidence	Confidence Level	0.95
	Confidence Interval	0.01

Performance Variable : QCfull

Affecting Models q_system

Impulse Functions

Reward Function *(Reward is over all Available Models)*

```
if( (q_system-queueC-Mark() + q_system-QC_Holding-Mark()) == (MAX_CAPACITY + 1)){
  return 1.0;
}
```

Simulator Statistics		Type	Steady State
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	10000.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

REFERENCES

[1] D. Daly, D. Deavours, J. Doyle, A. Stillman, P. Webster, and W. H. Sanders, "Möbius: An extensible framework for performance and dependability modeling," in *Tool Descriptions from the Multi-Workshop on Formal Methods in Performance Evaluation and Applications*, Zaragoza, Spain, September 6-10, 1999.

[2] D. D. Deavours, "The Möbius framework," Ph.D. dissertation, manuscript in preparation.

[3] J. M. Doyle, "Abstract model specification using the Möbius modeling tool," M.S. thesis, University of Illinois at Urbana-Champaign, 2000.

[4] R. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Boston: Kluwer Academic Publishers, 1996.

[5] G. Ciardo (private communication), Feb.17, 2000.

[6] V. S. Adve et al., "POEMS: End-to-end performance design of large parallel adaptive computational systems," *IEEE Transactions on Software Engineering, Special Issue on Software and Performance*, to be published.

[7] P. Webster, "Experimental design in the Möbius framework," M.S. thesis, manuscript in preparation.

[8] Postgre SQL, *http://www.postgresql.org.*

[9] Sun Microsystems, "JDBC API," *http://java.sun.com/products/jdk/1.2/docs/guide/jdbc.*

[10] J. Hoffman, "Introduction to structured query language," *http://w3.one.net/~jhoffman/sqltut.htm.*

[11] R. G. G. Cattell and D. K. Barry (Eds.), *The Object Data Standard: ODMG 3.0*. San Fransisco: Morgan Kaufmann Publishers, 2000.

[12] W. Whitt, "The queueing network analyzer," *The Bell System Technical Journal*, vol. 62, no. 9, pp. 2779-2815, 1983.

[13] R. Sadre and B. Haverkort, "FiFiQueues: Fixed-point analysis of queueing networks with finite-buffer stations," in *Proceedings of the 11th International TOOLS Conference*, Schaumburg, IL, USA, March 27-31, 2000, pp. 324-327.

[14] J. M. Sowder, "State-space generation techniques in the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1998.

[15] A. Williamson, "Discrete-event simulation in the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1998.